OpenSPARC™ T2 System-On-Chip (SOC) Microarchitecture Specification

Sun Microsystems, Inc.
www.sun.com

Part No. 820-2620-11
December 2007, Revision A

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

Contents

Figures

Tables

Preface

This *OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification* includes detailed functional descriptions of the OpenSPARC T2 System-on-Chip I/O components.

This manual also provides I/O signal list for each component. This processor expands Sun's throughput computing initiative by doubling the number of threads from the OpenSPARC T1 processor and adding support for industry standard I/O interfaces like PCI-Express and 10Gigabit Ethernet.

How This Document Is Organized

Chapter 1 describes the overall OpenSPARC T2

Chapter 2 describes the L2 Cache

Chapter 3 describes the Memory Control Unit (MCU)

Chapter 4 describes the Test Control Unit (TCU)

Chapter 5 describes the Clock Control Unit (CCU)

Chapter 6 describes System Interface Unit (SIU)

Chapter 7 describes the Non-Cacheable Unit (NCU)

Chapter 8 describes the Data Management Unit (DMU)

Chapter 9 describes the Miscellaneous I/O (MIO)

Chapter 10 describes the Debug Functions

Chapter 11 describes the Electronic Fuse Unit (EFU)

Chapter 12 describes the Reset Functions

Using UNIX Commands

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

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

  http://docs.sun.com

Shell Prompts

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

Typographic Conventions

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

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

Related Documentation

The documents listed as online are available at:

http://www.opensparc.net/

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

Documentation, Support, and Training

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

Third-Party Web Sites

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

Sun Welcomes Your Comments

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

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

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

OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification, part number 820-2620-10.

CHAPTER 1

OpenSPARC T2 Basics

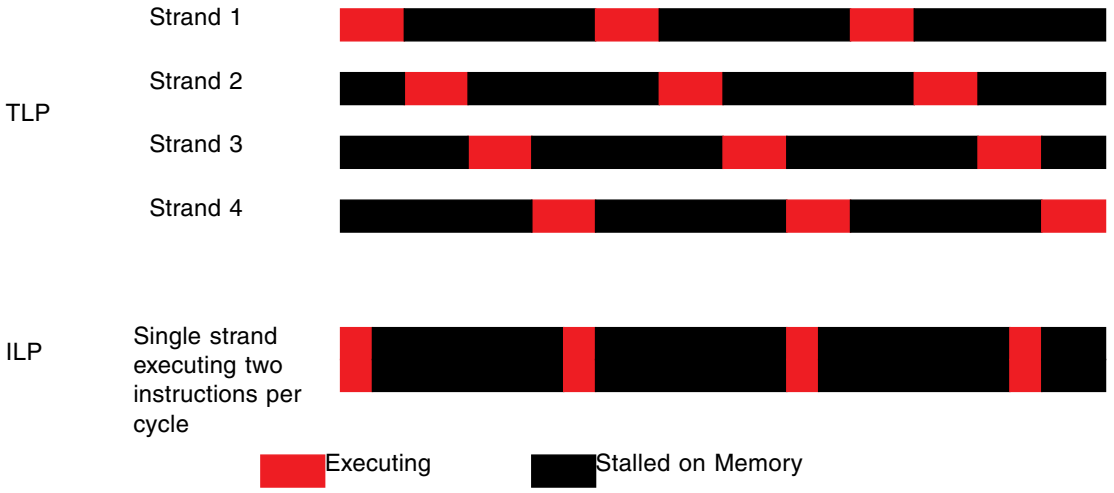
1.1 Background

OpenSPARC T2 is the follow-on chip multi-threaded (CMT) processor to the highly successful OpenSPARC T1 processor. The product line fully implements Sun's Throughput Computing initiative for the horizontal system space. Throughput Computing is a technique that takes advantage of the thread-level parallelism that is present in most commercial workloads. Unlike desktop workloads, which often have a small number of threads concurrently running, most commercial workloads achieve their scalability by employing large pools of concurrent threads.

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

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

FIGURE 1-1 Differences Between TLP and ILP



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

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

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

strand capable of executing only a single instruction per cycle is fully capable of completing tasks in the time required by the application, making this disadvantage a nonissue for nearly all commercial applications.

1.2 OpenSPARC T2 Overview

OpenSPARC T2 is a single chip multi-threaded (CMT) processor. OpenSPARC T2 contains eight SPARC physical processor cores. Each SPARC physical processor core has full hardware support for eight strands, two integer execution pipelines, one floating-point execution pipeline, and one memory pipeline. The floating-point and memory pipelines are shared by all eight strands. The eight strands are hard-partitioned into two groups of four, and the four strands within a group share a single integer pipeline.

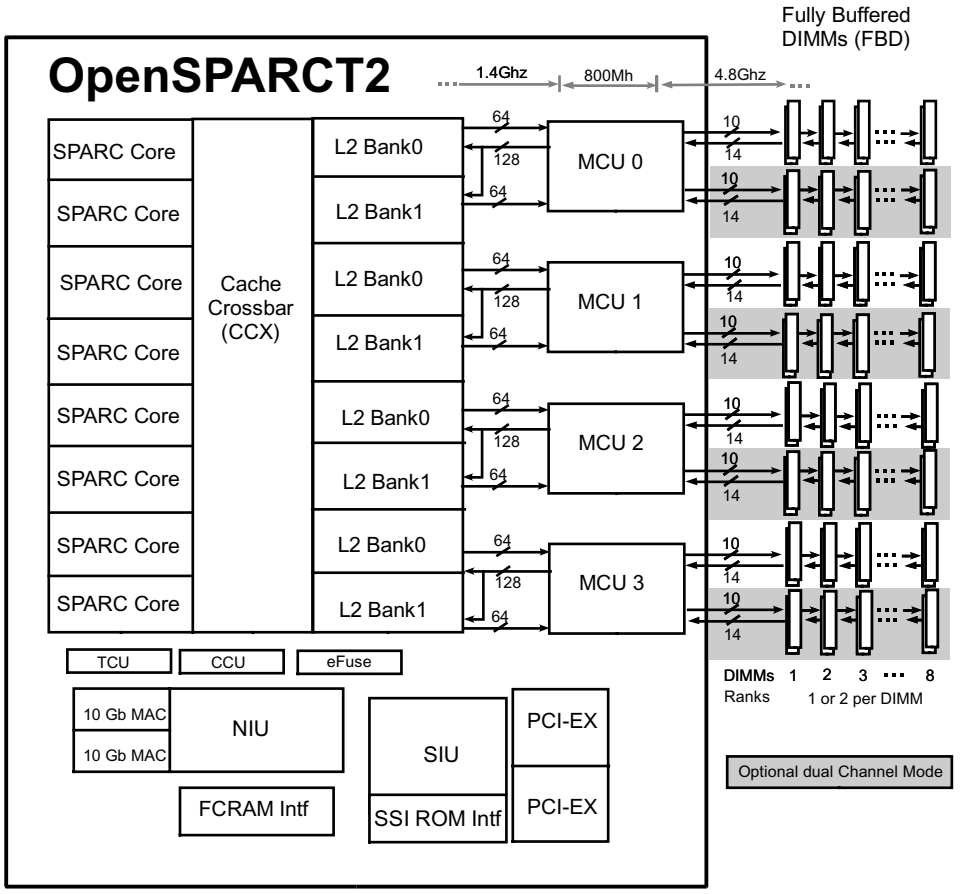
While all eight strands run simultaneously, at any given time at most two strands will be active in the physical core, and those two strands will be issuing either a pair of integer pipeline operations, an integer operation and a floating-point operation, an integer operation and a memory operation, or a floating-point operation and a memory operation. Strands are switched on a cycle-by-cycle basis between the available strands within the hard-partitioned group of four using a least recently issued priority scheme. When a strand encounters a long-latency event, such as a cache miss, it is marked unavailable and instructions will not be issued from that strand until the long-latency event is resolved. Execution of the remaining available strands will continue while the long-latency event of the first strand is resolved.

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

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

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

FIGURE 1-2 OpenSPARC T2 Chip Block Diagram



1.3 OpenSPARC T2 Components

This section describes each component in OpenSPARC T2.

1.3.1 SPARC Physical Core

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

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

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

1.3.2 SPARC System-On Chip (SoC)

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

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

1.3.3 L2 Cache

The L2 cache is banked eight ways. To provide for better partial-die recovery, OpenSPARC T2 can also be configured in 4-bank and 2-bank modes (with 1/2 and 1/4 the total cache size respectively). Bank selection based on physical address bits 8:6 for 8 banks, 7:6 for 4 banks, and 6 for 2 banks. The cache is 4 Mbytes, 16-way set associative with pseudo-LRU replacement (replacement is based on a used bit scheme). The line size is 64 bytes. Unloaded access time is 26 cycles for an L1 data cache miss and 24 cycles for an L1 instruction cache miss.

1.3.4 Memory Control Unit (MCU)

OpenSPARC T2 has four MCUs, one for each memory branch with a pair of L2 banks interacting with exactly one Dynamic Random-Access Memory (DRAM) branch. The branches are interleaved based on physical address bits 7:6, and support

1–16 Double Data Rate (DDR)2 DIMMs. Each memory branch is two Fully Bufered DIMM (FBD) channels wide. A branch may use only one of the FBD channels in a reduced power configuration.

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

1.3.5 Test Control Unit (TCU)

The TCU is the OpenSPARC T2 Test Control Unit and provides access to the chip test logic. It also participates in Reset, EFuse programming, clock stop/start sequencing, and chip debug. The TCU including JTAG is completely stuck-fault testable via ATPG manufacturing scan

1.3.6 Clock Control Unit (CCU)

The Clock Control Unit encompasses the following functions:

- PLL to drive the core and memory clocks
- Interfacing with random number generator
- UCB interface for programming the PLL's/RNG and reading RNG data
- Provide sync pulses for deterministic clock domain crossing
- Clock stretch and other test clocking mechanisms such as SerDes testing (via DTM) for OpenSPARC T2 .

1.3.7 System Interface Unit (SIU)

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

1.3.8 Non-Cacheable Unit (NCU)

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

1.3.9 Data Management Unit (DMU)

The DMU manages Transaction Layer Packet (TLP) to/from the PCI-Express Unit (PEU) and maintains the same ordering as from the PCI-Express Unit (PEU) and then to the SIU. For maintaining ordering between PEU and SIU, the DMU requires the policy that has Programmable Input/Output (PIO) reads pulling Direct Memory Access (DMA) writes to completion. When the PEU issues complete TLP transactions to the DMU, the DMU segments the TLP packet into multiple cacheline-oriented SIU commands and issues them to the SIU. The DMU also queues the response cachelines from SIU, reassembles the multiple cachelines into one TLP packet with maximal payload size. Furthermore, the DMU accepts and queues the PIO transactions requests from NCU, and coordinates with the appropriate destination, to which the address and data will be sent.

The DMU encapsulates the functions necessary to resolve a virtual PCI-Express packet address into a L2 cacheline physical address which can be presented on the SIU interface. The DMU also encapsulates the functions necessary to interpret PCI-Express message signaled interrupts, emulated INTX interrupts and provides the functions to post interrupt events to queues managed by software in main memory and generates the Solaris Interrupt Mondo to notify software. The DMU decodes INTACK and INTNACK from interrupt targets and conveys the information to the interrupt function so that it can move on to service the next interrupt if any (for INTACK) or replay the current interrupt (for INTNACK).

1.3.10 Miscellaneous Input/Output (MIO)

MIO holds majority of non-Serdes I/O's of OpenSPARC T2 . The I/O's in MIO block fall broadly under the functional categories of clock, reset, test (scan and ramtest),ssi interface, process control (PCM) , efuse program enable and debug. Most of the I/O's in MIO are on Boundary Scan chain under control of TCU. All the functional flops in MIO are connected on regular scan chain with scanin,scanout and flush reset capabilities under the control of TCU.

1.3.10.1 Network Interface Unit (NIU)

The NIU connects a pair of on-chip 10 Gb/s Ethernet MACs to the rest of the system. The NIU also contains the registers to control Ethernet traffic.

Note – The NIU microarchitecture description is not included in this document.

1.3.10.2 SSI ROM Interface (SSI)

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

Note – The SSI microarchitecture description is not included in this document.

1.3.11 Debug

This chapter describes OpenSPARC T2 HW features for post silicon debugability which involves debugging any issues that interfere with early bringup as well as debugging the difficult, complex bugs that eluded pre-silicon verification, and are unexpected or unusual corner cases. The overall goal of implementing these features is to make silicon debug more efficient, shortening the time to root cause complex bugs and thereby reducing time to remove and replace.

1.3.12 eFuse

The Efuse (electronic fuse) unit (EFU) contains an Efuse array macro (EFA), TCU interface and an Efuse controller(FCT). In a broad sense, the Efuse array is a non-volatile memory used to store information that needs to be programmed at the factory and used in the field.

The eFuse (Electronic Fuse) unit contains configuration information that is electronically burned in as part of manufacturing, including part Serial Number and Strand_Available information.

1.3.13 Reset

The Reset Unit asserts signals that cause other units to immediately revert to the initial state defined by the Programmer's Reference Manual.

The OpenSPARC T2 team has endeavored to keep OpenSPARC T2 as much the same as OpenSPARC T1 as possible. One major difference is that OpenSPARC T2 conforms to the CMP Programming Model.

CHAPTER 2

Level 2 Cache

This chapter contains the following sections:

2.1 L2-Cache Functional Description

The following sections describe the OpenSPARC T2 processor level 2 cache (L2-cache):

2.1.1 L2-Cache Overview

The OpenSPARC T2 L2 cache is 4 MB in size and is composed of 8 symmetrical banks interleaved on a 64 B boundary. Each bank operates independently of all others. Banks are 16 way set associative and 512KB in size. Block (line) size is 64 B. Each L2 bank has 512 sets.

The L2 cache accepts requests from the SPARC cores on the processor to cache crossbar (PCX) and responds on the cache to processor crossbar (CPX). The L2 is also responsible for maintaining on-chip coherency across all L1 caches on the chip by keeping a copy of all L1 tags in a directory structure. Since OpenSPARC T2 implements system on a chip with single memory interface and no L3 cache, there is

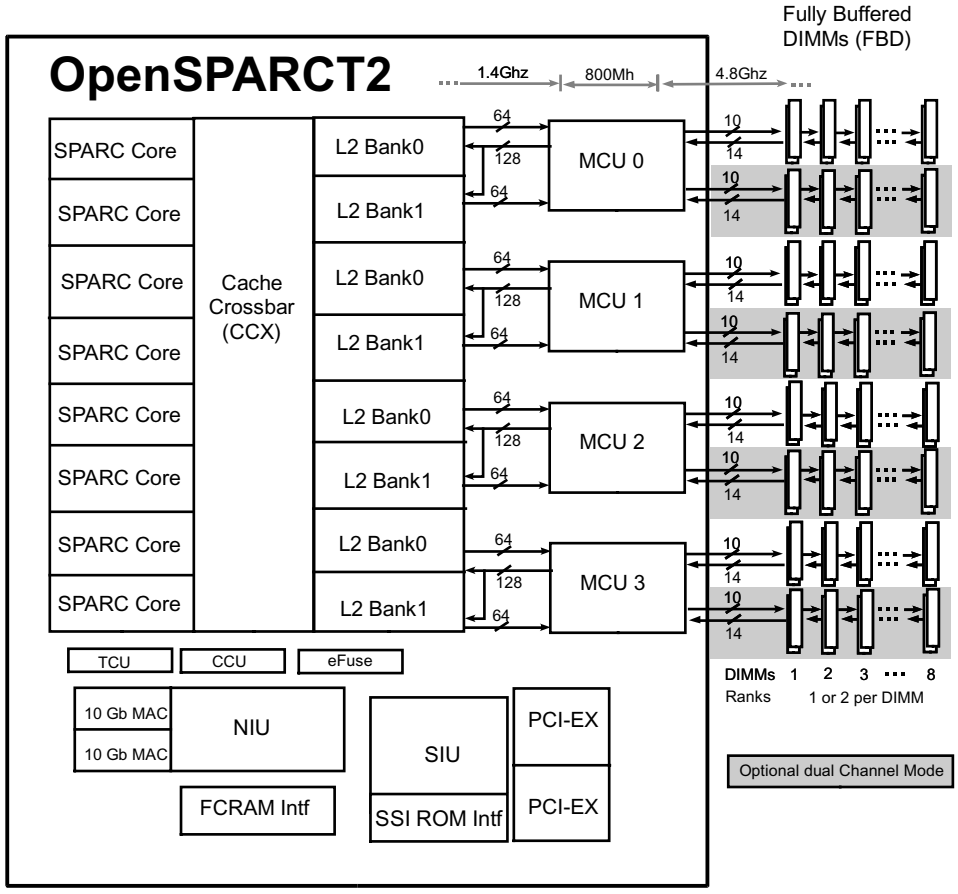
no off-chip coherency requirement for OpenSPARC T2 L2 cache other than being coherent with main memory. The L2 cache is a writeback cache and has lines in one of three states - invalid, clean, or dirty.

Each L2 bank has a 128b Fill interface and a 64b write interface with the dram controller.

Requests arriving on the I/O interface are sent to the L2 from the System Interface Unit.

The L2 cache unit works at the same frequency as the core (1.4 Ghz).

FIGURE 2-1 OpenSPARC T2 Processor Block Diagram



2.1.2 L2-Cache Block Functional Description

The L2 cache is organized into 8 identical banks as shown in the FIGURE 2-1. Each bank has its own interface with SIU,MCU and Crossbar.

Each L2 cache bank interfaces with the 8 cores through a Processor Cache Crossbar. The crossbar routes the L2 request (loads, ifetches, stores, atomics, asi accesses) from all 8 cores to the appropriate L2 bank. The crossbar also accepts read return data,invalidation packets and store ack packets from each L2 bank and forwards them to the appropriate core(s).

Every 2 L2 cache banks interface with one MCU to issue reads and evictions to DRAM on misses in the L2. Writebacks get issued 64bits at a time to MCU. Fills happen 128 bits at a time from MCU to L2.

For 64 byte I/O writes from SIU, L2 does not allocate, but issues the writes to DRAM through a 64 bit interface with MCU. There is a single 64 bit interface with MCU for writebacks and I/O writes, and hence round robin arbitration is used between the Writeback Buffer and the I/O Write Buffer for access to MCU.

Each L2 cache banks also accepts RDD (read to discard), WRI (block write invalidate) and WR8 (partial write with random byte enables) packets from SIU over a 32 bit interface and queues the packet in the SIU Q. RDD and WRI do not allocate in the L2. On a hit, WRI invalidates in the L2 and issues a 64 B block write to DRAM. On a hit, RDD gets back 64 B of data from L2. On a miss, RDD fetches data from DRAM but does not install in L2, while WRI (on a miss) issues a 64 B block write to DRAM. WR8 packets cause partial stores to happen in L2 like regular CPU stores with random byte enables.

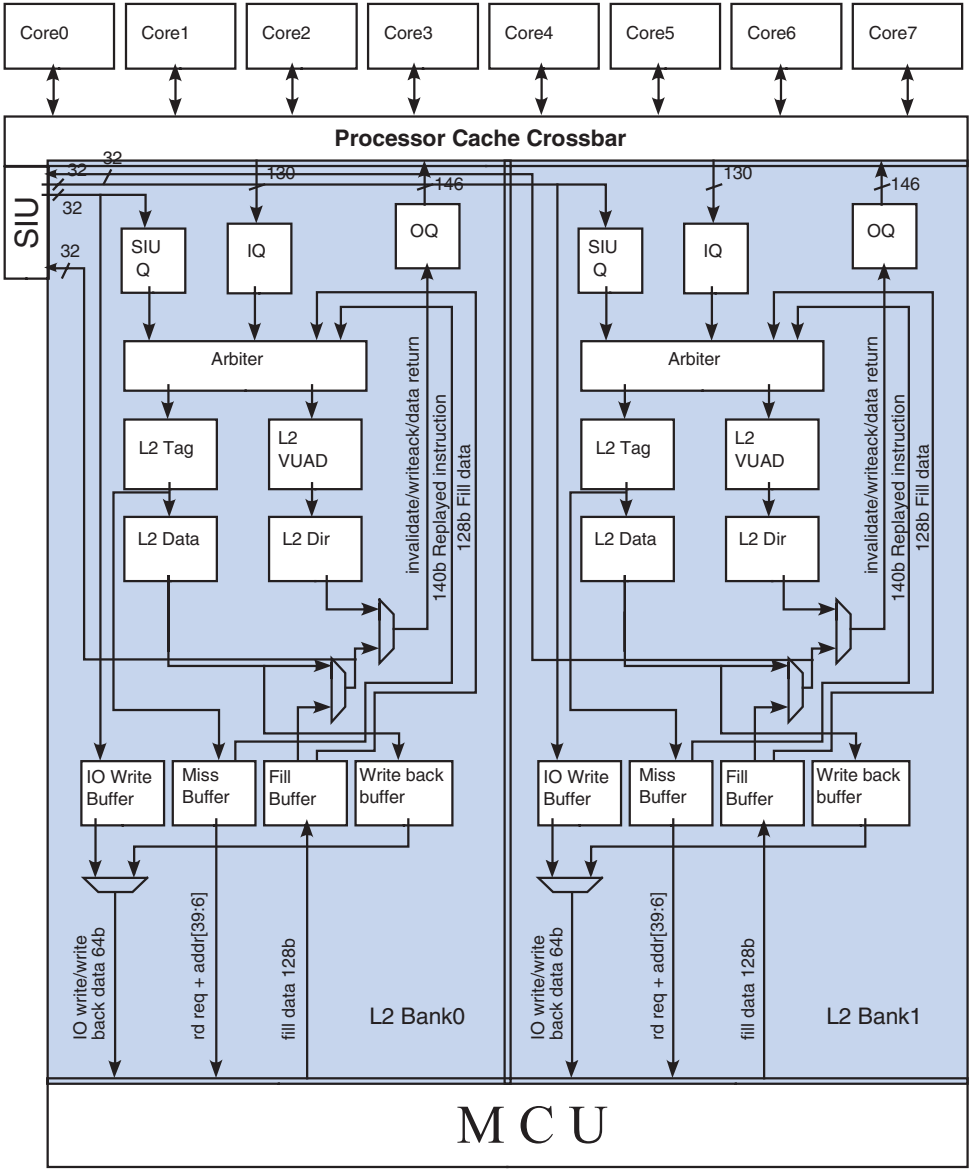
Each L2 cache bank is composed of the following sub-blocks:

- IQ: The input queue is a 16 entry FIFO which queues packets arriving on the PCX when they cannot be immediately accepted into the L2 pipe. Each entry in the IQ is 130 bits wide.
- SIUQ (SIU queue): Accepts RDD,WRI and WR8 packets from the SIU and issues them to the pipe after arbitrating against other requests.
- Arbiter: The arbiter manages access to the L2 pipeline from the various sources which request access. The IQ, MB, SIUQ, FB and stalled instruction in pipe all need access to the L2 pipe.
- L2 Tag: holds the L2 tag array and associated control logic. Tag is protected by SEC ECC
- L2 VUAD: contains the Valid, Dirty, Used and Allocated bits for the tags in L2 organized in an array structure. There is one array for Valid and Dirty bits and a separate array for Used and Allocate bits. Each array is protected by SEC DED ECC.
- L2 Data: Contains 512 KB of L2 Data storage and associated control logic. Data is protected by SEC DED ECC on a 32/7 boundary.
- L2 Directory: The directory maintains a copy of the L1 tags for coherency management and also ensures that the same line is not resident in both the icache and dcache (across all cores). The directory is split into an icache directory (icdir) and a dcache directory (dcdir), which are similar in size and functionality.
- Miss Buffer: The Miss Buffer (MB) has 32 entries and stores instructions which cannot be processed as a simple cache hit. This includes true L2 cache misses (no tag match), instructions that have the same cache line address as a previous miss or an entry in the Writeback Buffer, instructions requiring multiple passes through the L2 pipeline (atomics and partial stores), unallocated L2 misses, and accesses causing tag ECC errors.

- Fill Buffer: The Fill Buffer is an 8 entry buffer used to temporarily store data arriving from DRAM on an L2 miss request. Data arrives from DRAM in four 16B quad-words starting with the critical quad-word.
- Write Back Buffer: The Writeback Buffer is an 8 entry buffer used to store dirty evicted data from the L2 on a miss. Evicted lines are streamed out to DRAM opportunistically.
- I/O Write Buffer: The I/O Write Buffer is a 4 entry buffer which stores incoming data from the PCI-EX interface in the case of a 64 B write operation. Since the PCI-EX interface bus width is only 32 bits wide, the data must be collected over 16 cycles before writing to DRAM

FIGURE 2-2 shows a diagram of the major components of the L2 cache.

FIGURE 2-2 L2-Cache Organization



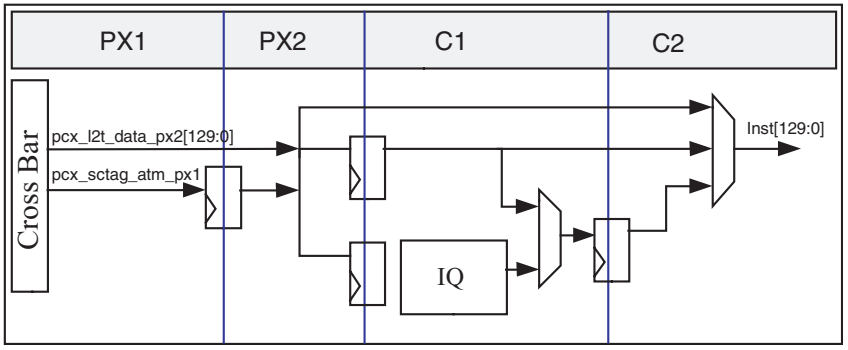
2.1.2.1 L2 Cache Interface Description

L2 cache interfaces with Crossbar, SIU and DRAM.

Crossbar

L2 cache receives requests from the core through the crossbar. These requests are received, decoded and forwarded to the arbiter logic by the Input queue (IQ) depending on the status of the arbiter block. The Input queue pipe line data path diagram is shown in FIGURE 2-3.

FIGURE 2-3 Input Queue Pipeline Data Path Diagram

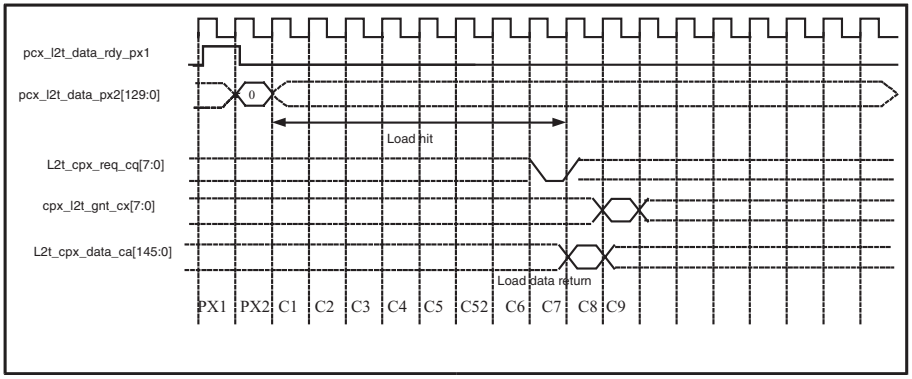


The timing diagram for a single load from PCX is shown in FIGURE 2-4.

The protocol for receiving a request from the crossbar is as follows:

The Input queue receives (pcx_l2t_data_rdy_px1) data valid signal followed by the data (pcx_l2t_data_px2) in the next cycle. Along with the data valid signal, the crossbar also dispatches a signal indicating if the instruction is atomic in nature(pxc_l2t_atm_px1). The request thus received is decoded into address, data and instruction fields in PX2 stage and forwarded to the arbiter logic to request access to L2 cache to process the request. If the arbiter accepts the request, it gets forwarded to L2 in the next clock, at which point the instruction reaches its C1 stage. If the arbiter is busy then it can either be sent after one or two clocks or recorded in the IQ array and dispatched later to the L2 pipe.

FIGURE 2-4 Timing Diagram for a Single Load from PCX

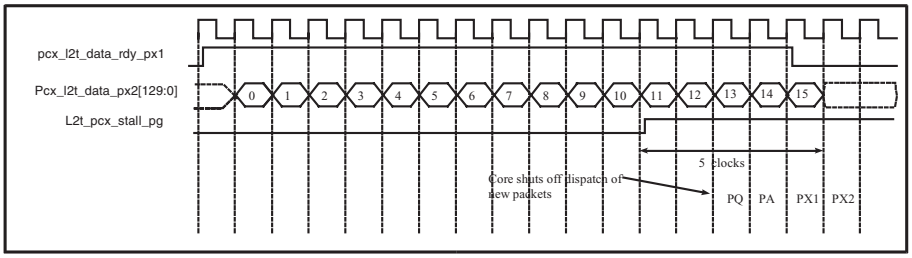


The protocol for L2 cache to send back a packet to the crossbar is as follows:

The L2 cache sends a request (l2t_cpx_req_cq) out in C7 of the pipeline if it has a packet to be dispatched. The packet may be return data for load/ifetch requests, acknowledgments for stores and invalidates for evictions and stores. The packet is dispatched in C8 (through l2t_cpx_data_ca). If the packet is consumed by the crossbar, an (cpx_l2t_gnt_cx) ack is received in C9. If an ack is not received from the crossbar within 1 or 2 cycles from C8, it gets retried from the flops at the input and output of the OQ respectively; if the ack gets received after 2 cycles, it gets retried from the OQ. In case the ack does not come for a long time, the new packets coming from the L2 pipe get accumulated in OQ until OQ fills up at which point the L2 pipe gets stalled.

The Input queue is 16 deep. PCX packets get written to IQ only when the L2 pipeline is stalled or busy and the PX2 arbiter does not accept any new PCX requests. IQ asserts l2t_pcx_stall_pg to crossbar when it is 5 short of being full. This is shown in FIGURE 2-5. These 5 cycles covers the packet shut off latency from core assuming the worst case latency of the core shutting off packet dispatch after dispatching an atomic packet.

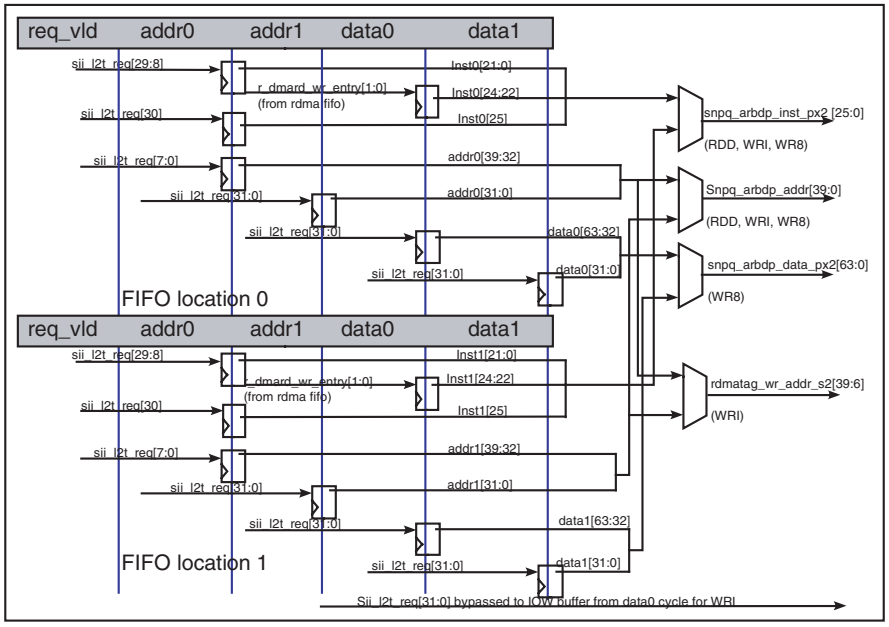
FIGURE 2-5 IQ written from PCX, PCX stall from IQ



SIU Interface

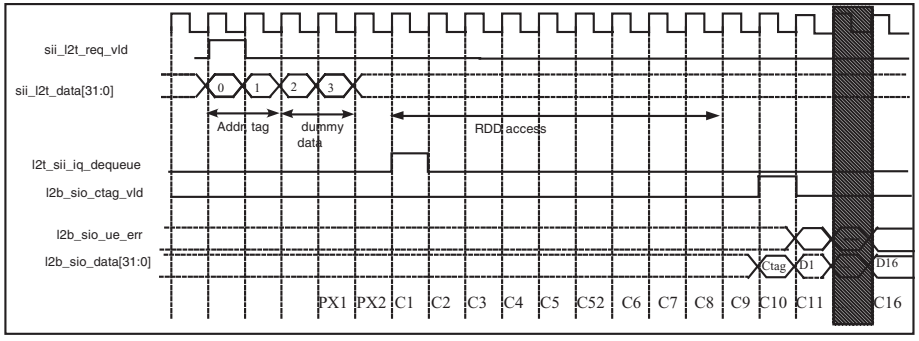
Requests from I/O's are received by L2 cache through SIU Queue block. There are 3 kinds of requests that can be received from the SIU: RDD (Read 64B), WRI (write 64B) and WR8 (write 8Bytes). FIGURE 2-6 shows the pipeline data path diagram for the SIU Queue Block.

FIGURE 2-6 SIU Queue Pipeline Data path Diagram



SIU dispatches requests to L2 cache through an unified address, data and instruction bus called sii_l2t_req. This bus is 32 bits wide.

FIGURE 2-7 Timing Diagram showing RDD request and read data return



The FIGURE 2-7 shows a typical RDD hit in L2.

The protocol to receive a request from SIU is as follows:

A valid signal (sii_l2t_req_vld) is sent along with the request to L2 cache. This signal is used to qualify a valid request transfer from the SIU block. Once the request is received by L2 cache, the instruction is registered and decoded into address, data and instruction fields as shown in FIGURE 2-7.

L2 SIU Queue block can record up to 2 requests in it's 2 deep fifo. Each fifo entry registers the incoming packet from SIU over 4 groups of registers as shown in FIGURE 2-7 for WR8 and RDD transactions and into 2 groups (address and tag) for WRI transaction.For RDD, SIU will issue two dummy (pad) cycles on the sii_l2t_data[31:0] bus, so that the RDD and WR8 pipeline within SIU Queue can stay the same

The requests are received serially. There are 2 counters on the SIU side for flow control. One counter tracks the number of transaction dispatched to L2 cache and the other tracks the number of WRIs issued to L2 cache. The transaction counter is maintained in the SIU side incrementing on a transaction dispatch to L2 cache and decrementing upon receiving l2t_sii_iq_dequeue. WRI counter is incremented on dispatching a WRI transaction to L2 cache and decremented upon receiving l2t_sii_wib_dequeue signal. I/O Write Buffer can hold up to 4 cache lines. The transaction counter would block issue of any more transactions that the 2 deep fifo in the L2 SIU queue block can hold, while the WRI counter will keep a track of overall number of WRI's issued (cannot exceed 4). Thus as long as the WRI's are issued without violating the transaction count specified by the transaction counter, and the WRI count of the WRI counter, there can be 4 WRI' s outstanding to DRAM at any point of time though the SIU queue is 2 deep only.

l2t _sii_iq_dequeue signal is asserted when an instruction is issued down the L2 pipe (WR8,WRI & RDD transactions) in C1 stage. l2t_sii_wib_dequeue is asserted when the contents of an I/O Write Buffer entry are streamed to DRAM (only WRI transaction).

1. RDD: 64 byte read request is received by L2 cache over 5 clocks. During the data cycle, dummy data is driven. The 64byte data from L2 is returned to SIU over 16 cycles with ctag_vld information.

2. WR8: 8 Byte writes are received by L2 cache over 5 clocks. The L2 treats this instruction in exactly the same way as a store. When the write data gets written into the L2 Data Array, an encoded 32 bit ack is sent out to SIU by asserting ctag_vld in the same clock.

3. WRI: 64 byte write is received by L2 cache over 19 clocks. The line being written is not allocated in the L2 cache. However if the write hits in the L2 cache, it invalidates the L2 cache entry and also copies of the line in L1 caches. The WRI packet gets written into the I/O Write Buffer from where the data gets written to DRAM opportunistically. l2t_sii_iq_dequeue signal is asserted when the write instruction is issued down the L2 pipe, and l2t_sii_wib_dequeue is asserted when

the contents of an I/O Write Buffer entry are streamed to DRAM. Also an encoded write ack is sent out to SIU on l2b_sio_data[31:0] by asserting ctag_vld indicating completion of the WRI.

Ordering of SIU Transactions in L2 (Data Returns and Write Acks from L2 to SIU):

1. For same address to the same L2 bank, read returns and write acks will be always in transaction order from SIU
2. For different addresses to the same L2 bank, depending on hit or miss, RDD's can send back data out of order.
3. For different addresses to the same L2 bank, WR8's (partial writes with byte masks) can send back acks out of order, depending on hit or miss. (WR8's do read modify writes, and WR8 ack gets sent only in the store update phase of the WR8). So if there are two back to back WR8's, and the first one misses, the second one hits: ack will get sent for the second one before the ack for the first one, while the first one is still waiting to fetch the data from memory.
4. For different addresses to the same L2 bank, WRI's will send back acks in transaction order from SIU as WRI's go straight to memory and do not update L2.

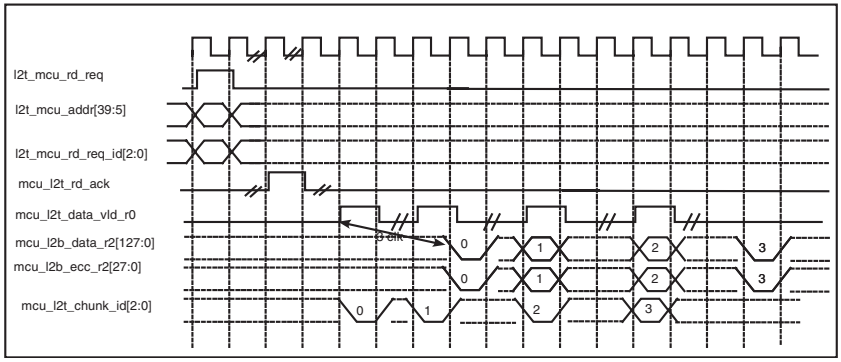
2.1.2.2 MCU Interface:

L2 cache issues read and write requests to MCU. All instructions which do not hit in L2 cache are recorded in the Miss Buffer (MB). Miss Buffer evaluates and sets a (dram_pick) bit if it needs to be issued to MCU.

Reads which need to be dispatched to MCU should satisfy the following criteria:

- Win arbitration among all pending reads (with the dram_pick bit set for reads).
- Should have no pending (read or write) transactions to MCU waiting for an ack.
- Should have enough place in the Fill Buffer for the read data to return.

FIGURE 2-8 Read Request from L2 Cache to MCU and Read Data Return



The protocol for sending out a read request is as follows:

A read request is dispatched to MCU by asserting a a read request (l2t_mcu_rd_req) signal. Along with the read request the address and a read request ID (l2t_mcu_rd_req_id) is dispatched in the same cycle. The read address is also recorded in the Fill Buffer. MCU records and processes the read request. A read ack (mcu_l2t_rd_ack) is sent back indicating that the read request was recorded.

When the data is ready, MCU returns the data to the L2 cache. The data is returned with the data_valid (mcu_l2t_data_vld_r0) being sent first. Data (128 bits wide mcu_l2t_data_r2), the read request id (mcu_l2t_qword_id[1:0]) and ECC information (mcu_l2t_ecc_r2 [27:0]) related to the data is sent after 3 clocks. The read data returned by the MCU gets recorded by the Fill Buffer. Upon receiving the data, the missed load/ifetch from the Miss Buffer gets replayed through the L2 pipe and reads the data from the Fill Buffer itself (critical 16B or 32B first) and sends data to requesting core. After this, the Fill Buffer requests arbiter to complete the fill. The qword data arrives from MCU in 4 packets. There is no relationship between the dispatch of packets.

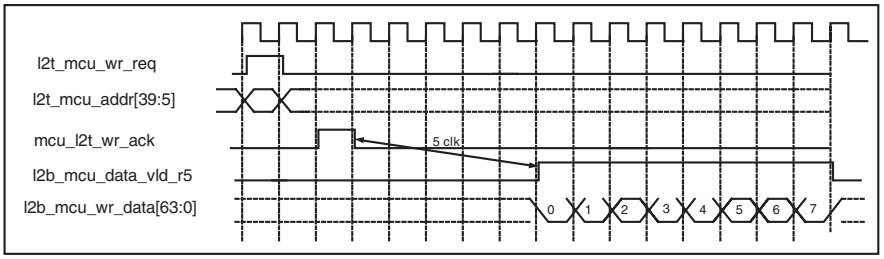
Also in case of a miss in L2 for a Block Init Store with PA[5:0] = 0, L2 will issue a dummy read request to DRAM (l2t_mcu_rd_dummy_req), but MCU will send back all 0's in 4 packets. L2 will install the line with all 0's in the data.

Writes to the MCU get issued when a request is recorded in the I/O Write Buffer (IOWB) or Write back Buffer (WBB).

The following condition needs to be satisfied for a write to be dispatched to MCU:

- Win arbitration among all pending writes. Writes can be in IOWB (WRI from SIU) or WBB (eviction).
- There should be no pending (read or write) transactions to MCU waiting for an ack.

FIGURE 2-9 MCU Write Transaction



The protocol is similar for writes:

A write request (l2t_mcu_wr_req) is sent to MCU along with the address for the write data. Upon receiving a write request, DRAM sends a acknowledge back indicating it is ready for receiving the write data. L2 cache takes 5 clocks upon receiving the ack to the time it starts to send data to DRAM in sizes of 8Byte.

2.1.3 L2 Pipeline

The L2 pipeline has 9 stages, the details of which are described below.

- Arbitration (PX2):
  - Mux between PCX, IQ, I/O, MBF and FBF and C1 (stalled) instructions
- Tag Access (C1):
  - Tag access, VUAD Read and Bypass
  - Tag Compare
  - Miss Buffer CAM Operation in Phase1.
  - Miss Buffer Hit logic
  - Generation of ECC for store data
  - Generation & check of ECC for the access address
  - WBB and Fb CAM in Phase 2.
- Way Sel Generation (C2):
  - Tag Hit logic
  - Replacement way logic (pseudo LRU)
  - Miss Buffer Hit generation and 2 cyc bypass.
  - Way select logic
  - Set, index, col, way sel, rd/wr, word enables xmit to the data array

    - Stall for mutlicycle operations (For e.g. Eviction, Fill etc.) or column offset collision.
- Way Sel Xmit (C3):
    - Set, index, col, way sel, rd/wr, word enables xmit in the data array
    - Evict way sel generation and eviction logic
    - MB tag write, MB Valid bit setting
    - Fb hit entry xmit to FB Data array (in l2b)
- Data Access cyc1 (C4):
    - Data array read/write cycle 1 (for load/store hit)
    - Fb data buffer read.
    - Way sel transmit to data array for Fill only
    - Setup directory inputs for CAM/write operations.
- Data Access cyc2 (C5):
    - Data array read/write cycle 2 (for load/store hit)
    - Data array read cyc 1 for eviction
    - WB tag write in the case of a dirty eviction
    - VUAD array Write
    - Way sel transmit within data array for Fill only
    - Stage Fb data
    - Write/CAM directory
- Data Access cyc3 (C52):
    - Data array read/write cycle 3 (for load/store hit), 4:1 mux for L2 data
    - FB and L2 data mux
    - Data array read cyc 2 for eviction
    - Data array write cyc 1 for Fill
- Data Return xmit (C6):
    - 16B data xmit to tag block
    - Data array read cyc 3 for eviction
    - Invalidation vector processing.
    - Request vector generation logic
    - Data array write cyc 2 for Fill
- Error Correction (C7):
    - Error Correction/Detection
    - Request vector to OQ/CPX
    - Data array write cyc 3 for Fill

- Data Response (C8):
  - L2 data and Invalidation data MUX
  - Data xmit to OQ/CPX
  - Write WBB data
  - 64b data merge for PSTs and 64b compare for CASX

2.1.4 L2 Interactions with Core

OpenSPARC T2 cores will use 8 bit Byte Mask fields for stores instead of 2 bit size field that OpenSPARC T1 cores use. The main reason for this is to support VIS partial stores with random byte enables.

This section describes the pipeline flow for a few representative L2 operations.

2.1.4.1 Load Hit

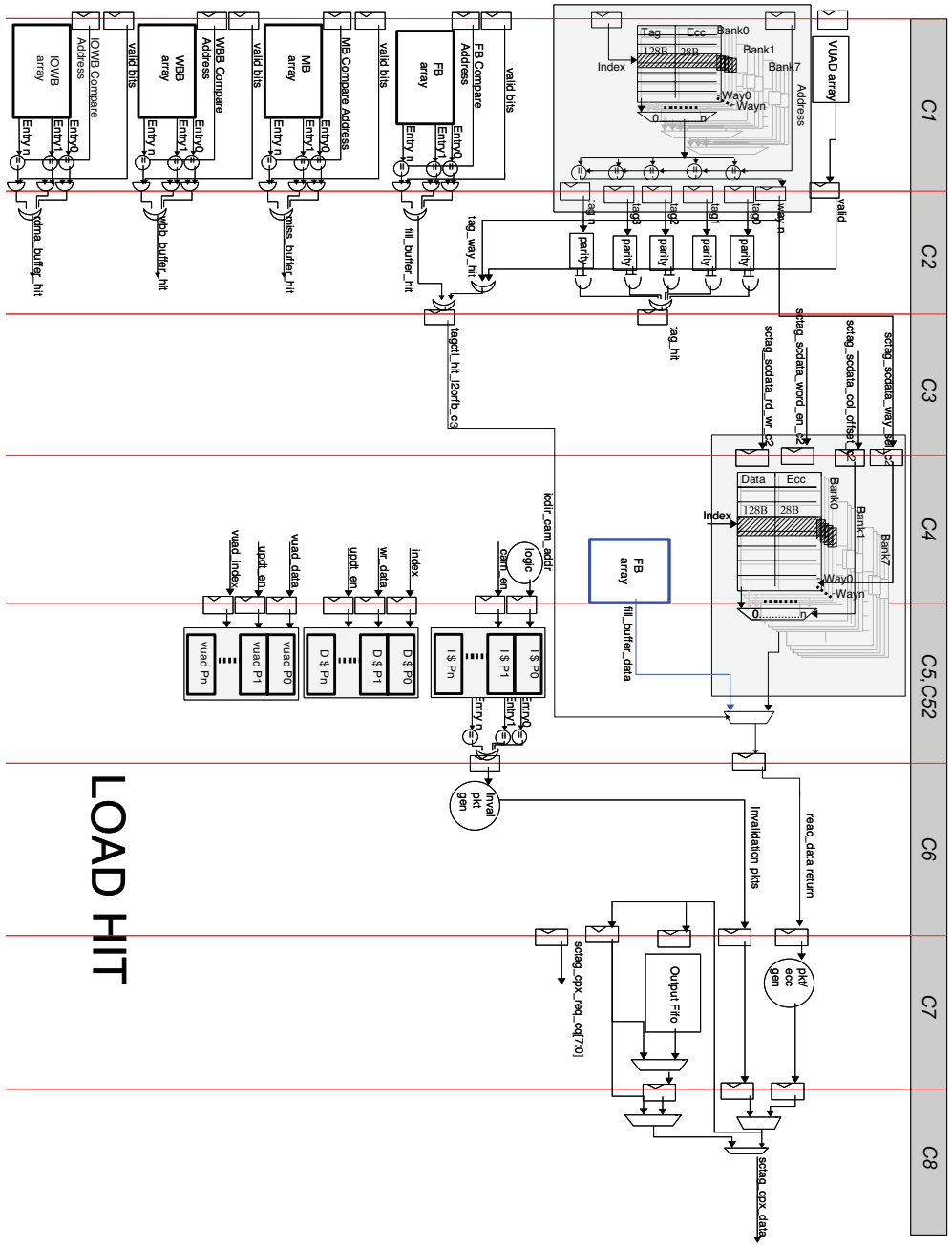
TABLE 2-1 Pipeline Diagram: Load Hit

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag, VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! Gen,xmit VUAD ECC check	way sel xmit in l2d	data array read cyc1 FB data read cycle Xmit inputs to directory	data array read cyc2 stage FB data D$ directory write I$ directory CAM VUAD write	data array read cyc3 4:1 mux mux with FB data	data xmit cycle gen inval. vector	request to the dest cpx queue check ECC on data	Mux Data/Inval. Vector data return to dest. cpx

Loads always return 16B of data, and lower address bits are ignored, i.e. if a load request to address 0x13 is presented to the L2, the 16 bytes at 0x10 are returned. The 8 bit byte mask field is ignored for loads. The different instruction types that fall in

the category of loads are: *load, prefetch, stream load, mmu load*. Out of these, *prefetch, stream load and mmu load* are non-cacheable (will have NC bit set in the PCX packet). These loads do not cam the I$ directory and do not update the D$ directory.

FIGURE 2-10 Load Hit



2.1.4.2 Store Hit

Eight bytes of store data is always sent to the L2. The LSU will ensure that the data is properly aligned to the 8B boundary. The 8 bit byte mask indicates which bytes are to be stored. Again, the lower address bits are ignored.

(This is different than OpenSPARC T1. OpenSPARC T1 L2 had to use the lower address bits along with the size to determine what to store.)

FIGURE 2-11 shows the timing diagram for a 8/4 byte store with the following combination of Byte Enables:

1111 1111

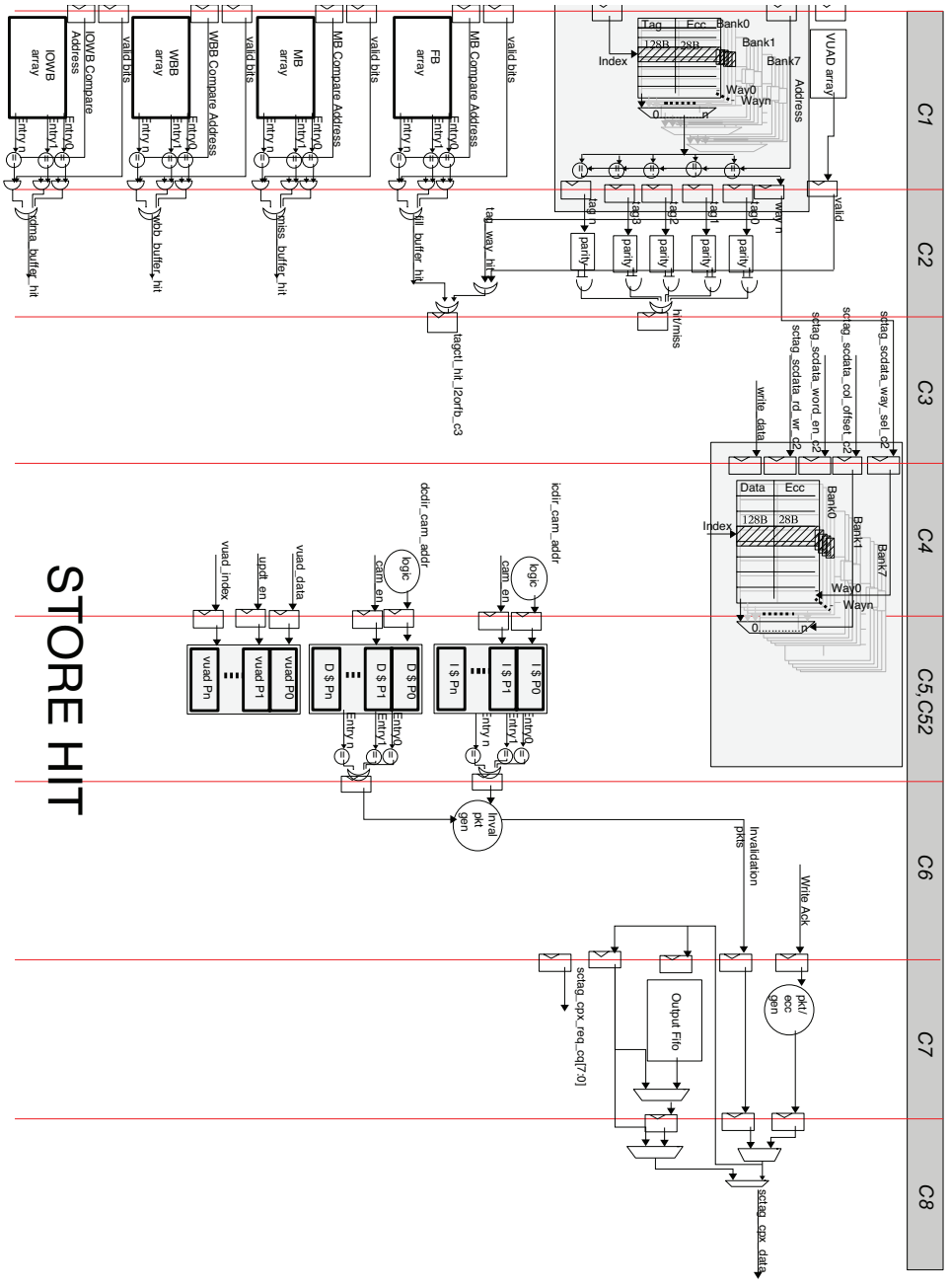
1111 0000

0000 1111

TABLE 2-2 Pipeline Diagram: Store Hit

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag,VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic perform store data ECC FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! gen,xmit VUAD ECC check	way sel xmit in l2d	ddata array write cyc1 Xmit inputs to directory	data array write cyc2 I$ and D$ directory CAM VUAD write	data array wr cyc3	gen inval.vector	request to the dest cpx queue (ack for write) check ECC on data	Mux Data/Inval. Vector

FIGURE 2-11 Store Hit



To improve the performance of stores from L1, L2 cache in OpenSPARC T2 would

send back acks to core in case stores from L1 hit to outstanding store miss to the same line in Miss Buffer. This would involve adding a control flop in Miss Buffer control logic to associate a load miss or store miss with the MB entry. However the ack will get sent back if it hits only in store misses. If any one of the addresses it hits is a load miss, the ack will not be generated.

Note – In OpenSPARC T1, stores are ack'ed when they make their first pass through the L2 pipe - hit or miss. The exception to this is when the store hits an entry in the Miss Buffer. The reason for not issuing the ack in this case is that if the entry in the MB were a load, the ack would cause the L1 to update before the load returned data, causing WAR hazard. However, if the entry in the MB was a store, no such hazard exists, and the ack can be issued.

Main reason for wanting to add this earlier ack capability in OpenSPARC T2 is to complement the addition of store pipelining in the L1 for stores going to the same L2 cache line. In this scheme, a store that follows another store to the same L2 line can be issued without waiting for the first store to be ack'ed, however in the absence of acks for these stores, the Store Buffer entries cannot be dequeued. This would stall dequeue of Store Buffer entries due to stores to different lines and different banks also that are behind the stores to the same line. In this particular case (stores to the L2 line are all waiting in the Miss Buffer for the data return from DRAM), this can amount to stalling the drain of the Store Buffer in L1 for ~160 cpu cycles, causing it to be filled up and thread(s) to stall.

The Store Buffer is only 8 entries per thread, and once it fills up, the thread stalls, so we want to minimize this. STB stalls cause a noticeable degradation in performance. The decrease in stalls from adding pipelining gained somewhere around 15% on SpecINT, so it's a worthwhile change.

2.1.4.3 Partial Store

Partial stores are stores which have any combination of byte masks other than 00001111,11110000 and 11111111.

Even for partial stores, eight bytes of store data is always sent to the L2. The LSU will ensure that the data is properly aligned to the 8B boundary. The 8 bit byte mask would indicate which bytes are to be stored. Again, the lower address bits (0,1,2) are ignored.

Partial stores are handled as a read-modify-write operation in two passes through the pipe. The first pass is shown in Timing Diagram . The second pass is identical to a store, except that the ack does not get sent again.

TABLE 2-3 Timing Diagram: Partial Store

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag,VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! Gen,xmit VUAD ECC check	way sel xmit in l2d	data array read cyc1 FB data read cycle Xmit inputs to directory	data array read cyc2 stage FB data I$ and D$ directory CAM VUAD write	C52 data array read cyc3 4:1 mux mux with FB data	data xmit cycle gen inval. vector	request to the dest cpx queue (ack for write) check ECC on data	Mux Data/inval vector Merge data

The merged data is written into the Miss Buffer and is readied for reissue in C9.

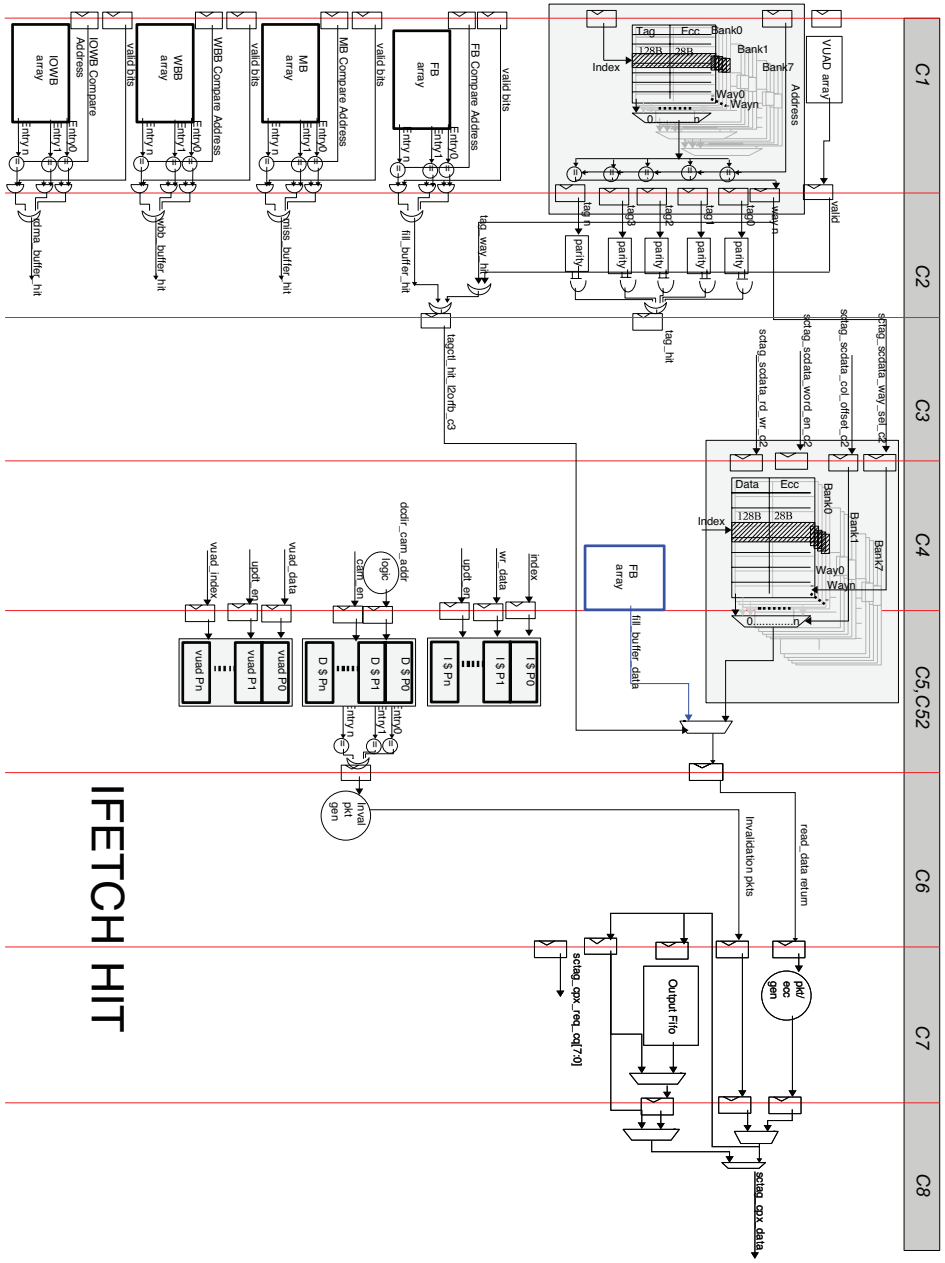
2.1.4.4 Ifetch Hit

ICache line is 32 B, so two data reads are required for an Instruction Fill request.

TABLE 2-4 Timing Diagram: Ifetch Hit

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag,VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! Gen,xmit VUAD ECC check stall next instru-ction	way sel xmit in l2d	data array read cyc1 FB data read cycle Xmit inputs to directory	data array read cyc2 stage FB data I$ directory write D$ directory CAM VUAD write	data array read cyc3 4:1 mux mux with FB data	data xmit cycle gen inval. vector	request to the dest cpx queue check ECC on data	Mux Data/Inval. Vector data return to dest. cpx

FIGURE 2-12 Ifetch Hit



The following 16B data block is transmitted in C9. Note that Ifetch misses are 32B

aligned, and 32 B get returned to the core over two consecutive cycles. For a 32 B ifetch with lower address bits non-zero (unaligned 32B read), the two 16B lines are returned in address order, not critical line first. The 8 bit byte mask field is ignored for Ifetch.

Note that if the NC bit is a 1 for an Ifetch request (L1 I$ is disabled), it will still cam the D$ directory and send an invalidation vector if there is a hit in the D$ directory.

2.1.4.5 Miss

An instruction that does not hit the L2 cache, Fill Buffer or the Writeback Buffer is queued in the Miss Buffer as a "true miss". Eviction is performed during the second pass of the miss operation. This is done to remove the hit/miss determination from the critical C1 stall signal. To improve the performance of stores from L1, L2 cache in OpenSPARC T2 would send back acks to core in case stores from L1 hit to outstanding store miss to the same line in Miss Buffer. This would involve adding a control flop in Miss Buffer control logic to associate a load miss or store miss with the MB entry. However the ack will get sent back if it hits only in store misses. If any one of the addresses it hits is a load miss, the ack will not be generated.

TABLE 2-5 Timing Diagram: Miss

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag,VUAD read tag compare MB CAM perform store data ECC (for store) VUAD bypass FB CAM WBB CAM	way sel logic, check VUAD ECC	write MB tag set MB valid bit					request to the dest cpx queue (ack for write in case it is a store miss but hits in one or more MB store miss entries)	

In C9, EVICT_READY bit gets set in MB. The instruction (along with data for a store) gets written to MB also in C9.

2.1.4.6 Eviction (Clean or Dirty)

The entry in the Miss Buffer is selected for issue in case the EVICT_READY bit gets set indicating it is ready for eviction. An evict instruction gets issued from the Miss Buffer which causes eviction to happen as it makes a pass down the pipe. This also clears the EVICT_READY bit provided evict instruction pass does not encounter a TECC error. Invalidation of L1 cache lines happen for eviction of clean or dirty lines. However only dirty lines are sent to DRAM, clean lines just get overwritten.

TABLE 2-6 Timing Diagram: Eviction

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag,VUAD read tag compare VUAD byp	perform pseudo LRU stall two cycles to avoid collision with instruction after evict instruction for data array access	way sel logic mux out evicted tag way sel Xmit to l2d	way sel xmit in l2d	data array read cyc1 I$ and D$ directory CAM write evicted tag in WBB VUAD Write	data array read cyc2	data array read cyc3 gen inval. vector	xmit inv response to dest cpx queues	inv packet to dest cpx data queues write WBB data	ready for request to DRAM

2.1.4.7 Fill

TABLE 2-7 Timing Diagram: Fill

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag write	stall three cycles to avoid collision with instruction after fill for data array access	Xmit FB entry to l2b	way sel xmit to l2d read FB	VUAD write Xmit way sel inside l2d mux fbdata with l2d	data array wr cyc 1	data array wr cyc2	data array wr cyc3		

FIGURE 2-13 Read Miss and Read Data Fill from DRAM

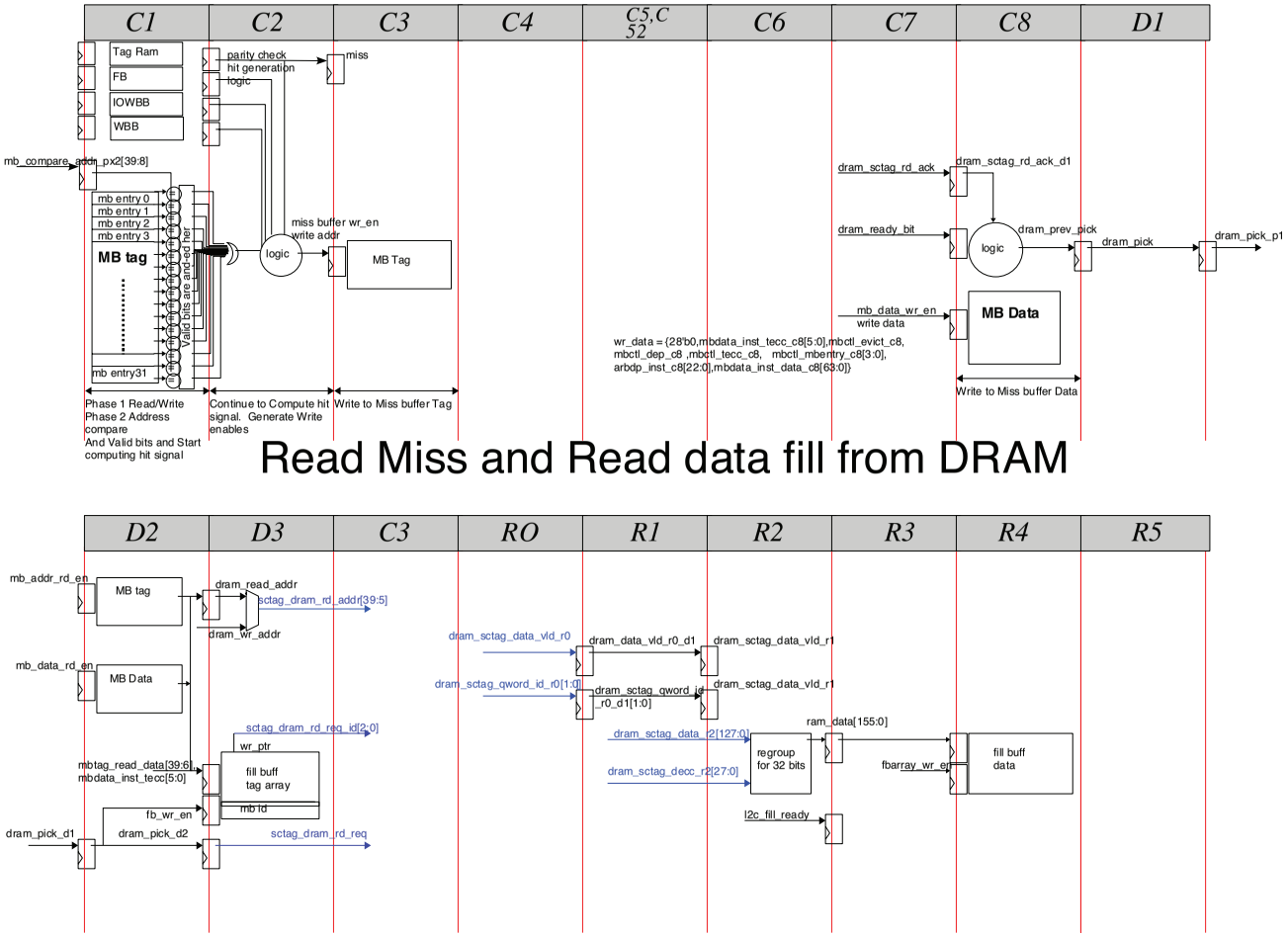
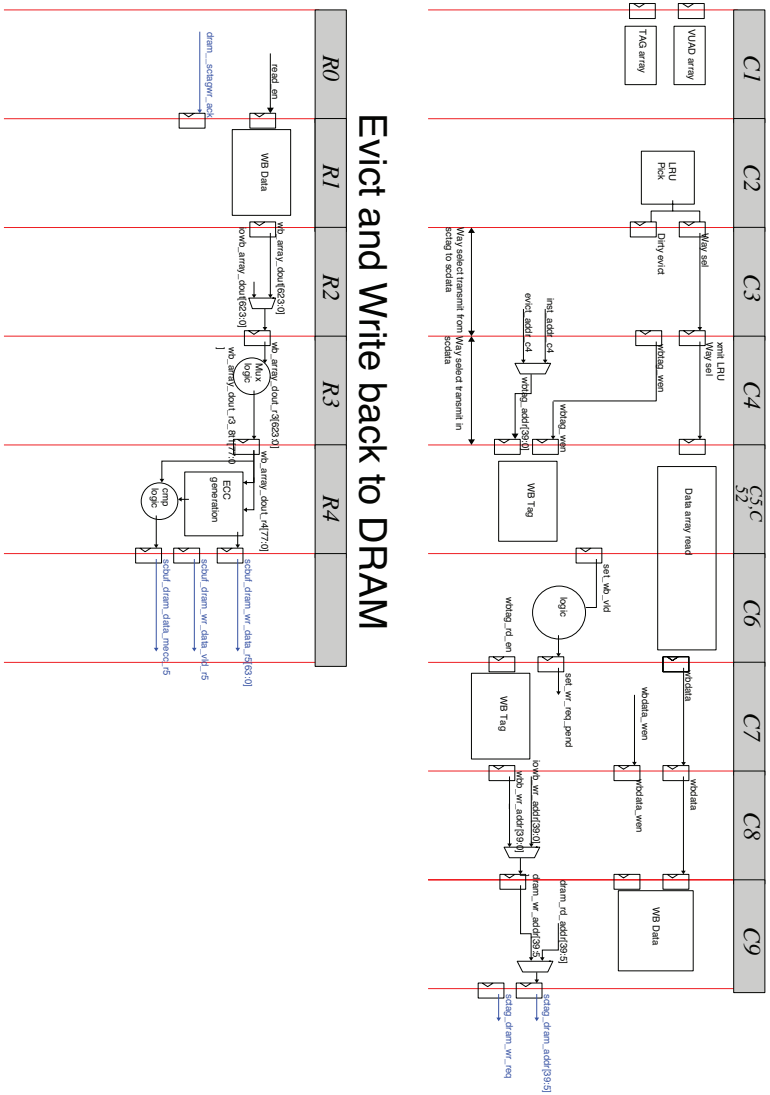

C1
C2
C3
C4
C5,C52
C6
C7
C8
D1
Tag Ram
FB
IOWBB
WBB
parity check
hit generation
logic
miss
mb_compare_addr_px2[39:8]
mb entry 0
mb entry 1
mb entry 2
mb entry 3
MB tag
mb entry31
Valid bits are and-ed her
miss buffer wr_en
write addr
logic
MB Tag
Phase 1 Read/Write
Phase 2 Address compare
And Valid bits and Start computing hit signal
Continue to Compute hit signal. Generate Write enables
Write to Miss buffer Tag
dram_sctag_rd_ack
dram_sctag_rd_ack_d1
dram_ready_bit
logic
dram_prev_pick
dram_pick
dram_pick_p1
mb_data_wr_en
write data
MB Data
wr_data = {28'b0,mbdata_inst_tecc_c8[5:0],mbctl_evict_c8,
mbctl_dep_c8 ,mbctl_tecc_c8, mbctl_mbentry_c8[3:0],
arbdp_inst_c8[22:0],mbdata_inst_data_c8[63:0]}
Write to Miss buffer Data
Read Miss and Read data fill from DRAM
D2
D3
C3
RO
R1
R2
R3
R4
R5
mb_addr_rd_en
MB tag
dram_read_addr
sctag_dram_rd_addr[39:5]
dram_wr_addr
mb_data_rd_en
MB Data
sctag_dram_rd_req_id[2:0]
wr_ptr
mbtag_read_data[39:6],
mbdata_inst_tecc[5:0]
fill buff
tag array
mb id
fb_wr_en
dram_pick_d1
dram_pick_d2
sctag_dram_rd_req
dram_sctag_data_vld_r0
dram_data_vld_r0_d1
dram_sctag_data_vld_r1
dram_sctag_qword_id_r0[1:0]
dram_sctag_qword_id_r0_d1[1:0]
dram_sctag_data_vld_r1
dram_sctag_data_r2[127:0]
dram_sctag_decc_r2[27:0]
regroup
for 32 bits
ram_data[155:0]
fbarray_wr_e
fill buff
data
l2c_fill_ready

FIGURE 2-14 Evict and Write back to DRAM



2.1.4.8 Atomics LDSTUB/SWAP 1st Pass

Same as a load with a merge in C8.

TABLE 2-8 Timing Diagram: Atomics LDSTUB/SWAP 1st Pass:

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag,VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! Gen, xmit VUAD ECC check	way sel xmit in l2d	data array read cyc1 FB data read cycle Xmit inputs to directory	data array read cyc2 stage FB data D$ directory write I$ directory CAM VUAD write	data array read cyc 3 4:1 mux mux with FB data	data xmit cycle gen inval. vector	request to the dest cpx queue check ECC on data	Mux Data/Inval. Vector data return to dest. Cpx Merge data as in Partial Store

The first pass through the L2 pipe reads 16B of data at the address requested (ignoring the lower bits), returns it to the requesting processor, and merges the swap/UB data. The merged data is written into the Miss Buffer and is readied for reissue in C9. The instruction then goes through a second pass upon which the new data is stored and an acknowledgment is sent to the requesting processor. The second pass of a ldstub/swap is same as that for a store hit.

For SWAP and LDSTUB, the bytes to write in L2 will be picked up from the Byte Mask itself. SWAP is always 32b aligned on 4 byte boundary and LDSTUB is always 8b.

2.1.4.9 Atomics CAS

CAS{X} instructions are handled as two packets. The first packet (CAS(1)) reads the data from memory, sends the data back to the requesting processor, and performs the comparison in C8. The second packet (CAS(2)) is inserted into the MB as a store. If the comparison result is true, the second packet proceeds like a normal store. If the result was false, the second pass proceeds to only generate the store acknowledgment. The data arrays are not written.

CASA/CASXA are similar, but with one difference. CASA is 32b, aligned on 4 byte boundary and CASXA is 64b. The compare and conditional store are assumed to be on an 8B boundary (except the load return which is always 16B). The 8 bit Byte Mask will indicate which bytes to compare and conditionally store.

2.1.4.10 Prefetch Invalidate Cache Entry (ICE)

L2 supports Prefetch ICE instruction which gets used by SW to flush lines in L2 based on an index and a way specified as part of the Physical Address in the instruction itself. Bits [39:37] of the PA has to be driven as 3'b011 by SW and the way, index, bank information would be on PA[21:18], PA[17:9] and PA[8:6] respectively. LSU issues a prefetch instruction over the crossbar to L2 with bit 116 (inv bit) of the cpx packet being 1'b1. On seeing this packet, L2 flips bit 39 to 1'b1 before storing it in IQ array or feeding to the pipe. Thus with PA[39:37] = 3'b111, it is guaranteed that the instruction always misses in L2 irrespective of 8/4/2 bank mode of operation.

The Prefetch ICE gets executed in L2 in two passes. First pass is like a regular prefetch which misses in the L2 tags.On the miss, the instruction gets written to the Miss Buffer and also the Evict bit gets set. However the DRAM read gets suppressed as this is a flush instruction only and no data needs to be read from memory.

In the second pass, an evict instruction gets issued from the Miss Buffer for the Prefetch ICE and it will use the way specified in PA[21:18] of the Prefetch ICE packet itself to pick the Eviction way and L2 Directory Lookup way, instead of the way picked by the LRU logic. Then the eviction proceeds like normal: an eviction invalidation packet gets generated and sent to the crossbar to invalidate all L1 ways for all cores that are included in that line. In case the line is dirty, a writeback happens to DRAM. In the eviction pass of the instruction, it gets deleted from the Miss Buffer. No response packet gets sent to the cores for the instruction itself.

Note that in case the Prefetch ICE instruction encounters a Tag Parity Error or VUAD CE in either the first pass, the error is ignored (not logged and reported) and the Prefetch ICE goes on as normal. However in its second pass if the Prefetch ICE detects a tag parity error, it will be re-inserted into the Miss Buffer, the eviction pass will not happen and a scrub will be issued from the Miss Buffer. After the scrub is complete, the eviction pass of the Prefetch ICE will occur and if this time there are no more tag parity error detected, the eviction pass will complete. This is because the tag parity error could have corrupted any bit of the address, so that unless corrected, the eviction of a dirty line would cause data corruption in memory. However if the eviction pass of the Prefetch ICE encounters a VUAD CE, the error would be ignored and the eviction pass would go through since we know the way already that has to be evicted.The VUAD data would get silently corrected before it gets written to the VUAD array in C5.

Note that since any error in the Dirty bit would have been silently cleaned in the first pass of Prefetch ICE itself, in the second pass of the Prefetch ICE, the way to flush would be identified correctly as clean or dirty and the dirty line would get

evicted to DRAM properly. Also if the Dirty bit error gets detected in C2 stage of the second pass of Prefetch ICE (the corruption in the array happened in between the first and second passes), the way to flush would still get correctly identified as clean or dirty in the second pass, as the data would get silently corrected in c2 stage of the second pass itself.

Prefetch ICE First Pass (Miss in L2)

TABLE 2-9 Timing Diagram: Prefetch ICE First Pass (Miss in L2):

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag, VUAD read tag compare MB CAM VUAD bypass FB CAM WBB CAM	miss detected Tag parity error NOT checked VUAD ECC checked, data corrected if SE detected (CE not logged)	write MB tag set MB valid bit		VUAD write					set evict_rdy in MB Prefetch ICE written in MB dram_rdy not set

In C9, EVICT_READY bit gets set in MB. The instruction gets written to MB also in C9. However dram_ready bit does not get set in C9, thereby stopping issue of a DRAM read request.

TABLE 2-10 Timing Diagram:2nd Pass of Prefetch ICE (Eviction plus Delete from Miss Buffer)

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag, VUAD read tag compare VUAD byp	pick replacement way from PA[21:18], Tag parity error NOT checked VUAD ECC checked, data corrected if SE detected. (CE not logged) stall two cycles to avoid collision with instruction after evict instruction for data array access	way sel logic mux out evicted tag way sel Xmit to l2d delete entry from MB	way sel xmit in l2d	data array read cyc1 I$ and D$ directory CAM write evicted tag in WBB write VUAD array	data array read cyc2	data array read cyc3 gen inval. vector	xmit inv response to dest cpx queues	inv packet to dest cpx data queues write WBB data (if dirty)	ready for WBK request to DRAM (if dirty)

Note that to ensure ordering, after the Prefetch ICE is inserted into the Miss Buffer, requests from crossbar and SIU are blocked from entering into the L2 pipeline, and the second pass of the Prefetch ICE (eviction pass) is not issued from the Miss Buffer until the Miss Buffer count becomes 1 (the Prefetch ICE is the only instruction in the Miss Buffer). After the second pass of Prefetch ICE completes, the stall of the crossbar and SIU requests are removed, and instructions that accumulated in IQ Array and Snoop queue can go through.

Diagnostic Read of Data Array:

TABLE 2-11 Timing Diagram: Diagnostic Read of Data Array

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag, VUAD read	disable way sel gen stall c1 instruc.	Gen way sel from decoded address Xmit way sel to l2d	Xmit way sel inside l2d	read data array cyc 1	read data array cyc 2	read data array cyc 3	Xmit 156 bits of data to l2t	Mux out 39 bits. Xmit req on CPX	Xmit 39 bits of data with rest of CPX packet

Diagnostic read access to the L2 data array is done through 64 bit read that access a 32 bit data subblock along with the corresponding 7 bit ECC. The instruction that gets used is ***Diagnostic Load***.

Diagnostic Write of Data Array:

TABLE 2-12 Timing Diagram: Diagnostic Write of Data Array

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag, VUAD read	disable way sel gen stall c1 instruc.	Gen way sel from decoded address Xmit way sel to l2d	Xmit way sel inside l2d	write data array cyc 1	write data array cyc 2	write data array cyc 3	Gen inv vector = 0's	Xmit write ack to CPX	Mux data/in v vector

Diagnostic write access to the L2 data array is done through 64 bit store that access a 32 bit data subblock along with the corresponding 7 bit ECC.The instruction that gets used is ***Diagnostic Store***.

Diagnostic Read of Tag Array

TABLE 2-13 Timing Diagram: Diagnostic Read of Tag Array

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
diagnostic decode	mux px2 index	read tag array	prepare way mux selects	mux out tag and flop	stage data	stage data	stage data	xmit req to CPX mux with data from other srcs (diagnostic/VUAD diagnostic/return data/inv vector)	xmit tag data to CPX

Diagnostic read to the L2 tag array is done through 64 bit read that accesses the tag along with the corresponding 6 bit ECC. The instruction that gets used is ***Diagnostic Load***.

Diagnostic Write of Tag Array:

TABLE 2-14 Timing Diagram: Diagnostic Write of Tag Arra

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
do nothing	enable write into the tag stall pipe for 3 cycles	write into the tag	do nothing	do nothing	do nothing	do nothing	do nothing	xmit request to cpx	xmit ack to CPX

Diagnostic write to the L2 tag array is done through 64 bit write that accesses the tag along with the corresponding 6 bit ECC.The instruction that gets used is ***Diagnostic Store***.

Diagnostic Read of VD/UA Array:

TABLE 2-15 Timing Diagram: Diagnostic Read of VD/UA Array

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
VUAD array read	stall pipe for 4 cycles mux out appropriate bits (VD or UA based on address)	flop o/p data	flop o/p data	flop o/p data	flop o/p data	flop o/p data	flop o/p data	xmit req to requesting cpx mux VUAD data with data from other srcs	xmit data to cpx

Diagnostic read to the L2 VD/UA arrays is done through a pair of address access ranges. The first accesses the valid and dirty bits for an entire set plus the parity for each of those bits across the set via 64 bit read. The second range accesses the AU bits for the entire set via 64 bit read.The instruction that gets used is ***Diagnostic Load.***

Diagnostic Write of VD/UA Array:

TABLE 2-16 Timing Diagram: Diagnostic Write of VD/UA Array:

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
do nothing	stall pipe for 4 cycles	stage wr data	stage wr data	vuad write	do nothing	do nothing	do nothing	xmit req to CPX	xmit ack to CPX

Diagnostic write to the L2 VD/UA arrays is done through a pair of address access ranges. The first accesses the valid and dirty bits for an entire set plus the parity for each of those bits across the set via 64 bit write. The second range accesses the used and allocate bits for the entire set via 64 bit write.The instruction that gets used is ***Diagnostic Store***.

Block Loads:

The core issues four 16B loads non-atomically. The L2 treats them as four separate load instructions.

Block Stores:

- issued by the core with bis bit (bit 114) and bst bit (bit 115) both =1 on the PCX packet.
- On a hit in L2, each store behaves like a normal store.
- On an L2 miss, line is fetched from DRAM, allocated in L2 and updated in L2.
- On directory CAM access, all matching L1's are invalidated. In addition, the entry in the directory that got hit also gets invalidated.

Block Init Stores:

- issued by the core with the bis bit (bit 114) of PCX packet = 1.
- On a hit in L2, each store behaves like a normal store.
- On a miss in L2, if PA[5:0] = 0, the line is initialized with all 0's by issuing a dummy read request to MCU instead of a regular read request, followed by the store happening.
- On a miss in L2, if PA[5:0]!= 0, it behaves like a regular store miss, in that it fetches the line from DRAM and then writes to it.
- On directory CAM access, all matching L1's are invalidated. In addition, the entry that got hit in the directory also gets invalidated.

Data Array Scrub:

Data array scrubbing refers to recomputing ECC for data across all ways in a particular index, detecting and correcting error in either data or ECC for each way. OpenSPARC T2 L2 data uses SEC/DED ECC (has a single bit error correction and double bit error detection). Through scrubbing single bit errors get corrected and double bit errors get flagged.

The Data Array in L2 gets scrubbed at regular (programmable) intervals after a data fill operation under CSR control. If the CSR bit enabling the Scrub Mode bit is on, then after a programmed interval of time a bit called TECC gets set by the Scrub controller logic. The scrubber gets called on the first fill with this bit set and starts the scrub operation for an index (which increments with every scrub routine and has no relationship to the fill index) right after the fill completes in the pipe. It scrubs 64 bits of data at a time for each way, so it takes (8 x 16 ways) = 128 back to back scrub

operations to complete the data scrub of that particular index in the L2 bank. While the scrub is going on the pipe stays stalled. Timing Diagram shows a typical data scrub operation following a fill.

TABLE 2-17 Timing Diagram: Fill

C1	C2	C3	C4	C5	C52	C6	C7	C8
tag write	stall four cycles	Xmit FB entry to l2b Fill Op with TECC = 1	way sel xmit to l2d read FB	VUAD write Xmit way sel inside l2d mux fbdata with l2d	data array wr cyc1	data array wr cyc2	data array wr cyc3	
	Start scrub FSM. stall pipe (cnt=0)	(cnt=1)	Setup tag read with scrub idx (cnt=2)	Read Tag Read Valid bit (cnt=3)	Gen scrub way (cnt=4)	Xmit Scrub way to l2d (cnt= 5)	Scrub read 1 (cnt = 6)	Scrub Read 2 (cnt=7)

TABLE 2-18 Timing Diagram: Data Scrub

C1	C2	C3	C4	C5	C52	C6	C7	C8
Scrub Read 3 (cnt=8)	Xmit to l2t (cnt=9)	Ecc corr. (cnt = 10)	Mux out 64 bit (cnt = 11)	Mux with c1 inst data Perform stecc & gen waysel l2d_wr & col_off (cnt = 12)	way sel xmit to l2d (cnt = 13)	Xmit way sel inside l2d (cnt = 0) Start scrub FSM. stall pipe	Scrub data array wr cyc1 (cnt = 1)	Scrub data array wr cyc2 (cnt = 2) Setup tag read with scrub idx

Tag Array Scrub:

Tag array scrubbing refers to recomputing ECC for tag across all ways in a particular index, detecting and correcting error in either tag or ECC for each way. OpenSPARC T2 L2 tag uses SEC ECC (has a single bit error correction, no double error detection). Through scrubbing single bit errors get corrected.

Once a parity error is detected in C2 on any Tag entry, TECC is marked as 1 in the instruction that gets written to the Miss Buffer in C3. Then the scrub instruction gets issued from the MB and enters the pipe. All 16 ways for the index with parity error

gets scrubbed, so the scrub operation takes a total of (8x16) = 128 L2 clocks during which the L2 pipeline stays stalled. Timing diagram TABLE 2-20 shows a typical Tag Scrub operation.

TABLE 2-19 Timing Diagram: Tag Scrub Operation

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tecc inst from MB assert pipe stall	Setup Tag read of corrupted index (cnt=0)	Setup Index (cnt = 1)	Tag Read (cnt = 2)	Setup Muxsel (cnt = 3)	Mux Tag (cnt = 4)	ECC corr. (cnt = 5)	Setup Write Index (cnt = 6)	Tag Write (cnt = 7)	Setup Tag read of corrupted index (cnt=0)

VUAD SBE Correction:

OpenSPARC T1 protects VD and UA arrays with parity. Parity check happens in C2, and if an error is detected, a fatal error trap gets taken. So for a false hit in C1, the read/write happens, but the machine gets fatal error trap and resets. Since this is one of the largest sources of fatal errors, OpenSPARC T2 would protect the VD and UA arrays with SEC DED ECC.

For every set, there would be 7 ECC bits with 16 Dirty bits and 16 Valid bits (i.e 32/7 ECC). Also for every set, there would be 7 ECC bits with 16 Allocate bits and 16 Use bits (i.e 32/7 ECC). Note that the 7th ECC bit is also the parity bit across 32 data bits and 6 ECC bits to detect double bit error. So a total of (39 + 39) = 78 bits of storage per set. Even though the Used bits need not be ECC protected (since their value is non-critical: any error in the used bits will cause potentially different replacement order, but still functionally correct operation), since VD and UA arrays would be implemented out of the same Register File array, L2 would protect the Use bits and the Allocate bits with ECC.

For any instruction from core or from SIU, the VD and UA arrays get read in C1 stage of the L2 pipe and get muxed with forwarded VD,UA bits from prior instructions in the pipe that are to the same index. The output of the mux gets written to a C2 flop. In case this C1 mux select points to the leg coming from the VD/UA arrays, ECC would get checked in C2 on the data from the arrays. The data will get corrected in C2 stage itself (for a single bit error) and will get written back to the VD,UA arrays with regenerated ECC in C5 stage of the pipe for all instructions other than diagnostic accesses. If a double bit (Uncorrectable) error gets detected in any one of VD or UA arrays, L2 will log LVU in L2 Error Status register which will cause L2 to assert fatal error reset request to the Reset block. The execution recovery and logging mechanism for Correctable VD/UA errors in L2 is as follows:

1. If the instruction was any flavor of load, store, atomic, ifetch from core, L2 would detect the Correctable error in C2 and log it as LVC (VUAD correctable error) with the syndrome in L2 Error Status register in C9 stage of the same pass.The PA[39:0] (index [8:0]) would be captured in the L2 UE/CE address register. The caming of the L2 directories, updates of the L2 directories and dispatch of crossbar packets back to cores would be gated off by the correctable error.If the error resulted in a false hit (tag match but valid = 1 while it should be 0), the data array operation (load or store) would still happen though the crossbar packet would be gated off. There is no memory corruption issue as store to an invalid way does not cause data corruption. Also the instruction would be moved into the Miss Buffer and readied for resissue in C9. However the DRAM ready bit would not be set in the Miss Buffer thereby disabling dispatch of requests to MCU.Once the instruction gets reissued down the L2 pipe, it would see corrected data in the VD/UA arrays and would execute properly based on correct state of the L2 lines at that index. The LVC error that got logged in the first pass would get reported to the Virtual Core specified in the L2_CONTROL_REG.ERRORSTEER field on bits [139:138] of Error Indication packet of crossbar after the occurrence of the next L2 fill if error reporting is enabled.

2. If the instruction was a RDD or WR8 or WRI from SIU, L2 would detect the Correctable error in C2 and log it as LVC (VUAD correctable error) with the syndrome in L2 Error Status register in C9 stage of the same pass .The PA[39:0] (index [8:0]) would be captured in the L2 UE/CE address register. The caming of the L2 directories, updates of the L2 directories and dispatch of data return and write ack packets in the first pass back to SIU would be gated off by the correctable error. If the error resulted in a false hit (tag match but valid = 1 while it should be 0), the data array operation (load or store) would still happen though the SIU packet would be gated off. There is no memory corruption issue as store to an invalid way does not cause data corruption.Also the instruction would be moved into the Miss Buffer and readied for reissue in C9. However the DRAM ready bit would not be set in the Miss Buffer thereby disabling dispatch of requests to MCU.Once the instruction gets reissued down the L2 pipe, it would see corrected data in the VD/UA arrays,would execute properly based on correct state of the L2 lines at that index and would return packets (read data, wri ack,wib_dequeue) to SIU. The LVC error that got logged in the first pass would get reported to the Virtual Core specified in the L2_CONTROL_REG.ERRORSTEER field on bits [139:138] of Error Indication packet of crossbar after the occurrence of the next L2 fill if error reporting is enabled.

3. If the instruction is of any of the types mentioned in (1) and (2) but issued from the Miss Buffer, all the things mentioned in (1) and (2) happen with one exception: the instruction does not get re-inserted into the Miss Buffer, but the valid bit in the Miss Buffer stays set, and the instruction gets replayed again through the vuad_ce_rdy and vuad_ce_replay bits being set in the Miss Buffer.

4. If the instruction was an evict instruction issued from the Miss Buffer and there is a VUAD CE detected and the instruction is not a Prefetch ICE instruction, the Evict Bit and Evict Ready bit stay set in the Miss Buffer, so that the eviction gets replayed from the Miss Buffer. By this time the VUAD CE has been corrected and the eviction happens as normal. Also in the eviction pass that detected the VUAD CE, the caming of the L2 directories, updates of the L2 directories and dispatch of crossbar packets back to cores and copying of the line to the Write Back Buffer are gated off by the correctable error.

5. If the instruction was any flavor of diagnostic, Icache Invalidate, Dcache Invalidate, Tecc, Fill, Prefetch ICE or replayed instruction from Miss Buffer hitting in the Fill Buffer Tags, L2 would not detect the Correctable error and would not log LVC. Also the instruction will proceed as normal as if nothing happened and would send back responses to the cores as normal. However for all of the above mentioned instructions other than Diagnostics, the corrected data would get written in the VUAD arrays in C5, basically doing a silent correction.

Timing Diagram TABLE 2-20 shows the error detection and correction pass for VUAD single bit correctable error for loads,stores,ifetches,atomics,wr8's,rdd's and wri's.

This pass gets followed by the instruction reissue from Miss Buffer by which time

TABLE 2-20 Timing Diagram: VUAD SBE Error Detection and Correction

C1	C2	C3	C4	C5	C52	C6	C7	C8	C9
tag read VUAD read VUAD bypass	VUAD ECC check Single Bit Error Detected	Write Instruction to MB Set DEP bit Disable SIU ack gen logic, IOWB eviction logic for WRI's	Gate of I$,D$ Dir CAM , I$,D$ Dir Update	Vuad write (corr-ected data			Gate off crossbar request		Ready Instruction for reissue Do not set DRAM_READY

the error is already corrected, and the instruction executes normally as shown by earlier pipe diagrams in the document.

Ordering of future instructions in L2 in the case of a VUAD SBE:

All instructions other than diagnostic, Icache Invalidate, Dcache Invalidate, Tecc, Fill or Prefetch ICE (and not issued from the Miss Buffer) that would detect VUAD SBE would set the DEP bit when they get inserted into the Miss Buffer. This would guarantee that a future instruction to the same PA would hit in the Miss Buffer and see the DEP bit set and would also get moved into the Miss Buffer. Also, this future instruction would not be issued to the pipe until the offending instruction has been issued to the pipe and its DEP bit cleared. This would maintain ordering and remove hazards. However it is possible that instructions to different PA would send acks and data back to crossbar and SIU out of order w.r.t order of arrival to L2. For loads, ifetches, atomic reads and rdd's this would not be an issue. However for stores out of order acks may be an issue if loads following the stores return old data instead of new data (this would have caused TSO ordering violation). But since we put the offending instruction in the Miss Buffer and set the DEP bit, loads would always be ordered after stores to the same address and would not complete until the store completes, and hence would return new data.

Also in case the instruction above hits against another address in the Miss Buffer and detects a VUAD CE, the instruction would set its DEP bit and get inserted into the Miss Buffer,but would not get replayed immediately. It would get replayed only when its dependency cleared. In this case the instruction would not set the vuad_ce_rdy and vuad_ce_replay bits in the Miss Buffer.

For any instruction other than diagnostic, Icache Invalidate, Dcache Invalidate, Tecc, Fill or Prefetch ICE and issued from the Miss Buffer that detects VUAD SBE, the DEP bit would not get set, however the Valid bit would not be cleared and the instruction would not get re-inserted into the Miss Buffer. The instruction would get replayed from the Miss Buffer through the vuad_ce_rdy and vuad_ce_replay bits being set in the Miss Buffer.

Tag Parity Error and VUAD Error detected in single pass:

Any instruction from core or SIU other than Diagnostics, Prefetch ICE and I$,D$ invalidates can detect tag parity error and VUAD SBE. It is architecturally possible for such an instruction to detect a tag parity error and VUAD SBE in the same pass. If that happens, the recovery and correction mechanism will be as follows:

1. Instruction would be inserted into the Miss Buffer in the first pass (in which it detected tag parity error and VUAD ce) and will be readied for reissue not in C9 but by the scrub instruction that is to follow.
2. The VUAD SBE would get corrected in the first pass itself (with the VUAD array getting updated in c5).
3. A scrub instruction will get issued from Miss Buffer which will walk down L2 pipe and do the scrub of the tag array

4. Original instruction will get issued from the Miss Buffer and this time will detect neither tag parity error nor VUAD SBE and will complete as normal.

2.1.4.11 L2 Interactions with SIU (System Interface Unit)

Block Reads (RDD's):

Block Read from SIU goes through L2 pipe like a regular load from the core. On a hit, 64 B of data is returned to SIU. On a miss, L2 does not allocate, but sends a non-allocating read to DRAM. It gets 64 bytes of data from DRAM and sends it back to SIU (read once data only) directly without installing in the L2 cache.T

TABLE 2-21 Timing Diagram: Block Reads

C1	C2	C3	C4	C5	C52	C6	C7	C8	
tag,VUAD read VUAD bypass tag compare Check ECC for Tags MB CAM and MB hit logic FB CAM WBB CAM	way sel logic xmit way sel to l2d rd/wr! Gen,xmit VUAD Ecc check stall next instruction	way sel xmit in l2d FB data read enable (on a miss)	data array read cyc1 FB data read cycle 1 write 64 B Fbdata to l2d flop	data array read cyc2 VUAD write stage FB data	data array read cyc3 mux with FB data write 64 B data to flop in l2d	data Xmit cyc 1 write 64 B data to flop in l2b	stage 64 B data in flop in l2b	32:1 Mux to get 32 bits of data from 64 bytes (in l2b,critical 32 bits first)	Check ECC on data (32 bits) flop data in l2b

In C10, L2 starts issuing data return to SIU 32 bits per clock from l2b block. While processing a block read from SIU, the L2 arbiter does not accept any other SIU read or write request until the block read is complete. This is because there is not enough queue space within L2 to hold the data that is getting streamed out in case a new block read request comes from SIU.

Write Invalidates (WRI's):

For a 64 B write (write invalidate from SIU), the SIU issues a 64 B write request to L2. The data goes to IOWB and waits there until the write makes it through the pipe after resolving any dependencies with the Miss Buffer entries (resolves ordering issues w.r.t prior accesses from the CPU to the same line). Once this happens, the IOWB empties its contents to DRAM, after arbitrating with the WBB.

When the write progresses through the pipe, it looks up the tags. If tag hit, it invalidates the entry and all L1 entries that match. If tag miss, it does nothing (just comes down the pipe) to maintain order.The only two cases where a WRI gets put into the Miss Buffer are on tag parity error (potential false miss case) or VUAD SBE (can be anything: true miss, false miss, true hit, false hit).

Partial Line Writes (WR8's):

When the SIU issues 8B writes to L2 with random byte enables, the L2 treats them just like 8B stores from core (i.e does 2-pass partial store if odd number of byte enables are active or if misaligned access, otherwise regular store). Data gets committed to L2 cache.

2.1.4.12 L2 Pipeline Stalls

- Same column stall - Each column (sub-bank) of the data array (also referred to as subbanks) requires two cycles to access. Therefore, the same column cannot be accessed in consecutive cycles. A one cycle stall is inserted if a collision is detected.
- Ifetch - Ifetch operations require two reads of the data array. A one cycle stall is inserted for any ifetch operation.
- Fill - Fill operation stalls the pipe for stall three cycles
- Eviction - Eviction operation stalls pipe by two cycles
- Diagnostics
- Tag/Data array scrubs

2.1.5 Functional Description of Sub-blocks

The L2 Cache Unit is composed of the following sub-blocks:

- L2 Tags
- L2 VUAD
- L2 Data

- Directory
- Input Queue (IQ)
- IU Queue (SIUQ)
- Output Queue (OQ)
- Arbiter
- Miss Buffer (MB)
- Fill Buffer (FB)
- Writeback Buffer (WBB)
- I/O Write Buffer (IOWB)

2.1.5.1 L2 Tags

TABLE 2-22 shows the physical address mapping for the L2 cache.

TABLE 2-22 Physical Address Mapping for the L2 Cache

39	18	17	9	8	6	5	4	3	0
tag		index		L2 bank		subbank addr		16b offset	

Given a bank of 512 KB with 64 B lines, the tag index is bits <17:9>. Each tag entry contains address<39:18> + 6 ECC bits. The state of each line is maintained using valid (V), used (U), allocated (A), and dirty (D) bits. These are stored in the VUAD array.

Each 22 bit tag is protected by 6 bits of ECC. A 16 way 27 bit compare (with the appropriate bits from the issuing instruction) is performed to generate the way selects for accessing the data array. This approach removes error detection from the "tag to data" critical path.

Thus total Tag Memory per bank of L2 is 28 KB for 64 B line size.

L2 Tag ECC

The L2 tag arrays are protected by ECC. For every 22 bits of tag, there are 5 SEC ECC bits and 1 parity bit (which covers all 27 bits). OpenSPARC T1 L2 does not detect Uncorrectable (Double Bit) errors for tag and OpenSPARC T2 L2 won't either (unless we get Epic 9 data which shows high enough FIT rate).

In pipe stage C1, {22 tag bits, 5 ECC bits} get compared with corresponding 27 bits in all of the 16 ways in the set. This prevents a false hit from happening. If {22 tag bits, 5 ECC bits} match in one entry, then it is a true hit. If a true hit does not happen, then it is miss. As the instruction moves to pipe stage C2, parity is recommitted for

each of 16 ways over {22 tag bits, 5 ECC bits}. If there is a parity error detected in C2, the instruction is moved into Miss Buffer and a scrub instruction is issued from the Miss Buffer to scrub and correct ECC and parity for any entry in error. After the scrub instruction is complete, the original instruction gets re-issued from the Miss Buffer.

Note that if the instruction did not hit in any of the tags in C1, there are 4 possibilities:

- there is no parity error in C2. In which case it would be a true miss and go to DRAM.
- There is a parity error in C2 and one of the ways differs by 1 bit only in the tag field w.r.t the instruction, then this could be a case of a false miss (if upon scrubbing this bit that is different gets chosen as the bit to be flipped). Then the instruction after getting replayed from the Miss Buffer will hit in L2.
- There is a parity error in C2 and one of the ways differs by 1 bit only in the tag field w.r.t the instruction, but the error is in one of the ECC bits or on another data bit, in which case the address is different, and it will miss in L2 after getting replayed from Miss Buffer and will go to DRAM.
- There is a parity error in C2 and instruction tag mismatches by more than 1 bit with each way, it will miss in L2 after getting replayed from Miss Buffer and will go to DRAM.

Note – What happens if tag hits in 2 ways ? OpenSPARC T2 L2 response is indeterministic. The SRAM circuits are protected from getting burnt out in such a case. The functional behavior of L2 is not defined.

2.1.5.2 L2 VUAD

This 4.9 KB (including ECC) dual ported array is used to maintain the state of every line in the L2 cache for each bank. The state of each line is maintained using the Valid (V), Used (U), Allocate (A) and Dirty (D) bits. Allocate bit indicates that the marked line has been allocated to a miss. This bit is also used in the processing of some special instruction's such as atomics and "partial" stores (since these do read-modify-writes, which involve 2 passes through the pipe, the line needs to be locked until the second pass completes; otherwise the line may get replaced before the second pass happens). The Used bit is a reference bit used in the replacement algorithm.

The Allocate bit (per way) gets set when a line gets picked for replacement. For a load or ifetch, it gets cleared when fill happens, and for store when store completes. The Used bit gets set when any store/load hits (1 per way). Used bits get cleared (all 16 at a time) when there are no unused or unallocated entries for that set. The dirty

bit (per way) gets set when a stores modifies the line. It gets cleared when the line is invalidated. The valid bit (per way) gets set when a new line is installed in that way. It gets reset when that line gets invalidated.

The L2 uses a Round Robin algorithm to pick replacement candidates. When there is a L2 miss and a line needs to be selected for replacement, in reference to a dynamic pointer which can point to any one of the 16 ways, the first entry with (A = 0 and U = 0) otherwise (A = 0) gets picked for replacement. All 16 ways get looked at in a wrap-around fashion starting from the way that is currently pointed at by this pointer.

The algorithm used to select a way, out of 16 ways, to be evicted out of the L2 cache is not a true LRU algorithm but Round Robin arbitration. Round Robin arbitration is done in two stages by dividing 16 ways in 4 quads of 4 ways each. First Round Robin is done within each quads to select one of the 4 ways and then Round Robin is done to select one of the four quads. A 4 bit state register is kept for each quad at each level. A one on a bit location corresponding to a way represents highest priority for that way. Every time an eviction takes place, state register is updated by shifting it left by one bit otherwise state of the register does not change. State register is used in C2 for the way selection and it is updated in the C3. On reset state register is initialized to a state such that way0 has the highest priority.

Way selection algorithm depends on the Used and Allocate bit of the VUAD array, read during C1, for the way selection. First priority is given to the ways that has not been Used and has not been Allocated for the eviction in the previous cycle. If there is no Unused and Unallocated way then a way that has not been previously Allocated is given preference. Invalid bit is not used for the way selection as if a way is Invalid then its Used bit will not be set, so checking Invalid bit is redundant.

Note – What happens if all 16 ways have A = 1? No new instructions can enter the pipe until at least one A = 0. This is guaranteed by MB stalling the pipe speculatively. Since MB is 32 entries in OpenSPARC T2, and since there are 16 ways per set, the L2 control logic will detect 12 entries of the same index in MB and speculatively stop accepting requests from PCX and SIUQ (this accounts for 4 instructions in flight in PX2,C1,C2,C3 that can take the count to 16 misses to the same index). The stall to PCX/IQ and SIUQ requests lasts for until the number of entries of the same index in MB reaches 11.

2.1.5.3 L2 VUAD ECC

OpenSPARC T1 protects VUAD array with parity. Parity check happens in C2, and if an error is detected, a fatal error trap gets taken. So for a false hit in C1, the read/write happens, but the machine gets fatal error trap and resets. Since this is one of the largest sources of fatal errors, OpenSPARC T2 would protect the VD and UA arrays with SEC DED ECC.

For every set, there would be 7 ECC bits with 16 Dirty bits and 16 Valid bits (i.e 32/7 ECC). Also for every set, there would be 7 ECC bits with 16 Allocate bits and 16 Use bits (i.e 32/7 ECC). Note that the 7th ECC bit is also the parity bit across 32 data bits and 6 ECC bits to detect double bit error. So a total of (39 + 39) = 78 bits of storage per set. Even though the Used bits need not be ECC protected (since their value is non-critical: any error in the used bits will cause potentially different replacement order, but still functionally correct operation), since VD and UA arrays would be implemented out of the same Register File array, L2 would protect the Use bits and the Allocate bits with ECC.

For any instruction from core or from SIU, the VD and UA arrays get read in C1 stage of the L2 pipe and get muxed with forwarded VD,UA bits from prior instructions in the pipe that are to the same index. The output of the mux gets written to a C2 flop. In case this C1 mux select points to the leg coming from the VD/UA arrays, ECC would get checked in C2 on the data from the arrays. The data will get corrected in C2 stage itself (for a single bit error) and will get written back to the VD,UA arrays with regenerated ECC in C5 stage of the pipe for all instructions other than diagnostic accesses. If a double bit (Uncorrectable) error gets detected in any one of VD or UA arrays, L2 will log LVU in L2 Error Status register which will cause L2 to assert fatal error reset request to the Reset block. If L2 detects Correctable SBE in C2, it will log it as LVC (VUAD correctable error) with the syndrome in L2 Error Status register in C9 stage of the same pass.The index [8:0] would be captured in the L2 UE/CE address register.

2.1.5.4 L2 Data

Each L2 data array bank is a single ported SRAM structure capable of performing the following operations:

- 16B read
- 64B read
- 8B write with any combination of word enables
- 64B write (with any combination of word enables). However fills would update all 64 bytes at a time.

Each L2 bank is 512 KB in size, with each logical line 64 B in size.

Each L2 data array bank is further subdivided into four sub-banks, also referred to as columns, each 16 B in width. These sub-banks are accessed based on bits <5:4> of the physical address. Loads (which are a maximum of 16 B in size) and stores (maximum of 8 B in size) access one subbank. Cache fills and line evictions are 64 B in size, and access four sub-banks per cycle.

Any L2 cache data array access takes two cycles to complete, so no sub-bank can be accessed in consecutive cycles. All access can be pipelined except, back to back accesses to the same sub-bank.

Each 32b word is protected by 7 bits of SEC/DED ECC. (Each line is 32 x [32 + 7 ECC] = 1248 bits). All sub-word accesses require a read modify write operation to be performed and are referred to in this document as "partial stores".

2.1.5.5 L2 Directory

The directory maintains a copy of the L1 tags for coherency management and also ensures that the same line is not resident in both the icache and dcache (across all cores). The directory is split into an icache directory (icdir) and a dcache directory (dcdir), which are similar in size and functionality.

The directory is written only when a load is performed. On certain data accesses (loads, stores and evictions), the directory is camed to determine whether the data is resident in L1 caches. The result of this CAM operation is a set of match bits which is encoded to create an invalidation vector to be sent back to the SPARC cores to invalidate L1 lines.

- Loads - The icdir is CAMed to maintain I/D exclusivity. The dcdir is updated to reflect the load data that fills the L1 cache.
- IFetch - The dcdir is CAMed to maintain I/D exclusivity. The icdir is updated to reflect the instruction data that fills the L1 cache.
- Stores - Both directories are CAMed. This ensures that (i) if the store is to instruction space, the L1 icache invalidates the line and does not pick up stale data; (ii) if a line is shared across SPARC cores, the L1 dcache invalidates other cores and does not pick up stale data; and (iii) the issuing core has the most current information on the validity of its line.
- Evictions from the L2 cache - Both directories are CAMed to invalidate any line that is no longer resident in the L2.

2.1.5.6 Directory Organization

D$ Dir:

There are 8 cores in OpenSPARC T2. L1 Dcache for each core has 128 sets, each set has 4 ways. Since L1 Dcache line size is 16 B, this gives a total of (8 x 128 x 4 x 16) bytes = 64 KB or 4K L1 lines in all cores together. Each L1 Dcache is 8KB.

Thus each L1 Dcache will map (128/8) 16 sets to each L2 bank. So for each L2 bank, the DCache directory will consist of (16 x 8) sets of L1 Dcache lines for all cores combined. This gives a total of 512 L1 Dcache lines per bank.

These 512 L1 Dcache line mappings gets organized physically in the DCache directory as follows:

Each Dcache directory has 16 panels arranged as 4 rows and 4 columns. Row gets accessed by address[5:4], column by address[10,9]. Each panel has 32 entries indexed by {cpu_id(3 bits), replacement way(2 bits)}.

For an update related to a load, the panel is accessed by address {10,9,5,4} and the entry within the panel to be updated is selected by {cpu_id(3 bits), replacement way(2 bits)} for the load.

For a store or a ICache mutual-exclusivity check on a Ifetch, which can potentially invalidate a maximum 8 L1 cache lines (one L1 line per core), the panel gets selected by address {10,9,5,4} and all 32 entries within that panel get CAMed against the store, based on which a invalidation vector gets generated for a max of 8 L1 Dcache line invalidation's (one per core). The way number for each L1 Dcache will be encoded as a 2 bit field in the inval vector.

For an eviction, since the L2 cache line is 64 bytes, 4 panels out of 16 will get CAMed based on address[10,9] (i.e 1 column). This would mean a total of 32x4 = 128 compares to invalidate a max of 4 cache lines per L1, i.e a max of 4x8 = 32 L1 Dcache lines for all cores combined. This will come out as 32 D$ L1 lines eviction vector from L2. The way number for each L1 Dcache will be encoded as a 2 bit field in the inval vector.

Each entry in the directory will store {L2 index[9 bits],L2 way[4 bits], parity, valid} i.e a total of 15 bits corresponding to the location in L2 that the L1 line maps to.

For a load hit, the entry gets updated with {L2 index[9 bits],L2 way[4 bits], parity}, while on a store or eviction or a ICache mutual-exclusivity check on a Ifetch, the {L2 index[9 bits],L2 way[4 bits]} gets CAMed against the stored value of each entry.

I$ Dir:

L1 Icache for each core has 64 sets, each set has 8 ways. Since L1 Icache line size is 32 B, this gives a total of (8 x 64 x 8 x 32) bytes = 128 KB or 4K L1 Icache lines in all cores together. Each L1 Icache is 16 KB.

Thus each L1 Icache will map (64/8) = 8 sets to each L2 bank. So for each L2 bank, the ICache directory will consist of (8 x 8) sets of L1 Icache lines for all cores combined. This gives a total of 512 L1 Icache lines per bank.

These 512 L1 Icache line mappings gets organized physically in the ICache directory as follows:

Each Icache directory has 16 panels arranged as 4 rows and 4 columns. Row gets accessed by {address[5],I$ replacement way[2]}, column by address[10,9]. Each panel has 32 entries indexed by {cpu_id(3 bits), I$ replacement way[1:0]}.

For an update related to a Ifetch hit, the panel is accessed by {address {10,9,5}, I$ replacement way[2]} and the entry within the panel to be updated is selected by {cpu_id(3 bits), I$ replacement way[1:0]} for the Ifetch hit.

For a store or a Dcache mutual-exclusivity check on a load, which can potentially invalidate a maximum 8 L1 Icache lines (one line per core), 2 panels gets selected by address {10,9,5} and all 32 entries within each panel get CAMed against the store, based on which a invalidation vector gets generated for a max of 8 L1 Icache line invalidation's (one per core). The way number for each L1 Icache line will be encoded as a 3 bit field in the inval vector.

For an eviction, since the L2 cache line is 64 bytes, 4 panels out of 16 will get CAMed based on address[10,9] (i.e 1 column). This would mean a total of 64x2 = 128 compares to invalidate a max of 2 Icache lines per L1, i.e a max of 2x8 = 16 L1 Icache lines for all cores combined. This will come out as 16 I$ L1 lines eviction vector from L2. The way number for each L1 Icache line will be encoded as a 3 bit field in the inval vector.

Each entry in the directory will store {L2 index[9 bits],L2 way[4 bits], parity, valid} i.e a total of 15 bits corresponding to the location in L2 that the L1 line maps to.

For a Ifetch hit, the entry gets updated with {L2 index[9 bits],L2 way[4 bits], parity}, while on a store or eviction or a Dcache mutual-exclusivity check on a load, the {L2 index[9 bits],L2 way[4 bits]} gets CAMed against the stored value of each entry.

2.1.5.7 SIU Queue (SIUQ)

The SIU Queue accepts RDD,WRI and WR8 packets from the SIU and issues them to the pipe after arbitrating against other requests.L2 SIU Queue block can record up to 2 requests from SIU in it's 2 deep fifo. The requests are received serially. A counter is maintained in the SIU side incrementing on a transaction dispatch to L2 cache and decrementing upon receiving l2t_siu_iq_dequeue or l2t_siu_wib_dequeue Signals from the L2 cache. l2t_siu_iq_dequeue signal is asserted when an instruction is issued down the L2 pipe (RDD,WRI and WR8 instructions). l2t_siu_wib_dequeue is asserted when the contents of a I/O Write Buffer entry gets streamed to DRAM (WRI).

2.1.5.8 Input Queue (IQ)

The input queue is a 16 entry FIFO which queues packets arriving on the PCX when they cannot be immediately accepted into the L2 pipe. Each entry in the IQ is 130 bits wide. The FIFO is implemented with a dual ported array. The write port is used for writing into the IQ from the PCX interface. The read port is for reading contents for issue into the L2 pipeline. If the IQ is empty when a packet comes on the PCX, the packet can pass around the IQ if it is selected for issue to the L2 pipe.

The IQ asserts a stall to the PCX when 11 entries are used in the FIFO. This allows for packets already in flight as shown in .

TABLE 2-23 Input Queue Pipeline

PQ	A	B	C	D	E	F				
PA		A	B	C	D	E	stall			
PX			A	B	C	D	E			
PX?				A	B	C	D	E		
C1					A	B	C	D	E	
C@ (count)						12	13	14	15	16

2.1.5.9 Output Queue (OQ)

The output queue is a 16 entry FIFO which queues operations waiting for access to the CPX. Each entry in the OQ is 146 bits wide. The FIFO is implemented with a dual ported array. The write port is used for writing into the OQ from the L2 pipe. The read port is for reading contents for issue to the CPX. If the OQ is empty when a packet comes from the L2 pipe, the packet can pass around the OQ if it is selected for issue to the CPX.

Multicast requests are dequeued from the FIFO only if all the destination CPX queues can accept the response packet.

When the OQ reaches its high water mark, the L2 pipe stops accepting inputs from the Miss Buffer or the PCX. Fills can happen while the OQ is full since they don't generate CPX traffic.

2.1.5.10 Arbiter

The arbiter manages access to the L2 pipeline from the various sources which request access. The IQ, MB, IO interface, and FB all need access to the L2 pipe. Access to the pipe is granted based on the following priority:

- Access currently stalled in the pipe
- Second packet of a CAS operation
- SIU instruction from SIU Queue
- Miss Buffer instruction
- Fill Buffer instruction
- Instruction from the IQ
- Background scrub request

2.1.5.11 Miss Buffer (MB)

The Miss Buffer (MB) has 32 entries and stores instructions which cannot be processed as a simple cache hit. This includes true L2 cache misses (no tag match), instructions that have the same cache line address as a previous miss or an entry in the Writeback Buffer, instructions requiring multiple passes through the L2 pipeline (atomics and partial stores), unallocated L2 misses, and accesses causing tag ECC errors.

Miss Buffer in OpenSPARC T2 L2 would be 32 entries instead of 16 as in OpenSPARC T1 L2. This is needed to reduce L2 stalls due to Miss Buffer full which affects the CPI of each of the 64 threads (due to the fact that Miss Buffer going full stalls all accesses to L2; load hits and store hits cannot happen). With DDR 280 Mhz, for TPCC, CPI per thread improves by 8% between 16 and 32 entry Miss Buffer and L2 stall due to MB full reduces from 6.4% to 0.08%. With DDR 333 Mhz, for TPCC, CPI per thread improves by about 7.6% with 32 entry Miss Buffer while L2 stall due to MB full goes closer to zero.

The Miss Buffer is divided into a dual ported RAM portion which holds store data and a CAM portion which contains the address.

A read request is issued to DRAM and the requesting instruction is replayed when the "critical quad-word" of data arrives from DRAM.

All entries in the Miss Buffer that share the same cache line address are linked in the order of insertion to preserve ordering. Instructions to the same address are processed in age order whereas instructions to different addresses are not ordered and exist as a free list.

When a MB entry gets picked for issue to the DRAM (load, store, ifetch miss), the entry gets copied into the Fill Buffer and a valid bit gets set. There can be up to 8 reads outstanding from L2 to DRAM at any point of time. Data can come from DRAM to L2 out of order w.r.t the address order. When the data comes back out of order, the MB entries get readied for issue in the order of data return. This means that there is no concept of age in the order of data returns to core as these are all independent accesses to different addresses. Thus when a later read gets replayed from the MB down the pipe and invalidates its slot in the MB, a new request from the pipe will take its slot in the MB, even while an older read has not yet returned data from DRAM.

In most cases, when a data return happens, the replayed load from the MB makes it through the pipe before the Fill Request can. Hence the valid bit of the MB entry gets cleared (after the replayed MB instruction execution is complete in the pipe) before the Fill Buffer valid bit. However if there are other prior MB instructions like partial stores that get picked instead of the MB instruction of concern, the fill request can enter the pipe before the MB instruction and in those cases the valid bit in the Fill

Buffer would get cleared prior to the MB valid bit. Thus the MB valid bit and FB valid bits always get set in the order of MB valid first, FB valid later. However they can get cleared in any order.

When the MB reaches its high water mark, the arbiter no longer accepts requests from the IQ or PCX.

2.1.5.12 Fill Buffer (FB)

The Fill Buffer is an 8 entry buffer used to temporarily store data arriving from DRAM on an L2 miss request. Data arrives from DRAM in four 16 B blocks starting with the critical quad-word. A load instruction waiting in the Miss Buffer can enter the pipeline after the critical quad-word arrives from DRAM (critical 16B will arrive first from DRAM) In this case, the data is bypassed. After all 4 quad-words arrive, the fill instruction enters the pipeline and fills the cache (and the Fill Buffer entry gets invalidated). For a non-allocating read (e.g I/O read), the data gets drained from the Fill Buffer directly to the I/O Interface when data arrives, and the Fill Buffer entry gets invalidated.

When the FB is full, the Miss Buffer cannot make requests to DRAM.

The Fill Buffer is divided into a RAM portion which stores the data returned from DRAM waiting for a fill to the cache and a CAM portion which contains the address.

2.1.5.13 Writeback Buffer (WBB)

The Writeback Buffer is an 8 entry buffer used to store dirty evicted data from the L2 on a miss. Evicted lines are streamed out to DRAM opportunistically. An instruction whose cache line address matches the address of an entry in the WBB is inserted into the Miss Buffer. This instruction must wait for the entry in the WBB to write to DRAM before entering the L2 pipe.

When the WBB reaches its high water mark, the arbiter no longer issues instructions from the Miss Buffer. This stops read requests to DRAM and allow writebacks to proceed.

The Writeback Buffer is divided into a RAM portion which stores the evicted data until it can be written to DRAM and a CAM portion which contains the address.

2.1.5.14 I/O Write Buffer (IOWB)

The I/O Write Buffer is a 4 entry buffer which stores incoming data from the PCI-EX interface in the case of a 64 B write operation. Since the PCI-EX interface bus width is only 32 bits wide, the data must be collected over 16 cycles before writing to

DRAM. An instruction whose cache line address matches the address of an entry in the IOWB is inserted into the Miss Buffer. This instruction waits for the entry in the IOWB to write to DRAM before entering the L2 pipe.

The I/O Write Buffer is divided into a RAM portion which stores the data from the IO interface until it can be written to DRAM and a CAM portion which contains the address.

It is the responsibility of the IO interface to use a handshaking protocol to track the state of the IOW Buffer.

The IO interface must never issue an operation requiring the buffer when the buffer is full.

2.1.6 Unit-level Interface Signals

TABLE 2-24 Unit Level Interface Signals

Signal Name	I/O	Size	From/ To	Timing	Description
Crossbar					
l2t_cpx_req_cq	O	8	CCX		Request to be drained out of L2
l2t_cpx_data_ca	O	146	CCX		Data from L2 cache
cpx_l2t_grant_cx	I	8	CCX		Grant to gain access to crossbar
l2t_cpx_atom_cq	O	1	CCX		First packet of Imiss
l2t_pcx_stall_pq	O	1	PCX		Cannot accept any more requests to L2Cache from core since the Input fifo is full.
pcx_l2t_data_rdy_px1	I	1	PCX		Cannot accept any more requests to L2Cache from core since the Input fifo is full.
pcx_l2t_data_px2	I	130	PCX		Data bus from core
pcx_l2t_atm_px1	I	1	PCX		Indicates atomic instruction
SIU					
l2t_sii_iq_dequeue	O	1	SIU		Entry in a IOWBB array has freed. l2t is unloading a request
l2t_sii_wib_dequeue	O	1	SIU		Write invalidate buffer (size= 4x64 B cache lines) is being unloaded
l2b_sio_data	O	32	SIU		Read Data to SIU.
l2b_sio_ue_err	O	1	SIU		UE on read data to SIU
l2b_sio_ctag_vld	O	1	SIU		Ack to SIU from L2
sii_l2t_req_vld	I	1	SIU		SIU request valid.
sii_l2t_req	I	32	SIU		SIU requests L2 cache to be serviced.
sii_l2b_ecc	I	7	SIU		Data ECC
DRAM					
l2t_mcu_rd_req	O				Read request to DRAM
l2t_mcu_rd_dummy_req	O				Flush request to MCU (=COMMIT)

TABLE 2-24 Unit Level Interface Signals

Signal Name	I/O	Size	From/ To	Timing	Description
l2t_mcu_rd_req_id	O		MCU		Request id
l2t_mcu_addr	O		MCU		Read/write Address
l2t_mcu_wr_req	O		MCU		Write request to mcu
l2b_mcu_wr_data_r5	O		MCU		Write back data to memory
l2b_mcu_data_vld_r5	O		MCU		Writeback Data Valid signal
l2b_mcu_data_mecc_r5	O		MCU		Error signal for mcu
mcu_l2t_rd_ack	I	1	MCU		Read request recorded
mcu_l2t_wr_ack	I	1	MCU		Write request recorded
mcu_l2t_qword_id_r0	I	2	MCU		Quad-word number for a transaction
mcu_l2t_data_vld_r0	I	1	MCU		Valid signal with data
mcu_l2t_rd_req_id_r0	I	3	MCU		Read request ID returned
mcu_l2t_scb_mecc_err	I	1	MCU		Async Scrub Error Signals from mcu
mcu_l2t_scb_secc_err	I	1	MCU		Async Scrub Error Signals from mcu
mcu_l2b_data_r2	I	128	MCU		Fill data from mcu
mcu_l2t_mecc_err_r2	I	1	MCU		Error information
mcu_l2t_secc_err_r2	I	1	MCU		Error information
mcu_l2b_ecc_r2	I	28	MCU		SEC ECC Information

2.1.7 RAS

2.1.7.1 General Overview

The FIT rates for L2 structures in OpenSPARC T2 in Epic8c are similar to their OpenSPARC T1 counterparts. To improve FIT rates L2, OpenSPARC T2 improves protection on L2 structures already protected on OpenSPARC T1 (e.g VUAD Array).

L2 consists of the following two major structures that are candidates for protection.

The first type is 6-device, single-ported SRAM cells optimized for density, such as L2 data arrays. These SRAM cells have high Failure In Time (FIT) rates (300-400 FITs per Mb in Epic8c). All L2 SRAMs have ECC protection.

The second type is a CAM cell, whose FIT rate may be 1/2 of a standard SRAM cell. CAM cells are difficult to protect. Adding parity to a CAM cell eliminates false CAM hits due to single-bit errors, but cannot detect false misses.

In L2, only the I$ and D$ Directory CAM's are protected by parity. None of the other CAM structures in L2 are big enough to contribute to the overall FIT rate and hence are not protected by parity.

2.1.7.2 RAS support in L2 sub-blocks

L2 Data Arrays

The L2 data arrays are protected via SEC/DED ECC on a word (32 bit) basis. A correctable error on a core data read or write results in an error being logged in one of the core ESRs and, if enabled, causes a precise or disrupting trap request to the core making the request. On a load, data gets corrected (if a single bit error was detected) when returned to the core but the error still gets reported to the core on ERR bits of the CPX packet for the load return. A core data read results from an instruction cache miss, a data cache miss, an atomic operation, a partial store or a store of less than 32 bits. A correctable error on an I/O data read or write results in an error being logged in a global ESR and, if enabled, causes a disrupting trap to the core identified by the ERRORSTEER field of the L2 Control Register. Hardware corrects the error, and rewrites the L2 line with corrected data.

An uncorrectable error on an L2 data read by a core is logged to a global ESR, and, if enabled, causes a disrupting trap to the core on a store. If the core request is due to an instruction fetch or load due to data cache miss or atomic operation, the error is actually a precise trap.

In the case of an I/O read or write with an uncorrectable error, the error is logged in a global ESR, and a disrupting trap is signaled to the core identified in the ASI_CMP_ERROR_STEERING register.

L2 Tag Arrays

The L2 tag arrays are protected by SEC ECC. A correctable error is logged to a global ESR, and, if enabled, Signals a disrupting trap request to the core identified in the ERRORSTEER field of the L2 Control Register. Hardware (scrubber) corrects and re-writes the tag. The operation is completed by replaying from Miss Buffer.

OpenSPARC T1 L2 does not detect Uncorrectable double bit errors for tag and OpenSPARC T2 L2 wont either. This is because in OpenSPARC T2 with total FIT rates in the ballpark of 400, the double bit errors in the L2 tag are contributing only 0.1 FITs.

L2 VUAD Arrays

The VUAD array contains valid, used, allocated, and dirty bits. OpenSPARC T1 protects this array with parity. Any single-bit error in the valid, allocated, or dirty bits can lead to data corruption and is fatal. OpenSPARC T2 protects the VD and UA arrays via SEC DED ECC since it is one of the largest sources of fatal errors.

A correctable error in VD or UA array (LVC) is logged in L2 ESR. If enabled, the error generates a disrupting trap request to the core identified in the ERRORSTEER field of the L2 control Register. Hardware corrects the entry, and the replayed access is completed.

An uncorrectable error in VD or UA array (LVU) is also logged in L2 ESR. However this would cause OpenSPARC T2 to assert warm reset.

L2 Directories

L2 protects it's I$ and D$ directories with parity. Parity error is fatal. The L2 directory performs a background parity detect which is synchronized with a store issue down the L2 pipe. The entry being checked for parity can be reset using dbginit_l so as to make a test repeatable.

Miss Buffer

The Miss Buffer contains miss requests as well as multi-pass L2 operations. The buffer contains data and address (tag) entries. The tag array is an 32 entry CAM of 40 bits each. A false hit on a tag can result in data corruption. A false miss can also result in data corruption. OpenSPARC T1 does not protect the tags or data. OpenSPARC T2 also does not protect the data or tags due to its small contribution to the FIT rate.

Fill Buffer

The Fill Buffer contains memory read data. This data is either cacheable reads to be written to the L2, or non-allocating cacheable data forwarded to the I/O interface. The buffer contains data and address (tag) entries. The data is protected by SEC/DED ECC and ECC is checked on the way from Fill Buffer to L2 pipe. The tag array is an 8 entry CAM of 40 bits each. A false hit on a tag can result in data corruption. A false miss can result in multiple fills for the same line outstanding, reducing performance. OpenSPARC T1 does not protect the tags. OpenSPARC T2 also does not protect the tags due to its small contribution to the FIT rate.

A correctable data ECC error is logged to a global ESR and, if enabled, generates a disrupting trap request to the core identified in the ERRORSTEER field of the L2 Control Register. Hardware corrects the error before writing the data into the L2.

An uncorrectable data ECC error is logged to a global ESR, and, if enabled, generates a disrupting trap request to the core identified in the ERRORSTEER field of the L2 Control Register.

Writeback Buffer

The Writeback Buffer contains modified evicted L2 data to be written back to memory, The data portion is protected by SEC/DED ECC and ECC is checked on the way from WBB to mcu. The tag is implemented as an 8 entry CAM with 40 bits per entry. OpenSPARC T1 does not protect the tag. OpenSPARC T2 does not protect the tag due to its small contribution to overall FIT rate.

If a correctable ECC error occurs on the data, the error is logged, and, if enabled, a disrupting trap request is generated to the core identified by the ASI _CMP _ERROR _STEERING register. Hardware corrects the error before writing the data to memory.

If an uncorrectable ECC error occurs on the data, the error is logged, and, if enabled, generates a disrupting trap request to the core identified by the ASI_ CMP_ ERROR_ STEERING register.

I/O Write Buffer

The I/O Write Buffer collects I/O write data prior to writing it to DRAM. The buffer consists of tag and data sections. OpenSPARC T1 and OpenSPARC T2 protect the data with SEC/DED ECC and ECC is checked on the way from IOWB to DRAM. The tag is not protected.

If a correctable ECC error occurs on the data, the error is logged, and, if enabled, a disrupting trap request is generated to the core identified by the ASI_ CMP_ ERROR_ STEERING register. Hardware corrects the error before writing the data to memory.

If an uncorrectable ECC error occurs on the data, the error is logged, and, if enabled, a disrupting trap request is generated to the core identified by the ASI_ CMP_ ERROR_STEERING register. Software could possibly retry the write operation through the device driver.

2.1.7.3 NotDATA in L2 (new feature in OpenSPARC T2)

Uncorrectable errors are not necessarily unrecoverable. Program execution may or may not be affected depending on the not be affected depending on the condition under which the error occurs. In either case, HW must signal the occurrence of the error when it detects it to the appropriate SW error handler. However the chances for recovery are greatly enhanced if only the offended processor reports the error, and the others do not.

To meet this high level goal, as part of a requirement from SPARC SWG RAS working group to have UE from DRAM stored in L2/L3 caches as Notdata, OpenSPARC T2 L2 would support a scheme for detecting Notdata on UE from DRAM without using extra bits of storage.

The scheme involves flipping the computed ECC bits for the data with UE from DRAM and storing it with the data. The idea is when a subsequent access happens to the line in L2 that would do an ECC check, the generated ECC would be 1's complement of the stored ECC, resulting in check bits[6:0] being all 1's. This would indicate NotData.

System requires that NotData syndrome be protected from single bit errors (SBE's). Single bit error correction is not required. However it is required that the error be reported (even as UE), and that the error is never mistaken as valid data, or data with correctable errors.

The ECC logic in L2 after detecting single bit error in the data or ECC portion of the NotData packet will treat it as an uncorrectable error. A double bit error can be a problem since the ECC logic would potentially treat it as valid data with CE. But the chance of such failure is very remote and double bit error protection on NotData is not a system requirement.

The following sections describe the mechanisms of detecting UE on FIll and storing NotData in L2 and also L2 behavior on subsequent accesses to L2 from core, SIU, scrubber and DRAM (eviction) finding NotData.

Detecting UE on a Fill & storing NotData in L2:

If MCU detects UE on data return from DRAM, it will indicate UE to L2 on a 16 byte quad-word boundary and also invert all ECC bits associated with the 16 bytes of data. This data with inverted ECC bits will then get written to the Fill Buffer and eventually get written to the L2 data array on the fill. Thus on the UE, MCU will write data marked as Notdata itself into the L2 cache. However once the data is returned to L2, there are three possibilities:

1. replayed load/ifetch/atomic reads from the Fill Buffer array before the fill happens and detects UE.

In this case if an UE gets detected on the data read from the Fill Buffer, it would return load/ifetch/atomic data to Core but mark the data as UE in the CPX packet. Also the UE would be logged as DAU error in L2. The Core will take a precise trap. Then the fill would happen and store NotData in the array. Subsequent accesses would see Notdata in the L2.

2. replayed load/ifetch/atomic reads from the Fill Buffer array before the fill happens and does not detect UE but UE is another 16 byte chunk in the same line.

In that case the load/ifetch/atomic will complete as normal. Later on when the FIll happens, if the UE is on another 16 Byte chunk for the same line (as indicated by a UE bit stored in FIll Buffer from DRAM), the UE would be logged as DAU error in L2 and assert a disrupting trap to CPU. The fill would store NotData in the array so the subsequent accesses see NotData in L2.

3. 1. replayed load/ifetch/atomic reads from the data array itself after the fill.

In this case if there was UE detected during the fill, the UE would be logged as DAU error in L2 and would cause a disrupting trap to the core. The fill would store NotData in the array. Later on if the replayed load/ifetch/atomic reads the 16 Byte chunk that has NotData in it, it will return data to Core but mark the data as NotData in the CPX packet. This would cause a precise trap. All subsequent accesses that hit would also see NotData.

If the FIll was for a store miss from CPU or SIU, UE would be logged as DAU error in L2 and cause a disrupting trap. The fill would store NotData in the array so the subsequent accesses see NotData in L2.

Detecting UE on a Scrub:

When the L2 data array scrubber detects UE in a line in the data array, it will log the UE as LDSU bit in L2 and will issue a CPX Error packet to the core specified in the ERRORSTEER field of the L2 Control Register.

L2 behavior on subsequent accesses to L2 from core, SIU, scrubber and DRAM (eviction) finding NotData.

1. Load/Ifetch/atomic hit from CPU in L2 encountering NotData :

When a load/ifetch/atomic hit detects NotData, it will set NDSP bit in L2 and report NotData to the requesting core on the CPX packet.

2. Partial store hit from CPU encountering Notdata:

When a partial store hit in L2 encounters NotData it will set NDSP bit in L2 and issue a Error Indication packet to the core indicating NotData. A disrupting trap would get issued. The store will not happen in L2.

3. 4/8 byte store from CPU:

The store would complete irrespective of the NotData in the data array. ALthough NotData gets marked for all 4 byte chunks of a 16 Byte line segment, all 4 NotData segments may get overwritten with 2/4 back to back stores and would cause the NotData symptom to be lost. In that case, when the disrupting trap caused by the UE on the related Fill gets the issued, the trap handler will just cause an eviction of the dirty line to DRAM (without knowing whether it has Notdata or not). L2 will detect Notdata on the data in the eviction path and if there is no trace of any NotData, MCU will just write it out to DRAM without polluting the ECC on 16 byte boundary. Otherwise L2 will signal UE to MCU and MCU will write it out to DRAM polluting the ECC on 16 byte boundary.

4. SIU load hit in L2 encountering NotData:

SIU load hitting in L2 encountering NotData will return data marked with UE to SIU which will get propagated to PCI_EX /BSC. NDDM bit will be set in L2 and a CPX Error packet will be issued to core specified in the ERRORSTEER field of the L2 Control Register indicating NotData and a disrupting trap will be taken.

5. SIU partial store hit in L2 encountering NotData:

When a partial store from SIU in L2 encounters NotData in L2 it will set NDDM and issue a Error Indication packet to the core specified in the ERRORSTEER field of the L2 Control Register indicating NotData. The store will not happen in L2. A disrupting trap will be taken.

6. 4/8 byte store from SIU:

The store would complete irrespective of the NotData in the data array. ALthough NotData gets marked for all 4 byte chunks of a 16 Byte line segment, all 4 NotData segments may get overwritten with 2/4 back to back stores and would cause the NotData symptom to be lost.In that case, when the disrupting trap caused by the UE on the related Fill gets the issued, the trap handler will just cause an eviction of the dirty line to DRAM (without knowing whether it has UE or not). UE will detect UE on the data in the eviction path and if there is no trace of any NotData, it will just write it out to DRAM without polluting the parity. Otherwise it will signal UE to MCU and MCU will write it out to DRAM polluting the data to indicate multi-bit error.

7. Data Ram Scrubber encountering NotData in L2:

When Data Ram Scrubber in L2 detects NotData it will do nothing, and will keep the data and ECC bits unchanged.

8. Eviction from L2 encountering NotData:

When NotData is detected on evicted data from Write Back Data Buffer to DRAM, L2 would indicate to MCU UE for each 16 byte chunk same as today, and the MCU would write the data to DRAM after flipping multiple ECC bits according to Galois Field Hemming code.

L2 behavior on NotData reported from Core

1. When uncorrectable errors occur on the store data in the store buffer, core would indicate it to L2 by asserting INV bit (116) of the pcx packet to 1'b1. L2 would then accept the store and do the write, but would mark the data as NotData.
2. To deal with UE on the compare data of a CASA instruction, LSU will assert the INV bit (116) of the pcx packet for both the CAS1 and CAS2 packets. When the L2 sees this bit asserted, it will force the compare result to be "true" so that L2 will be updated. Also instead of storing the swap data, it will write NotData.

2.1.7.4 Error reporting by L2

L2 cache reports the different errors it detects to the cores through encoded ERR[1:0] field on the CPX packets.

Loads, stream loads, mmu loads, prefetches, ifetches, atomics send back UE/CE/Notdata error on the data read, on ERR[1:0] (139:138) bits of the Load return, Stream Load Return, MMU Load return, Prefetch Return, Ifill Return 1, Ifill Return 2, Swap/Ldstub return and CAS return packets respectively to the requesting virtual core.

LDAU/LDAC/NDSP errors respectively get recorded in L2 Error Status registers (please refer to *OpenSPARC T2 Programmer's Reference Manual*) for UE/CE/Notdata Error with R/W bit being a 0.

In case a UE/Notdata is detected on a CAS1 or swap/ldstub read pass, the store does not happen in L2 in the CAS2 and swap/ldstub writes passes (leaving the data and ECC unchanged). However in the CAS 2 ack packet and the swap/ldstub ack packet, ERR[1:0] (bits 139:138) gets driven as valid to the requesting virtual core with the encoding reflecting CE/UE/Notdata.

On a atomic miss, when the line is returned from DRAM, the fill always happens first and then the load of the atomic is replayed from the Miss Buffer.Now if the line has CE or UE in it, on the fill, an L2 Error indication packet will get sent to the core indicating CE or UE. Note that CE here means that the line is already corrected and error free on account of MCU cleaning it up when writing to L2. If it was a CE, the error would not be visible in the L2 after the fill as the line is already corrected, and the replayed load of the atomic will not see any error and will just return good data. The store that follows will just complete as normal without any errors. If it was a UE, the error would be still persisting in the line when the fill happens and so the

replayed load will see Notdata and will indicate a Notdata on the load return packet to the core. The store that follows will also see the error and indicate Notdata on the store ack packet. The store will not happen in L2.

Stores, Stream Stores on detecting UE/CE/Notdata on the data do not report the error on the ERR[1:0] (139:138) bits of the Store ack, Stream Store ack packets respectively. However the errors get reported on bits 139:138 of Error Indication packet later to the requesting virtual core after the occurrence of the next L2 fill. On the detection of the error, LDAU/LDAC/NDSP errors respectively get recorded in L2 Error Status registers (please refer to *OpenSPARC T2 Programmer's Reference Manual*) for UE/CE/Notdata Error with R/W bit being a 1. Stores do not happen to the L2 in case of UE and Notdata errors leaving the data and ECC unchanged.

Directory parity (LRU) error and VUAD Uncorrectable Error (LVU) cause a fatal error warm reset and does not get reported to the core on the Error Indication Packet.

All other errors (LDWC,LDWU,LDRC,LDRU,LDSC,LDSU,DAC,DAU,DRC,DRU,DSC,DSU,LTC,LVC, NDDM) would get reported to the Virtual Core specified in the L2_CONTROL_REG.ERRORSTEER field on bits [139:138] of Error Indication packet of crossbar after the occurrence of the next L2 fill.

L2 would drive bit 137 of the CPX packet on an L2 Error indication CPX packet as 1 in case of LDRC error on a SIU RDD.This would indicate to the core that core should take a SW_recoverable trap instead of a HW_corrected error trap. For all other Correctable errors asserted by L2 on the Error Indication packet, this bit will be 0 indicating HW_corrected error. Also for other error types like UE and Notdata, this bit would be driven as 0.

2.1.8 VDFT Features

The following DFT features are supported by L2 cache.

- BIST
- Full scan
- Shadow can

2.1.9 Critical Path Analysis

Following are the critical paths in OpenSPARC T1 L2 cache:

- VUAD access in C1: Memory Access(10g) + 1mm xmit (2g) + 4-1 mux(3.5g) + 4-1 mux(3.5g) + 2-1 mux(2g) = 21 gates.

- Way Sel Generation in C2: c2 compare logic(2g) + 2.32 mm xmit in the tag array (4.7g) + 600U xmit to tagctl(1.2g) + waysel logic(5g) +3mm xmit to middle of the bottom of l2d (6g) = 19 gates.

Transmit from l2t to l2d is based on the existing proposed full chip floor plan.

- Arbitration in PX to tag access: Arb logic to sel PX addr (12g) + 2.7mm xmit to VUAD (6g) + array setup (1g) = 19 gates
- Data cache access in C4, C5 and C52.
- Data return in C8. Xmit from deccdp to oqdp (2g) + 3-1 mux in oqdp (3g) + xmit from oqdp to ccx (5g) + setup in the CCX (10g) = 20 gates.

2.1.10 Performance

The following are the L2 cache performance data for 1.4 Ghz cmp clk and 333 Mhz DRAM clk:

- Load-Use latency seen by core on L1 miss and L2 hit: 21 cmp clks = (0.714x20) = 14.994 nsec

W′ |PQ PA PX1 PX2 | C1 C2 C3 C4 C5 C52 C6 C7 C8 |

CQ CA CX1 CX2 | CX3 E M B W

[where the CPU related stages are as follows:

W′ - arb for pcx

PQ - send pcx request

PA - send pcx packet

CX3 - packet received from cpx

E - IFU signaled to restart thread

M - data written to L1 cache (if load was cacheable)

B - data sent to EXU/FGU

W - data written to register file or bypassed to dependent instruction in E stage]

- Load-Use latency seen by core on L1 miss and L2 miss: 93 cmp clks= (0.714 x 93) = 66.4 nsec

W′ -> C1: 6 cmp clks

C2 -> Req to MCU: 10 cmp clks

MCU -> DRAM address on memory bus: 4 cmp clks for ack sync + 2 DRAM clks = 12.4 cmp clks

DRAM address -> first DRAM data (critical 16/32 B first) on pins: 8 DRAM clocks

= 33.6 cmp clks

MCU -> L2 first critical data: 3 DRAM clks = 12.6 cmp clks

L2 first critical data -> C2 stage of replayed Load: 4 cmp clks

C3 stage of replayed load -> return critical 16B to core: 9 cmp clks

Data seen by Core -> earliest it gets used in Core pipe = 5 cmp clks

- Peak L2 load bandwidth with back to back hits in different sub-banks: 1.4 Ghz x16 bytes = 22.4 GB/s

(16 bytes of data getting returned from L2 to core every cmp clock over Crossbar)

- L2 store bandwidth with 8 byte back to back stores that hit in L2 in different sub-banks: 1.4 Ghz x 8 bytes = 11.2 GB/s

2.2 Appendix

2.2.1 Debug mode/initialization mode

L2 cache comes out disabled after reset. One of the bits in the L2 cache control registers have to be set in order to enable L2 cache. When L2 cache is disabled, there are no accesses to L2 cache tag ram, data ram, VUAD arrays. All the instructions are treated as a miss. However, diagnostic accesses to tag ram, data ram and VUAD arrays are permitted.

The behavior of L2 cache when disabled is as follows:

In this mode the Miss Buffer operates to it's full capacity. However, the Fill Buffer is just one (line) deep. All the instructions still follow the pipeline described in chapter 5.

All the loads and stores issued to L2 cache gets recorded in Miss Buffer (MB). Loads recorded in Miss Buffer gets issued to DRAM. The (load) data returned from DRAM is recorded in the Fill Buffer (FB). The instruction gets replayed from the Miss Buffer and data gets returned from the Fill Buffer.

When a Store transaction is encountered in Miss Buffer (MB), the store data gets transferred to the Fill Buffer and a read gets issued to DRAM to fetch the line. When the read data is returned, the store transaction now gets replayed from the Miss Buffer and does a data merge with the Fill data before getting written into the Writeback Buffer (WBB). Stores are issued out of Writeback Buffer to DRAM.

2.2.2 Reset sequence for L2 cache

In L2 cache, parity bits in the tag array, valid bits in VUAD array and the directories should be initialized before L2 cache is enabled to guarantee coherency and correct functionality.

The directory valid bits are cleared with flash reset during POR_. The reset block drives the flash reset. When the valid bits are cleared (not valid) then the entries are don't care. Hence, the parity bits are not initialized to good parity. Clearing valid bits in the directory informs the L2 cache that there are no valid lines in L1.

BISI or ASI's are used to initialize the VUAD arrays by clearing all the valid bits. This informs L2 cache that there are no valid lines in L2.

BISI or ASI's are used to initialize the tag array with good parity. This eliminates the possibility of any error cases from happening.

CHAPTER 3

Memory Control Unit (MCU)

This chapter contains the following sections:

3.1 Overview

The DRAM memory control unit (MCU) interfaces to external registered DDR2 FBDs through a unidirectional high-speed link to service load and store requests from two L2 cache banks of the on-chip L2 Cache unit. Each load and store request from a L2 cache bank has a data size of 1 cache line, 64 bytes. There are 4 physical instantiations of MCU in the OpenSPARC T2 CPU.

The features of the MCU are as follows:

- Maximum memory of 128 GB per MCU branch using 8 GB DDR2 FBDs (assuming 2 Gb DRAM parts).
- Supports DDR2 SDRAM clock frequencies up to 400 MHz (800 MHz double data rate). Internally, the MCU runs at the DDR rate.
- Supports up to 16 ranks of DDR2 FBDs per channel (8 pairs of double sided FBDs).
- Supports 128 bits of write data and 16 bits ECC per SDRAM cycle and 256 bits of read data and 32 bits ECC per SDRAM cycle.
- System peak memory bandwidths (4 branches) with 800 MHz DDR parts: 50 GB/s for reads, 25 GB/s for writes.
- Uses 10-bit Southbound and 14-bit Northbound FBD channel protocols running at 12 times the SDRAM cycle rate.
- Supports DDR2 SDRAM burst length of 4 with when using both FBD channels in an MCU, burst length of 8 when using 1 FBD channel.
- ECC generation, check, correction.
- Programmable DDR2 SDRAM power throttle control.
- The FBD Hot Plug feature is not supported.

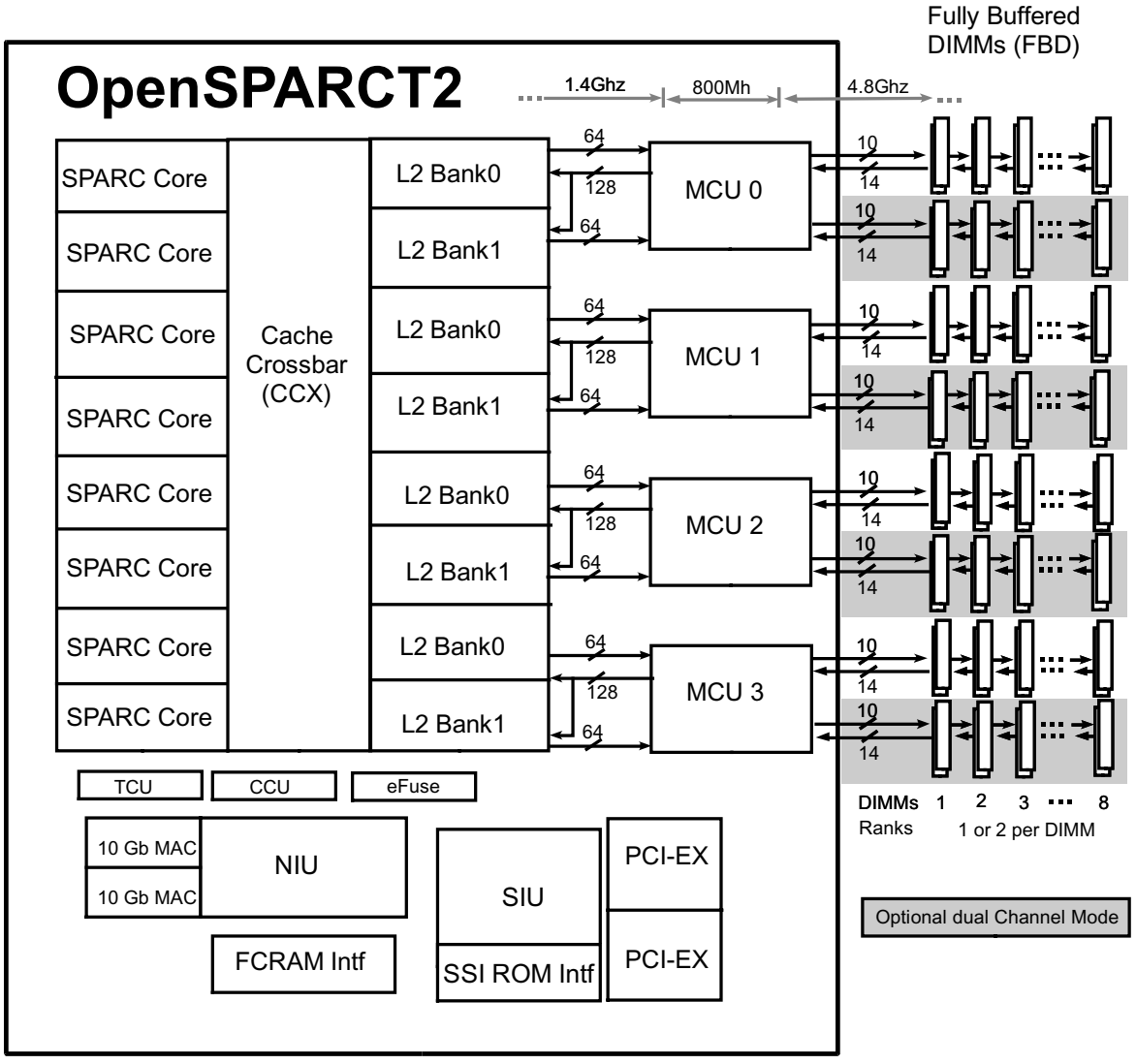
3.1.1 Changes from OpenSPARC T1 MCU design

- Use higher DDR2 SDRAM frequency: 266MHz, 333 MHz and 400MHz instead of 166MHz to 200MHz.
- Uses FBDIMM channels to access memories instead of direct DDR2 interface.
- Interface to two L2 cache banks per MCU instead of one or two L2 Cache banks interface per MCU.
- Minimum configuration with one DIMM per MCU branch.

3.1.2 Changes to OpenSPARC T2 MCU to support FBD

- Added new FBD controller with channel initialization, error detection, and frame encode and decode logic.
- Updated address decoding to support upto 16 ranks of DIMMs. Can support either one or two channels per MCU.
- Write data rate reduced to half DDR rate. Data is buffered in AMB to allow more flexibility in issuing write commands.
- Read and write operations to different DIMMs can occur in parallel. Reads and writes to a single FBD must be scheduled so that there are no data collisions on the DIMM's local DDR2 bus. However, since the Northbound and Southbound channels are independent, read data from one DIMM can be returning to the host at the same time that write data is being sent to different DIMM.
- Have separate read and write schedulers that communicate with each other to ensure there are no FBD bus data collisions.
- No dead cycle when switching read or write commands between DIMMs; however,this is still needed when switching access to the other sides of same DIMM.
- Include sync frame generation to AMBs in state machine, at least once every 42 frames.
- Remove read DQS strobe placement support. OCD and ODT support will be programmed through the AMBs.
- Spread transactions over different DIMMs instead of staying in one DIMM as long as possible to keep thermal dissipation better spread across DIMMs.
- Support L0s power saving mode.

FIGURE 3-1 OpenSPARC T2 System Overview



3.2 Terminology and Configuration

3.2.1 DRAM Terminology

- DIMM: Dual Inline Memory Module. Industry standard SDRAM module package. A stick of memory.
- Channel: Port connecting Processor chip to DIMM.
- DRAM chip: single chip inside the DIMM. We differentiate the types by how many bits it outputs and its capacity. (x4 means 4 bit output, x8 means 8 bit output, x16, x32 etc and 256Mbit or 512Mbit capacity). Most common ones are the x4, x8 output.
- Bank: Most DDR SDRAM chips are broken up into 4 or 8 logical banks internally to enable full pipelining of memory operations.
- Rank: A group of data that can be accessed from a DIMM. Each DIMM has two chip selects. When a DIMM has two ranks, each chip select accesses DRAMs on one side of the DIMM independently. When a DIMM has one rank, both chip selects must be asserted at the same time to access all DRAMs on the DIMM. For x4 SDRAMs, single rank DIMMs have 18 devices and double rank DIMMs have 36 devices.
- RAS/CAS: RAS stands for "Row Address Strobe." When this signal is asserted, a particular bank is enabled. It is also often referred to as "ACTIVE" command. CAS stands for "Column Address Strobe." When this signal is asserted, the column address and Read/Write Signals are transmitted.
- Refresh: DRAM requires what is often referred to as "REFRESH" cycle. Every row in the DRAM requires a "REFRESH" access every 15.6uS/7.8uS.
- Single-channel Mode: This is a low-power configuration with one DIMM per memory channel. Only 72 bits of the 144 external IO pins are used, and the memory burst length is 8. While it is possible to support two DIMMs per channel with this configuration, it is only expected to be used with one DIMM.

3.2.2 FBD Terminology

- Advanced Memory Buffer (AMB) - The AMB buffers memory traffic between the host and the SDRAMs. Requests are sent by the host to the AMB across a high speed link, and the AMB drives the requests to the SDRAMs using the DDR2 protocol.
- Bit Lane - A differencial pair of Signals in one direction.

- Cyclic Redundancy Code (CRC) - An error detection code sent with data across the FBD link to protect the data from errors. When a CRC error is detected, the faulty frame must be retransmitted.
- DDR Branch - A minimum aggregation of DDR channels which operate in lock-step to support error correction. A rank spans a branch. In OpenSPARC T2, a branch will consist of one or two DDR channels.
- DDR Channel - A DDR channel consists of a data channel with 72 bits of data and an addr/cntrl channel.
- DDR Data Channel - a DDR data channel consists of 72 bits of data divided into 18 data groups.
- Frame - Groups of bits containing commands or data sent across the link over 12 cycles.
- FBD - Fully Buffered DIMM.
- Link - High-speed parallel differential point-to-point interface.
- Linear Feedback Shift Register (LFSR) - A shift register where the data input to the last register is a function of the outputs of other registers.
- Northbound (NB) - the direction of Signals running from the farthest DIMM toward the host.
- Slot - Socket for a DIMM.
- Southbound (SB) - the direction of Signals running from the host controller toward the DIMMs.
- Training Sequence (TS) - A sequence of bits sent per bit lane from the host to the FBDs to initialize the channel operation.
- Unit Interval (UI) - Average time interval between voltage transitions of a signal. Approx. 200 ps for DIMMs running at 800 MHz.

3.2.3 DDR Branch Configuration

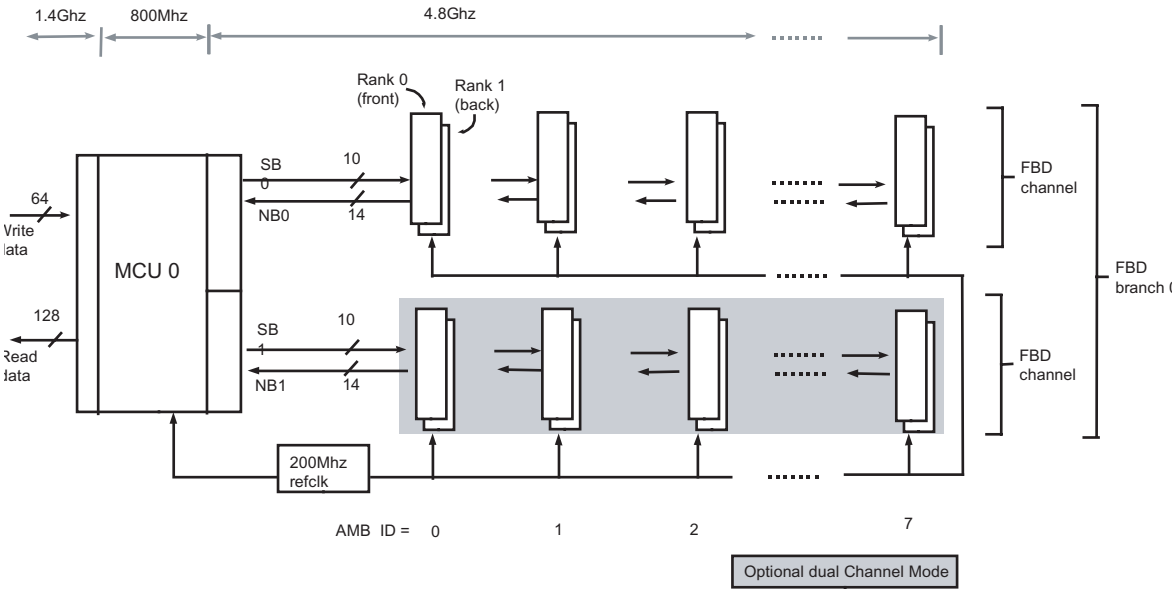
The following are key assumptions made during the design of this controller:

- x4 and x8 DIMMs are supported.
- DIMM capacity, configuration, and timing parameters cannot be different within a memory branch.
- Each DDR branch can have a different memory size and a different kind of DIMM (e.g. a different number of ranks or different cas latency). Software should not use address space bigger than 4 times the lowest memory capacity in a branch because the cache lines are interleaved across channels, and using different sized memories can create holes in the address space.
- DRAM banks are always closed after read or write command by issuing an autoprecharge command.

- Burst length is 4 (BL=4) when using a two channels per DDR branch. Burst length is 8 (BL=8) when using a single channel per branch.
- There is a fixed 1 dead cycle for switching commands from one rank on a DIMM to the other rank on the same DIMM.
- Reads, Writes, and Refreshes across DDR branches have no relationship to each other. They are all independent.

There are four independent DDR branches per CPU chip, each controlled by a separate MCU. Each branch can be configured with one or two channesl. and supports up to 16 ranks of DIMMs as shown in FIGURE 3-2. Each channel can be populated with up to 8 single- or dual-rank FBDs. When a branch is configured with two channels, the two FBDs that share the same AMB_ID are accessed in lock-step. Data is returned 144 bits per frame for 8 frames in single channel mode and 288 bits per frame for 4 frames in dual channel mode. In either mode, the total data transfer size is 512 bits, or 64 bytes, the cache line size for the L2 cache.

FIGURE 3-2 DDR Branch Configuration



Each FBD contains 4 or 8 internal banks that can be controlled independently. These internal banks are controlled inside the SDRAM chips themselves. Accesses can overlap between different internal banks. In a normal configuration, every Read and Write operation to SDRAM will generate a burst length of 4 with 16 bytes of data

transferred every half memory clock cycle. In single-channel mode, Reads and Writes will have a burst length of 8 with 8 bytes of data transferred every half memory cycle.

TABLE 3-1 Supported Memory Organization

DIMM	Base Device	Part	Ranks	# of Devices	Min. Memory per Branch	Max. Memory per Branch
512 MB	256 Mb	x4	1	18	512 MB	8 GB
1 GB	512 Mb	x4	1	18	1 GB	16 GB
1 GB	256 Mb	x4	2	36	1 GB	16 GB
2 GB	1 Gb	x4	1	18	2 GB	32 GB
2 GB	512 Mb	x4	2	36	2 GB	32 GB
4 GB	2 Gb	x4	1	18	4 GB	64 GB
4 GB	1 Gb	x4	2	36	4 GB	64 GB
8 GB	2 Gb	x4	2	36	8 GB	128 GB
512 MB	512 Mb	x8	1	9	512 MB	8 GB
1 GB	512 Mb	x8	2	18	1 GB	16 GB
1 GB	1 Gb	x8	1	9	1 GB	16 GB
2 GB	1 Gb	x8	2	18	2 GB	32 GB
2 GB	2 Gb	x8	1	9	2 GB	32 GB
4 GB	2 Gb	x8	2	18	4 GB	64 GB

3.2.3.1 Physical Address Mapping

The 40-bit physical memory address PA[39:0] request from the 8 L2 banks are decoded and mapped to one of the four MCUs by address bits PA[8:7].

39 - 9	8 - 6	5 - 4	3 - 1
DIMMs Memory Address	L2 Bank Select	L2 Cacheline Sub-Address	16-byte Offset

The L2 memory write requests are 64-byte aligned: PA[5:0] = 6'h00. A partial cache line memory write is not supported by the MCU.

The L2 physical memory read address requests are 16-byte aligned: PA[3:0]=4'h0. The read data returned will be in the following order based on the L2 cache line sub address, PA[5:4] (PA[6:4] for single-channel mode).

TABLE 3-2 Read Data Return Order for BL=4

L2 cacheline sub- address, PA[5]	16-byte data return order
0	0,1,2,3
1	2,3,0,1

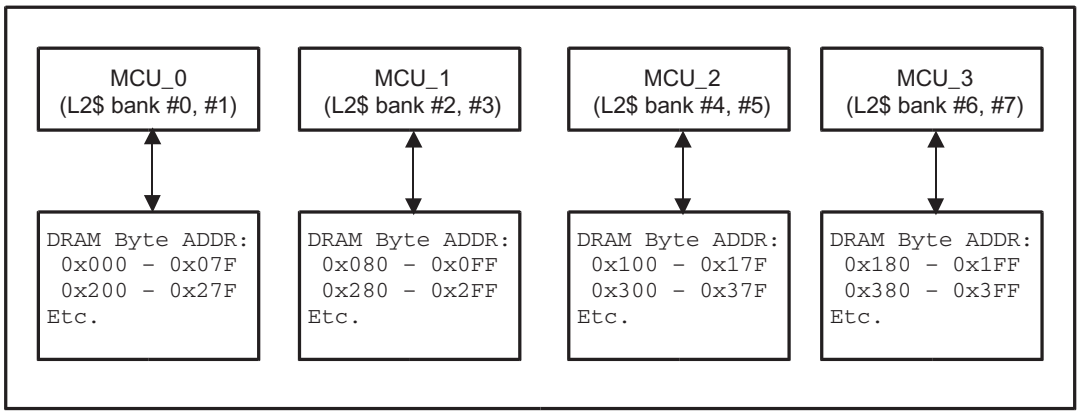
TABLE 3-3 Read Data Return Order for BL=8

L2 cacheline sub- address, PA[5]	8-byte data return order
0	0,1,2,3,4,5,6,7
1	4,5,6,7,0,1,2,3

The L2 cache bank select (PA[8:6]) is mapped to the 4 memory branches as shown in FIGURE 3-3.

FIGURE 3-3 L2 Cache Banks Memory Branch Mapping

	5	4	0
DIMMs memory address	L2 bank select	L2 cache line sub address	16-byte offset



3.2.4 FBD Channel Configuration

The FBD specification supports two southbound channel configurations and five northbound channel configurations. OpenSPARC T2 will support both southbound configurations - the 10-bit and 10-bit failover modes - and two of the northbound configurations - the 14-bit and 14-bit failover modes. These modes support data packets of 64 bits data and 8 bits ECC. The 10-bit southbound mode provides 22 bits of CRC while the 10-bit failover mode has 10 bits of CRC. The 14-bit northbound mode provides 24 bits of CRC on read data (12-bits per 72-bit data packet), and the 14-bit failover mode provides 12 bits of CRC (6 bits per 72-bit data packet).

During channel initialization, software will determine if a channel can be fully utilized (10-bit southbound or 14-bit northbound mode) or if a failover mode must be used in which one of the bit lanes is muxed out.

3.3 DDR2 FBD Usage

The following sections detail DDR2 FBD information specific to the OpenSPARC T2 MCU.

Note – "The OpenSPARC T2 Memory Control Unit (MCU) implements a DDR2 FBD design model that is based on various JEDEC-approved DDR2 SDRAM and FBDIMM standards. JEDEC has received information that certain patents or patent applications may be relevant to FBDIMM Advanced Memory Buffer standard (JESD82-20) as well as other standards related to FBDIMM technology (JESD206) (For more information, see http://www.jedec.org/download/search/FBDIMM/Patents.xls).

Sun Microsystems does not provide any legal opinions as to the validity or relevancy of such patents or patent applications. Sun Microsystems encourages prospective users of the OpenSPARC T2 MCU design to review all information assembled by JEDEC and develop their own independent conclusion."

3.3.1 FBD Channel Initialization

The FBD channels must be initialized through a software interface. This allows more flexibility in the initialization over a dedicated hardware state machine. The registers used for initialization are detailed in Section 19.6.

Software must perform the following sequence of events in order to initialize an FBD channel:

1. Drive Electrical Idle on the SB channel's TX outputs by setting the Channel State Register to 'Disable'. Channels must remain in Disable state for at least tDisable (51 frames) before transitioning to Calibrate state.
2. To transition to Calibrate state, set Channel State Register to 'Calibrate' for longer than twice tClkTrain time (42 frames). Once the AMBs are in the Calibrate state, they must remain in this state for at least tCalibrate time (480K frames).
3. Drive Electrical Idle on SB channel to transition AMBs to Disable state. Remain in Disable state for at least tDisable time (51 frames).
4. Set the Channel State Register to 'Training' to begin driving TS0 patterns on the SB channel to transition the AMBs to the Training state. The TS0 patterns are sent to the last AMB until TS0 patterns are received on the northbound channel with the AMB_ID from the last AMB. Software will use the Training State Loopback

registers to determine how many correct TS0 patterns have been received on the northbound channel. This training requires approx. 275 frames with eight DIMMs per channel. After several correct TS0 patterns have been received on 13 of 14 of the bit lanes, initialization can procede to step 5.

5. Set the Channel State Register to 'Testing' to begin driving TS1 patterns on the SB channel to transition the AMBs to the Testing state. The IBIST engine within the MCU will take over after the TS1 header has been sent, and it will signal the MCU upon its completion so the MCU can send the trailer and begin the next training sequence. After several TS1 patterns with the AMB_ID of the last AMB have been received correctly, and software/IBIST has determined that at least 9 southbound and 13 northbound bit lanes are working, initialization can procede to step 6.

6. Set the Channel State Register to 'Polling' to begin driving TS2 patterns on the SB channel to transition the AMBs to the Polling state. Continue sending TS2 patterns to the last AMB until correct TS2 patterns are received on the NB channel. This determines the read round trip delay for the channel. TS2 patterns can be sent to intermediate AMBs to determine which channel protocols they support and to check that they can properly merge their data into the NB data stream. AMBs that are not able to merge their data into the NB data stream correctly will assert their Data_Merge_Error status bit. Once initialization reaches the L0 state, software can check these bits to determine how to adjust the Command_to_Data_Incr registers in the AMBs.

Set the Channel State Register to 'Config' to begin driving TS3 patterns on the SB channel to transition the AMBs to the Config state. The TS3 patterns program the configuration of the SB and NB channels (always 10 SB and 14 SB for OpenSPARC T2) and which channel bits are muxed out if using a fail over mode. TS3 patterns are issued until the patterns are correctly received on the NB channel.

7. Set the Channel State Register to 'L0' to transition AMBs to L0 state. After 4 consecutive NOPs have been sent on the SB channel, the channel is ready to accept channel and DRAM commands.

3.3.2 FBD Commands

TABLE 3-4 FBD DRAM Commands

DRAM Cmds	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Activate	DS2	DS1	DS0	1	Addr		RS						DRAM Bank and Address											
Write	DS2	DS1	DS0	0	1	1	RS						DRAM Bank and Address											
Read	DS2	DS1	DS0	0	1	0	RS						DRAM Bank and Address											
Precharge All	DS2	DS1	DS0	0	0	1	RS	X	X	X	X	1	1	1	X	X	X	X	X	X	X	X	X	X
Precharge Single	DS2	DS1	DS0	0			RS	DRAM Bank				1	1	0	X	X	X	X	X	X	X	X	X	X
Auto Refresh	DS2	DS1	DS0	0			RS	X	X	X	X	1	0	1	X	X	X	X	X	X	X	X	X	X
Enter Self Refresh	DS2	DS1	DS0	0			RS	X	X	X	X	1	0	0	X	X	X	X	X	X	X	X	X	X
Exit Self Refresh / Exit Power Down	DS2	DS1	DS0	0			RS	X	X	X	X	0	1	1	X	X	X	X	X	X	X	X	X	X
Enter Power Down	DS2	DS1	DS0	0			RS	X	X	X	X	0	1	0	X	X	X	X	X	X	X	X	X	X
reserved	X	X	X	X	X	X	X	X	X	X	X	0	0	X	X	X	X	X	X	X	X	X	X	X

TABLE 3-5 FBD Channel Commands

Channel Cmds	23	22	21	[20:14]	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Debug:: In-band Events	EV7	EV6	EV5	7'b0001111	1	EV4	EV3	EV2	EV1	EV0	PV7	PV6	PV5	PV4	PV3	PV2	PV1	PV0
Debug:: Relative Timing	PH5	PH4	PH3	7'b0001111	0	PH2	PH1	PH0	RT9	RT8	RT7	RT6	RT5	RT4	RT3	RT2	RT1	RT0
Debug:: Exposed Info	EX16	EX15	EX14	7'b0001110	EX13	EX12	EX11	EX10	EX9	EX8	EX7	EX6	EX5	EX4	EX3	EX2	EX1	EX0
reserved	X	X	X	7'b000110x	X	X	X	X	X	X	X	X	X	X	X	X	X	X
reserved				7'b000110xx	X	X	X	X	X	X	X	X	X	X	X	X	X	X

TABLE 3-5 FBD Channel Commands

Channel Cmds	23	22	21	[20:14]	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DRAM CKE per DIMM	DS2	DS1	DS0	7'b0000111	BC	X	X	X	X	X	DE7	DE6	DE5	DE4	DE3	DE2	DE1	DE0
DRAM CKE per RANK	DS2	DS1	DS0	7'b0000110	BC	X	X	X	X	X	D3R1	D3R0	D2R1	D2R0	D1R1	D1R0	D0R1	D0R0
Write Config Reg	DS2	DS1	DS0	7'b0000101	DS3	TID	X	A10	A9	A8	A7	A6	A5	A4	A3	A2	0	0
Read Config Reg	DS2	DS1	DS0	7'b0000100	DS3	X	X	A10	A9	A8	A7	A6	A5	A4	A3	A2	0	0
reserved	X	X	X	7'b0000011	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Soft Channel Reset	X	X	X	7'b0000010	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Sync	X	X	X	7'b0000001	X	SD1	SD0	X	X	X	X	IER	ERC	EL0s	X	X	R1	R0
Channel NOP	X	X	X	7'b0000000	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Refer to the FB-DIMM Architecture and Protocol and Advanced Memory Buffer specifications for a full explanation of the channel commands.

3.3.2.1 FBD Frame Formats

Data is transmitted across the southbound and northbound channels in frames. For the southbound channel, 10 bits of data are sent per cycle over 12 cycles. For the northbound channel 14 bits of data are sent per cycle over 12 cycles. The next two sections show the format of the frames.

Southbound Frame Formats

The southbound frame format consists of two sections, the "A" command section and the "B"/"C" command or Data section. The 24-bit "A" command section is contained in the first four cycles of the frame. The aC[23:0] bits in TABLE 3-6 shows the location of the "A" command. Bits F[1:0] determines the format of the last 8 cycles of the frame as shown in TABLE 3-7. Bits aE[13:0] are the CRC value protecting the aC[23:0] and F[1:0] fields.

The FE[21:0] bits in TABLE 3-6 are CRC bits protecting the 72 bits of command or data in the remaining 8 cycles. Bits FE[13:0] are exclusive-ORed with the aE[13:0] field of the following frame.

In failover mode, the data for bit 9 is neither transmitted nor used in CRC calculations.

TABLE 3-6 Common Features of Normal Southbound Frames

	Bit									
Transfer	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	aE0	aE7	aE8	F0	aC20	aC16	aC12	aC8	aC4	aC0
1	aE1	aE6	aE9	F1	aC21	aC17	aC13	aC9	aC5	aC1
2	aE2	aE5	aE10	aE13	aC22	aC18	aC14	aC10	aC6	aC2
3	aE3	aE4	aE11	aE12	aC23	aC19	aC15	aC11	aC7	aC3
4	FE21									
5	FE20									
6	FE19									
7	FE18									
8	FE17									
9	FE16									
10	FE15									
11	FE14									
	FE0	FE7	FE8							
	FE1	FE6	FE9							
	FE2	FE5	FE10	FE13						
	FE3	FE4	FE11	FE12						

TABLE 3-7 Southbound Frame Type Encoding

Frame Format	F1	F0	Comments
Command	0	0	Frame contains one or more commands plus optional data
reserved	0	1	
Command + Wdata	1	WSn	Frame contains an "A" command plus 72 bits of Wdata

Command Frame Format

TABLE 3-8 shows the format of a southbound frame with three commands, aC[23:0], bC[23:0], and cC[23:0].

TABLE 3-8 Command Frame Format

Transfer	Bit									
	9	8	7	6	5	4	3	2	1	0
0	aE0	aE7	aE8	F0	aC20	aC16	aC12	aC8	aC4	aC0
1	aE1	aE6	aE9	F1	aC21	aC17	aC13	aC9	aC5	aC1
2	aE2	aE5	aE10	aE13	aC22	aC18	aC14	aC10	aC6	aC2
3	aE3	aE4	aE11	aE12	aC23	aC19	aC15	aC11	aC7	aC3
4	FE21	0	0	0	bC20	bC16	bC12	bC8	bC4	bC0
5	FE20	0	0	0	bC21	bC17	bC13	bC9	bC5	bC1
6	FE19	0	0	0	bC22	bC18	bC14	bC10	bC6	bC2
7	FE18	0	0	0	bC23	bC19	bC15	bC11	bC7	bC3
8	FE17	0	0	0	cC20	cC16	cC12	cC8	cC4	cC0
9	FE16	0	0	0	cC21	cC17	cC13	cC9	cC5	cC1
10	FE15	0	0	0	cC22	cC18	cC14	cC10	cC6	cC2
11	FE14	0	0	0	cC23	cC19	cC15	cC11	cC7	cC3

Command Frame with Data Format

TABLE 3-9 shows the southbound frame format where the "B" command has a 32-bit data payload. The BE[3:0] bits are byte enables for the data. This format is used to write to internal control registers of the AMB.

TABLE 3-9 Command Frame with Data Format

Transfer	Bit									
	9	8	7	6	5	4	3	2	1	0
0	aE0	aE7	aE8	F0	aC20	aC16	aC12	aC8	aC4	aC0
1	aE1	aE6	aE9	F1	aC21	aC17	aC13	aC9	aC5	aC1
2	aE2	aE5	aE10	aE13	aC22	aC18	aC14	aC10	aC6	aC2
3	aE3	aE4	aE11	aE12	aC23	aC19	aC15	aC11	aC7	aC3
4	FE21	0	0	0	bC20	bC16	bC12	bC8	bC4	bC0

TABLE 3-9 Command Frame with Data Format *(Continued)*

Transfer	Bit									
5	FE20	0	0	0	bC21	bC17	bC13	bC9	bC5	bC1
6	FE19	0	0	0	bC22	bC18	bC14	bC10	bC6	bC2
7	FE18	0	0	0	bC23	bC19	bC15	bC11	bC7	bC3
8	FE17	BE0	D28	D24	D20	D16	D12	D8	D4	D0
9	FE16	BE1	D29	D25	D21	D17	D13	D9	D5	D1
10	FE15	BE2	D30	D26	D22	D18	D14	D10	D6	D2
11	FE14	BE3	D31	D27	D23	D19	D15	D11	D7	D3

Command +WData Frame Format (4-bit Device)

Write data is sent in the southbound frame when F[1] is 1. Write data for a write command is sent to an AMB over four cycles. TABLE 3-10 shows the format of the F[1:0] field for these four frames. The WS[2:0] field identifies the target AMB. Each AMB must speculatively store the write data in an accumulation buffer until it determines that the data is for it. If the data is for that AMB, the data is stored in the Write Data Buffer; otherwise, it is discarded.

TABLE 3-10 WData Address Delivery

Wdata Frame	F1	F0
0	1	WS0
1	1	WS1
2	1	WS2
3	1	0

TABLE 3-11 shows the format of the Command with Wdata frame.

TABLE 3-11 Command+Wdata Frame Format (4-bit Device)

Transfer	Bit									
	9	8	7	6	5	4	3	2	1	0
0	aE0	aE7	aE8	F0	aC20	aC16	aC12	aC8	aC4	aC0
1	aE1	aE6	aE9	F1	aC21	aC17	aC13	aC9	aC5	aC1
2	aE2	aE5	aE10	aE13	aC22	aC18	aC14	aC10	aC6	aC2

TABLE 3-11 Command+Wdata Frame Format (4-bit Device) *(Continued)*

Transfer	Bit									
3	aE3	aE4	aE11	aE12	aC23	aC19	aC15	aC11	aC7	aC3
4	FE21	C17D0	C15D0	C13D0	C11D0	C9D0	C7D0	C5D0	C3D0	C1D0
5	FE20	C17D1	C15D1	C13D1	C11D1	C9D1	C7D1	C5D1	C3D1	C1D1
6	FE19	C17D2	C15D2	C13D2	C11D2	C9D2	C7D2	C5D2	C3D2	C1D2
7	FE18	C17D3	C15D3	C13D3	C11D3	C9D3	C7D3	C5D3	C3D3	C1D3
8	FE17	C18D0	C16D0	C14D0	C12D0	C10D0	C8D0	C6D0	C4D0	C2D0
9	FE16	C18D1	C16D1	C14D1	C12D1	C10D1	C8D1	C6D1	C4D1	C2D1
10	FE15	C18D2	C16D2	C14D2	C12D2	C10D2	C8D2	C6D2	C4D2	C2D2
11	FE14	C18D3	C16D3	C14D3	C12D3	C10D3	C8D3	C6D3	C4D3	C2D3

Northbound Frame Formats

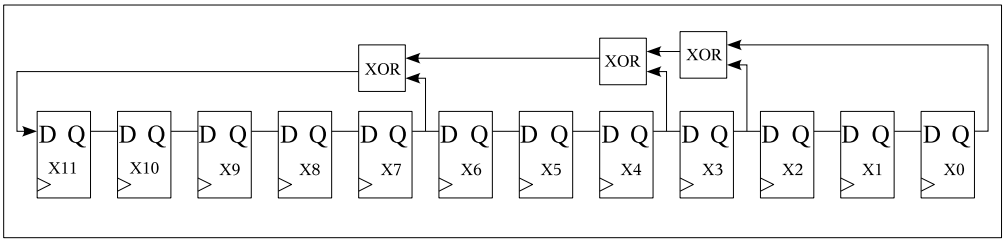
Northbound frames return status information or read data. Alert Frames implicitly tell if an error has occurred on the southbound channel. A Status Frame is an explicit status response to a Sync Frame on the southbound channel. When no Status or Read Data Frame is expected and no Alert Frame is detected on the northbound channel, then an Idle frame is expected.

Bit 13 of the northbound frame is not transmitted or used in CRC calculations when in failover mode.

Northbound Idle Frame Format

The Idle Frames are sent by the AMBs on the northbound channel to indicate that it is still operating correctly. The bits within the frame are determined by a 12-bit LFSR with polynomial x^12 + x^7 + x^4 + x^3 + 1. An example implementation is shown in FIGURE 3-4.

FIGURE 3-4 Idle Frame LFSR Counter



The initial value is 12'b000000000001, and the LFSR cycles through 2^12 - 1 states before repeating. The pattern in the LFSR is mapped to the Idle Frame bit lanes, i.e. X0 maps to bit lane 0, X1 to bit lane 1, etc. The 13th bit lane contains the value of X0 for the first 6 cycles of the frame and the inverse of X0 for the last 6 cycles. The 14th bit lane contains the value of X0 for all 12 cycles of the frame. TABLE 3-13 shows the format of the first Idle Frame.

TABLE 3-12 First Northbound Idle Frame Format

Xfer	Bit													
	13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	1	0	0	0	0	0	0	0	0	0	0	0	1
1	1	1	0	0	0	0	0	0	0	0	0	0	0	1
2	1	1	0	0	0	0	0	0	0	0	0	0	0	1
3	1	1	0	0	0	0	0	0	0	0	0	0	0	1
4	1	1	0	0	0	0	0	0	0	0	0	0	0	1
5	1	1	0	0	0	0	0	0	0	0	0	0	0	1
6	1	0	0	0	0	0	0	0	0	0	0	0	0	1
7	1	0	0	0	0	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	0	0	0	0	1
9	1	0	0	0	0	0	0	0	0	0	0	0	0	1
10	1	0	0	0	0	0	0	0	0	0	0	0	0	1
11	1	0	0	0	0	0	0	0	0	0	0	0	0	1

Alert Frame Format

An AMB will begin sending Alert Frames in place of Idle Frames on the northbound channel whenever an error has been detected on the southbound channel and will continue to do so until it receives a Soft Channel Reset command or a channel reset. The Alert Frame format is the inverse of the corresponding Idle Frame. TABLE 3-13 shows the format of the Alert Frame that replaces the Second Idle Frame.

TABLE 3-13 Alert Frame Replacing First Idle Frame

Xfer	Bit													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	0
1	0	0	1	1	1	1	1	1	1	1	1	1	1	0
2	0	0	1	1	1	1	1	1	1	1	1	1	1	0
3	0	0	1	1	1	1	1	1	1	1	1	1	1	0
4	0	0	1	1	1	1	1	1	1	1	1	1	1	0
5	0	0	1	1	1	1	1	1	1	1	1	1	1	0
6	0	1	1	1	1	1	1	1	1	1	1	1	1	0
7	0	1	1	1	1	1	1	1	1	1	1	1	1	0
8	0	1	1	1	1	1	1	1	1	1	1	1	1	0
9	0	1	1	1	1	1	1	1	1	1	1	1	1	0
10	0	1	1	1	1	1	1	1	1	1	1	1	1	0
11	0	1	1	1	1	1	1	1	1	1	1	1	1	0

Data Frame Format

TABLE 3-14 shows the format of a Northbound Read Data Frame for x4 devices. It contains two 72-bit data packets, each with 12-bit CRC protection.

TABLE 3-14 Northbound Data Frame Format

Xfer	Bit													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	E1.0	E1.11	C17.D2	C16D0	C14.D2	C13.D0	C11.D2	C10.D0	C8.D2	C7.D0	C5.D2	C4.D0	C2.D2	C1.D0
1	E1.1	E1.10	C17.D3	C16D1	C14.D3	C13.D1	C11.D3	C10.D1	C8.D3	C7.D1	C5.D3	C4.D1	C2.D3	C1.D1

TABLE 3-14 Northbound Data Frame Format *(Continued)*

Xfer	Bit													
2	E1.2	E1.9	C18.D0	C16D2	C15.D0	C13.D2	C12.D0	C10.D2	C9.D0	C7.D2	C6.D0	C4.D2	C3.D0	C1.D2
3	E1.3	E1.8	C18.D1	C16D3	C15.D1	C13.D3	C12.D1	C10.D3	C9.D1	C7.D3	C6.D1	C4.D3	C3.D1	C1.D3
4	E1.4	E1.7	C18.20	C17.D0	C15.D2	C14.D0	C12.D2	C11.D0	C9.D2	C8.D0	C6.D2	C5.D0	C3.D2	C2.D1
5	E1.5	E1.6	C18.D3	C17.D1	C15.D3	C14.D1	C12.D3	C11.D1	C9.D3	C8.D1	C6.D3	C5.D1	C3.D3	C2.D0
6	E2.0	E2.11	C17.D2	C16.D0	C14.D2	C13.D0	C11.D2	C10.D0	C8.D2	C7.D0	C5.D2	C4.D0	C2.D2	C1.D0
7	E2.1	E2.10	C17.D3	C16.D1	C14.D3	C13.D1	C11.D3	C10.D1	C8.D3	C7.D1	C5.D3	C4.D1	C2.D3	C1.D1
8	E2.2	E2.0	C18.D0	C16.D2	C15.D0	C13.D2	C10.D0	C10.D2	C9.D0	C7.D2	C6.D0	C4.D2	C3.D0	C1.D2
9	E2.3	E2.8	C18.D1	C16.D3	C15.D1	C13.D3	C10.D1	C10.D3	C9.D1	C7.D3	C6.D1	C4.D3	C3.D1	C1.D3
10	E2.4	E2.7	C18.D2	C17.D0	C15.D2	C14.D0	C12.D2	C11.D0	C9.D2	C8.D0	C6.D2	C5.D0	C3.D2	C2.D0
11	E2.5	E2.6	C18.D3	C17.D1	C15.D3	C14.D1	C12.D3	C11.D1	C9.D3	C8.D1	C6.D3	C5.D1	C3.D3	C2.D1

Northbound Register Data Frame Format

TABLE 3-15 shows the format for an AMB Register Read Data Frame.

TABLE 3-15 Northbound Register Data Frame Format

Xfer	Bit													
	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	E1.0	E1.11	0	0	0	0	0	0	D30	D24	D18	D12	D6	D0
1	E1.1	E1.10	0	0	0	0	0	0	D31	D25	D19	D13	D7	D1
2	E1.2	E1.9	0	0	0	0	0	0	0	D26	D20	D14	D8	D2
3	E1.3	E1.8	0	0	0	0	0	0	0	D27	D21	D15	D9	D3
4	E1.4	E1.7	0	0	0	0	0	0	0	D28	D22	D16	D10	D4
5	E1.5	E1.6	0	0	0	0	0	0	0	D29	D23	D17	D11	D5
6	E2.0	B2.11	0	0	0	0	0	0	0	0	0	0	0	0
7	E2.1	E2.10	0	0	0	0	0	0	0	0	0	0	0	0
8	E2.2	E2.9	0	0	0	0	0	0	0	0	0	0	0	0
9	E2.3	E2.8	0	0	0	0	0	0	0	0	0	0	0	0
10	E2.4	E2.7	0	0	0	0	0	0	0	0	0	0	0	0
11	E2.5	E2.6	0	0	0	0	0	0	0	0	0	0	0	0

Northbound Status Frame Format

TABLE 3-16 shows the format of the northbound Status Frame. The Status Frame is sent in response to Sync Frame on the southbound channel. The Status Frame returns at the time that the first Read Data Frame would return if a read were issued at the time of Sync Frame plus an additional delay determined by the SD[1:0] field of the Sync command. Each AMB sends its status back on the bit lane corresponding to its AMB_ID. The S[3:0] bits are the status information from the configuration register selected in the Sync command (TABLE 3-17). The SP bit is the odd parity of S[3:0].

TABLE 3-16 Status Frame Format

Xfer	Bit													
	13	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	DBS0	DAS0	D9S0	D8S0	D7S0	D6S0	D5S0	D4S0	D3S0	D2S0	D1S0	D0S0
1	0	0	DBS1	DAS1	D9S1	D8S1	D7S1	D6S1	D5S1	D4S1	D3S1	D2S1	D1S1	D0S1
2	0	0	DBS2	DAS2	D9S2	D8S2	D7S2	D6S2	D5S2	D4S2	D3S2	D2S2	D1S2	D0S2
3	0	0	DBS3	DAS3	D9S3	D8S3	D7S3	D6S3	D5S3	D4S3	D3S3	D2S3	D1S3	D0S3
4	0	0	DBSP	DASP	D9SP	D8SP	D7SP	D6SP	D5SP	D4SP	D3SP	D2SP	D1SP	D0SP
5	0	1	0	1	0	1	0	1	0	1	0	1	0	1
6	1	0	1	0	1	0	1	0	1	0	1	0	1	0
7	0	1	0	1	0	1	0	1	0	1	0	1	0	1
8	1	0	1	0	1	0	1	0	1	0	1	0	1	0
9	0	1	0	1	0	1	0	1	0	1	0	1	0	1
10	1	0	1	0	1	0	1	0	1	0	1	0	1	0
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1

TABLE 3-17 Status Bit Description

Field	Name	Description
FBD Status 0		
SP	Parity	Parity of S[3:0]
S3	NBDE	Northbound Debug Event
S2:S1	Thermal_Trip	AMB thermal information for thermal management
S0	Alert_Asserted	An error has been detected by the AMB.

TABLE 3-17 Status Bit Description *(Continued)*

Field	Name	Description
FBD Status 1		
SP	Parity	Parity of S[3:0]
S[3:1]	reserved	reserved
S[0]	Data_Merge_Error	AMB cannot meet northbound data merge timing requirement.
FBD Status 2		
SP	Parity	Parity of S[3:0]
S[3:0]	reserved	reserved
FBD Status 3		
SP	Parity	Parity of S[3:0]
S[3:0]	reserved	reserved

Refer to the FB-DIMM Architecture and Protocol and Advanced Memory Buffer specifications for a full explanation of the frame formats and AMB register definitions.

3.3.3 SDRAM Initialization

The initialization sequence within the MCU for the SDRAMs will still follow the same flow; however, the interface will be different. The MCU will have to initialize the SDRAMs indirectly through registers in the AMBs. The MCU will issue a command to the AMBs and then poll status registers to determine when the AMBs have completed issuing the command to the SDRAMs.

After SDRAM initialization is complete, the MCU will begin scheduling commands directly to the SDRAMs.

The DDR2 SDRAMs must be powered up and initialized in a predefined manner. Operational procedures other than those specified may result in undefined operation. TABLE 3-18 shows the sequence of steps required for POWER UP and Initialization.

TABLE 3-18 SDRAM Power Up and Initialization Sequence

Step	Required Action
1.	Apply power to VDD.
2.	Apply power to VDDQ.
3.	Apply power to VREF and to the system VTT.

TABLE 3-18 SDRAM Power Up and Initialization Sequence *(Continued)*

Step	Required Action
4.	Start clock and maintain stable condition for 200 s.
5.	Apply No Operation or Deselect command and take CKE high.
6.	Wait minimum of 400ns, then issue a Precharge-all command.
7.	Issue Extended Mode Register 2 Set (EMRS(2)) command.
8.	Issue Extended Mode Register 3 Set (EMRS(3)) command.
9.	Issue Extended Mode Register 1 Set (EMRS(1)) command to enable DLL.
10.	Issue Mode Register Set (MRS) command to reset DLL.
11.	Issue Precharge-all command.
12.	Issue 2 or more Auto-Refresh commands.
13.	Issue MRS command with low on A8 to initialize device operation (i.e. to program operating parameters without resetting the DLL).
14.	At least 200 clocks after step 8, execute OCD Calibration (Off Chip Driver Impedance adjustment). If OCD calibration is not used, EMRS OCD Default command (A9=A8=A7=1) followed by EMRS OCD Calibration Mode Exit command (A9=A8=A7=0) must be issued with other parameters of EMRS.
15.	The DDR2 SDRAM is now ready for normal operation.

3.3.4 DDR2 SDRAM Commands

TABLE 3-19 shows the truth table for the commands supported by the DDR2 SDRAMs.

TABLE 3-19 DDR2 SDRAM Command Truth Table

Function	CKE Previous Cycle	CKE Current Cycle	CS#	RAS#	CAS#	WE#	Bank	Address
Mode/Extended Mode Register Set	H	H	L	L	L	L	BA	Op-Code
Auto-Refresh	H	H	L	L	L	H	X	X
Self-Refresh Entry	H	L	L	L	L	H	X	X
Self-Refresh Exit	L	H	H	X	X	X	X	X
			L	H	H	H		
Single Bank Precharge	H	H	L	L	H	L	BA	A10=L
Precharge All Banks	H	H	L	L	H	L	X	A10=H

TABLE 3-19 DDR2 SDRAM Command Truth Table *(Continued)*

Function	CKE Previous Cycle	CKE Current Cycle	CS#	RAS#	CAS#	WE#	Bank	Address
Bank Activate	H	H	L	L	H	H	BA	Row Address
Write								Column Address A10=L
Write with Auto-Precharge								Column Address A10=H
Read								Column Address A10=L
Read with Auto-Precharge	H	H	L	H	L	H	BA	Column Address A10=H
No Operation	H	X	L	H	H	H	X	X
Device Deselect	H	X	H	X	X	X	X	X
Power Down Entry	H	L	H	X	X	X	X	X
			L	H	H	H		
Power Down Exit	L	H	H	X	X	X	X	X
			L	H	H	H		

Commands Supported by OpenSPARC T2

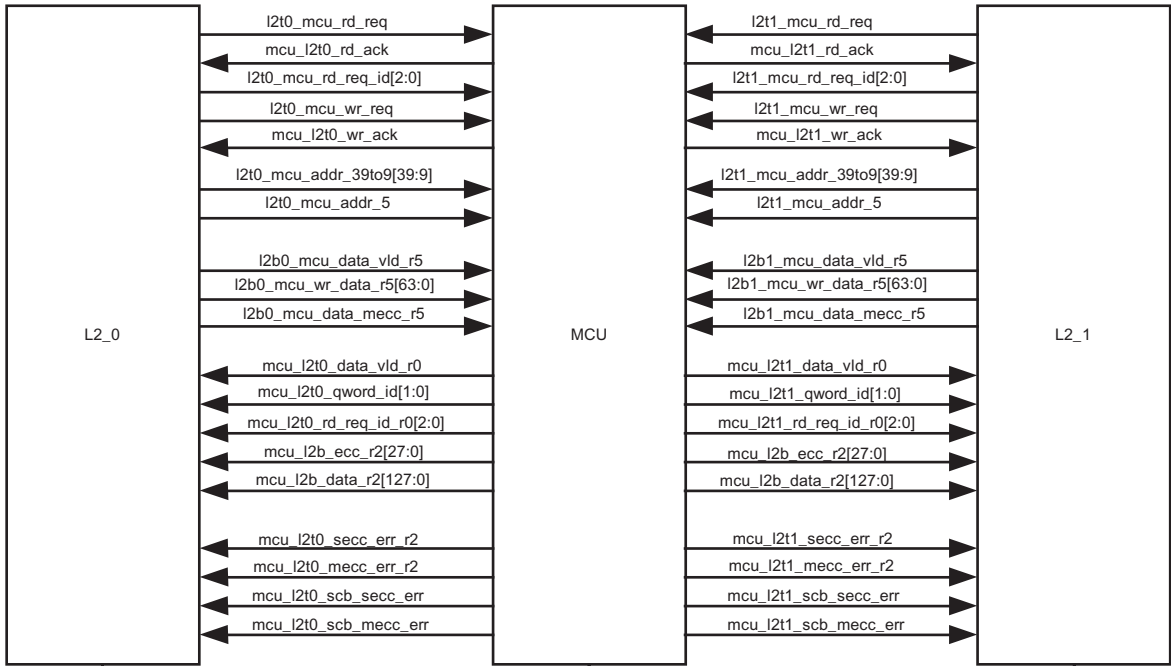
- Mode/Extended Mode Register Set - program Mode or Extended Mode Register which controls operation of SDRAM.
- Auto-Refresh - refresh all SDRAM banks.
- Self-Refresh - place SDRAM in refresh mode controlled by a timer within the SDRAM.
- Power Down - place SDRAM in low power mode.
- Single Bank Precharge - deactivate row in a particular bank.
- Precharge All Banks - deactivate rows in all banks.
- Bank Activate - activate a row within a particular bank.

- Write with Auto-Precharge - perform a write operation and deactivate the bank after completion.
- Read with Auto-Precharge - perform a read operation and deactivate the bank after completion.
- No Operation/Device Deselect - no operation.
- (Read and Write without Autoprecharge are not supported in OpenSPARC T2 since a bank is always closed after a transaction.)

3.4 MCU-L2 Cache Interface

An L2 cache bank can send one read or write request at a time to the MCU. Once it sends a request it must wait for the appropriate acknowledge before sending the next request. There is a delay of three cycles from the completion of a transaction (acknowledge for a read, last data word for a write) until the next request can be made. There can be a total of eight outstanding read requests and eight outstanding write requests from each L2 cache bank at any time.

FIGURE 3-5 MCU-L2 Cache Interface Signals



3.4.1 MCU Read Transaction

The L2 Cache makes a read request by asserting l2t_mcu_rd_req to the MCU at the same time that l2t_mcu_addr[39:4] and l2t_mcu_rd_req_id[2:0], the index into the L2 cache's fill buffer for returning data, are valid. The request is registered in the MCU and held in the l2clk domain until l2if_cmp_ddr_sync_en is high. (l2if_cmp_ddr_sync_en is a registered version of cmp_ddr_sync_en , the top-level clock synchronization signal.) This signal is used to synchronize the Signals crossing from the l2clk (1.4 GHz) domain to the drl2clk (800 MHz) domain and indicates that the request will be stored in the Read Request Queue on the next drl2clk cycle. FIGURE 3-6 shows the timing of the read request and the internal acknowledge. There is a two-deep FIFO on the read request port. When a read request comes in, it is placed in the FIFO. An acknowledge for a transaction is sent to the L2 cache when that transaction reaches the head of the FIFO, either when a transaction is placed into an empty FIFO or when both entries are full and then an entry is dequeued.

No flow control is needed for the returning read data because the L2 cache will guarantee space to receive the data. When mcu_l2t_data_vld_r0 signal is asserted, the Signals mcu_l2t_qword_id_r0[1:0] and mcu_l2t_read_req_id[2:0] are driven at the same time, and mcu_l2b_data_r3[127:0] and mcu_l2b_ecc_r3[27:0] are driven three cycles later. Also, mcu_l2t_secc_err_r3 or mcu_l2t_mecc_err_r3 will be

asserted at the same time as the data if a correctable or uncorrectable error, respectively, occurred in the corresponding data beat. The data is returned to the L2 cache over several cycles because of the difference between l2clk and drl2clk. FIGURE 3-6 and FIGURE 3-7 show an example of a 6 to 1 l2clk to drl2clk ratio for a read request. The order in which the data beats are returned to the L2 cache depends on the bit PA[5], as discussed in section 3.3.

Since an L2 bank can only have 8 outstanding read requests at a time, and the MCU can handle eight outstanding reads per L2 bank, the MCU does not have to keep track of the number of outstanding reads.

Reads are not necessarily serviced in the order they are received from the L2 cache. Transactions are scheduled in order to limit the amount of dead data cycles on the bus to the DIMMs. Transaction scheduling is more fully discussed in section 10.2.3 under Arbitration.

FIGURE 3-6 Read Request Timing

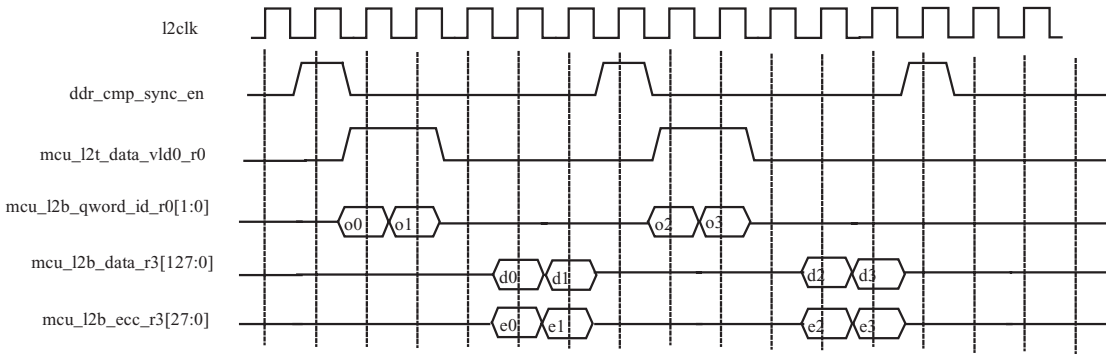
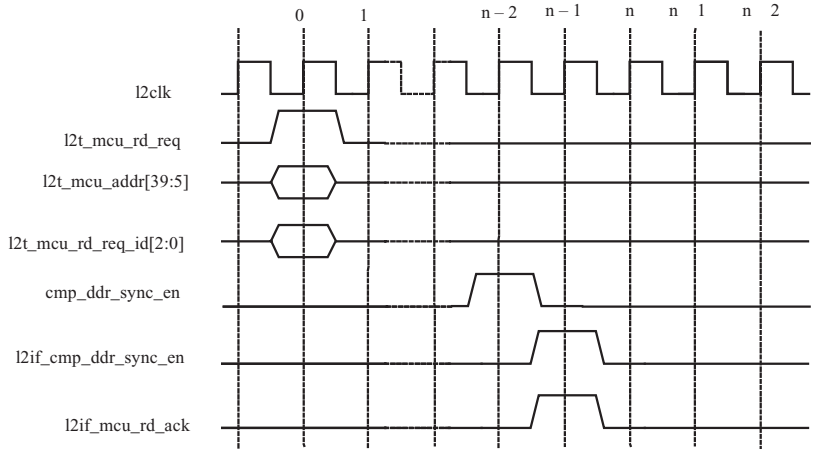


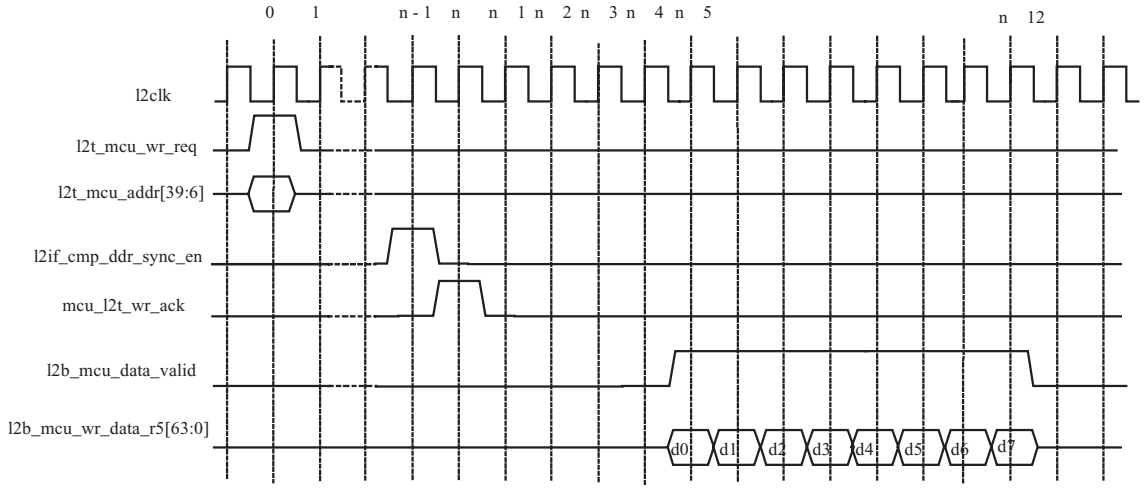
FIGURE 3-7 Read Data Return Timing



3.4.2 MCU Write Transaction

The L2 Cache makes a write request by asserting l2t_mcu_wr_req to the MCU at the same time that l2t_mcu_addr[39:6] is valid. Once the transaction is placed into the Write Request Queue, mcu_l2t_wr_ack is sent back to the L2 cache to indicate that it can now send another write transaction. l2b_mcu_data_valid and the write data l2b_mcu_wr_data_r5[63:0] are asserted 5 cycles after the acknowledge. The write data is sent to the MCU over 8 cycles for a total of 64 bytes of data.

FIGURE 3-8 Write Request Timing



3.5 DDR2 SDRAM Transaction Timing

3.5.1 Memory Read

The MCU reads data from the external memory by:

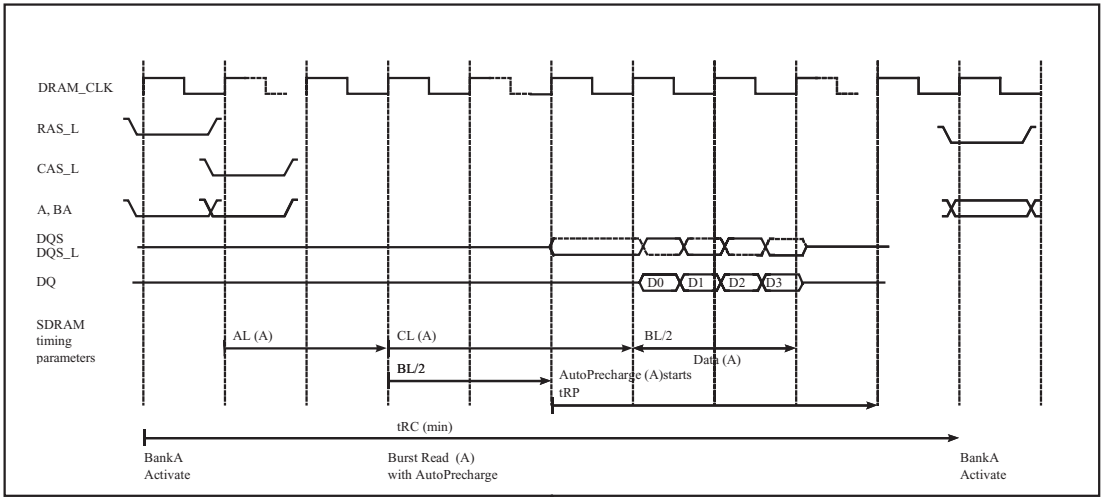
- issuing a bank activate by assert RAS and driving the row address and the bank select.
- issuing a Posted CAS Burst Read with AutoPrecharge command by asserting CAS and driving the column address.
- waiting for a delay of AL (Additive Latency) + CL (CAS latency) before sampling the read data.
- sampling data returning in a burst length of 4 in 2 drl2clk cycles.

A new Bank Activate command may be issued to the same bank if the following conditions are satisfied:

- The RAS precharge time (tRP) has been satisfied from the clock cycle at which the AutoPrecharge begins.
- The RAS cycle time (tRC) from the current Bank Activate has been satisfied.

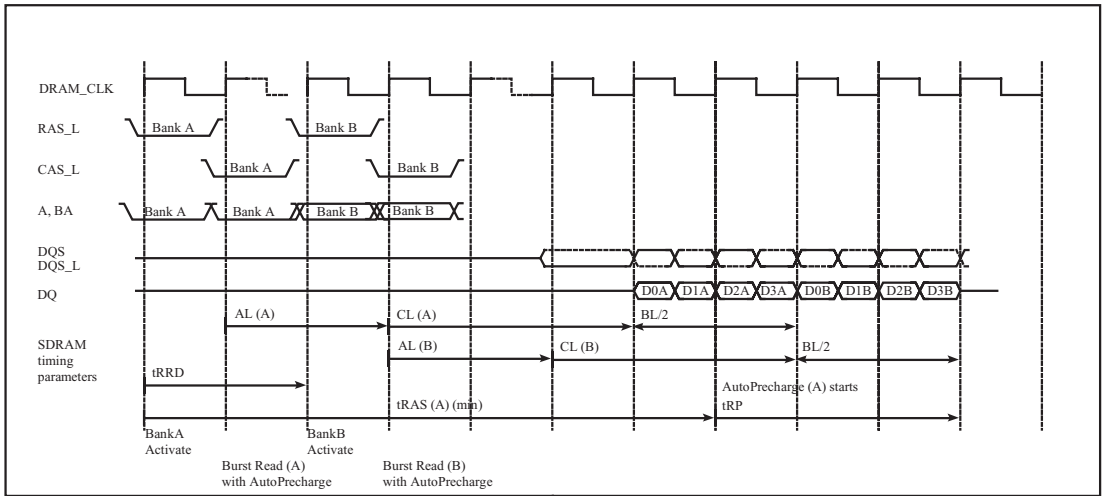
FIGURE 3-9 shows a single burst read of length 4 (BL=4) with AutoPrecharge, followed by a reactivation of the same bank.

FIGURE 3-9 Memory Burst Read with AutoPrechare, same bank reactivated



For seamless back-to-back memory reads, a different bank (B) can be activated during the memory read of bank (A). Bank (B) can start a burst read with AutoPrecharge command after a delay of BL/2 = 2cycles from the bank (A) burst read with AutoPrecharge command. FIGURE 3-11 shows a seamless burst read with AutoPrecharge command of 2 different banks.

FIGURE 3-10 Memory Burst Read with AutoPrecharge with multiple banks activated



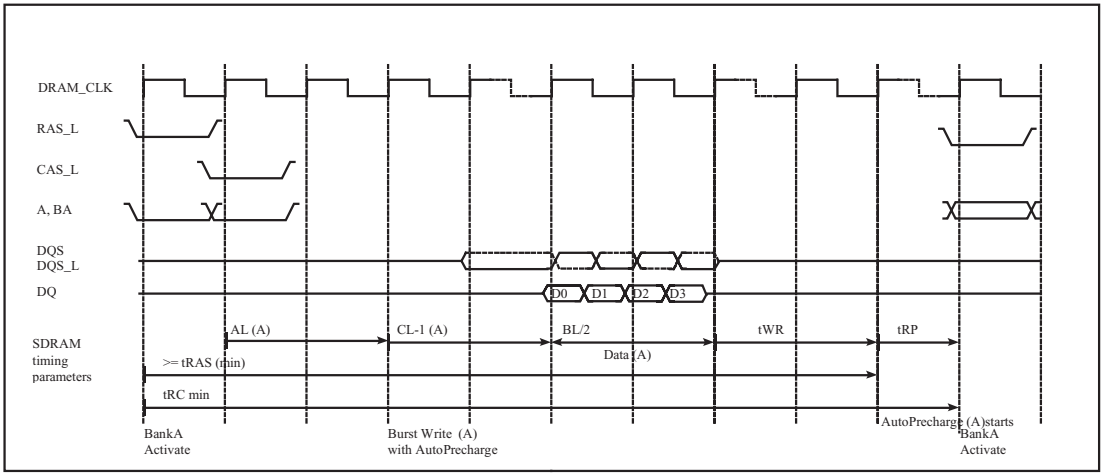
Note – There is an additional cycle delay when switching from one rank to a different rank.

3.5.2 Memory Write

The MCU writes data to the external memory by:

- issuing a bank activate by asserting RAS and the row address and the bank selects.
- issuing a Posted CAS Burst Write with AutoPrecharge command by asserting CAS and driving the column address .
- waiting for delay of (AL + CL-2) cycles from the CAS assertion, deasserting the datastrobe (DQS/DQS_L) for 1 cycle.
- waiting for a delay of (AL + CL-1) cycles from the CAS cycle before driving the datastrobe and the write data.

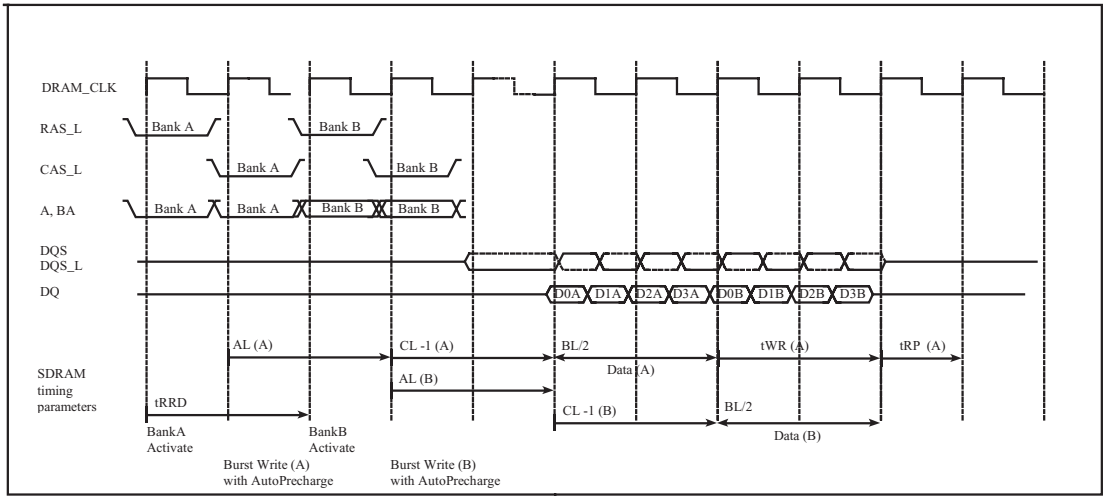
FIGURE 3-11 Memory Burst Write with AutoPrecharge and same bank activate



The same bank can be reactivated after a total delay of (1 + AL + CL-1 + BL/2 + tWR + tRP) cycles for reactivating the same bank. The following diagram shows a single burst write with AutoPrecharge command.

For a seamless memory write, a different bank (B) can be activated during the memory write of bank (A). The bank (B) burst write with AutoPrecharge command after a delay of BL/2 from the bank (A) burst write with AutoPrecharge command. The following diagram shows a seamless burst write with AutoPrecharge command of 2 different banks.

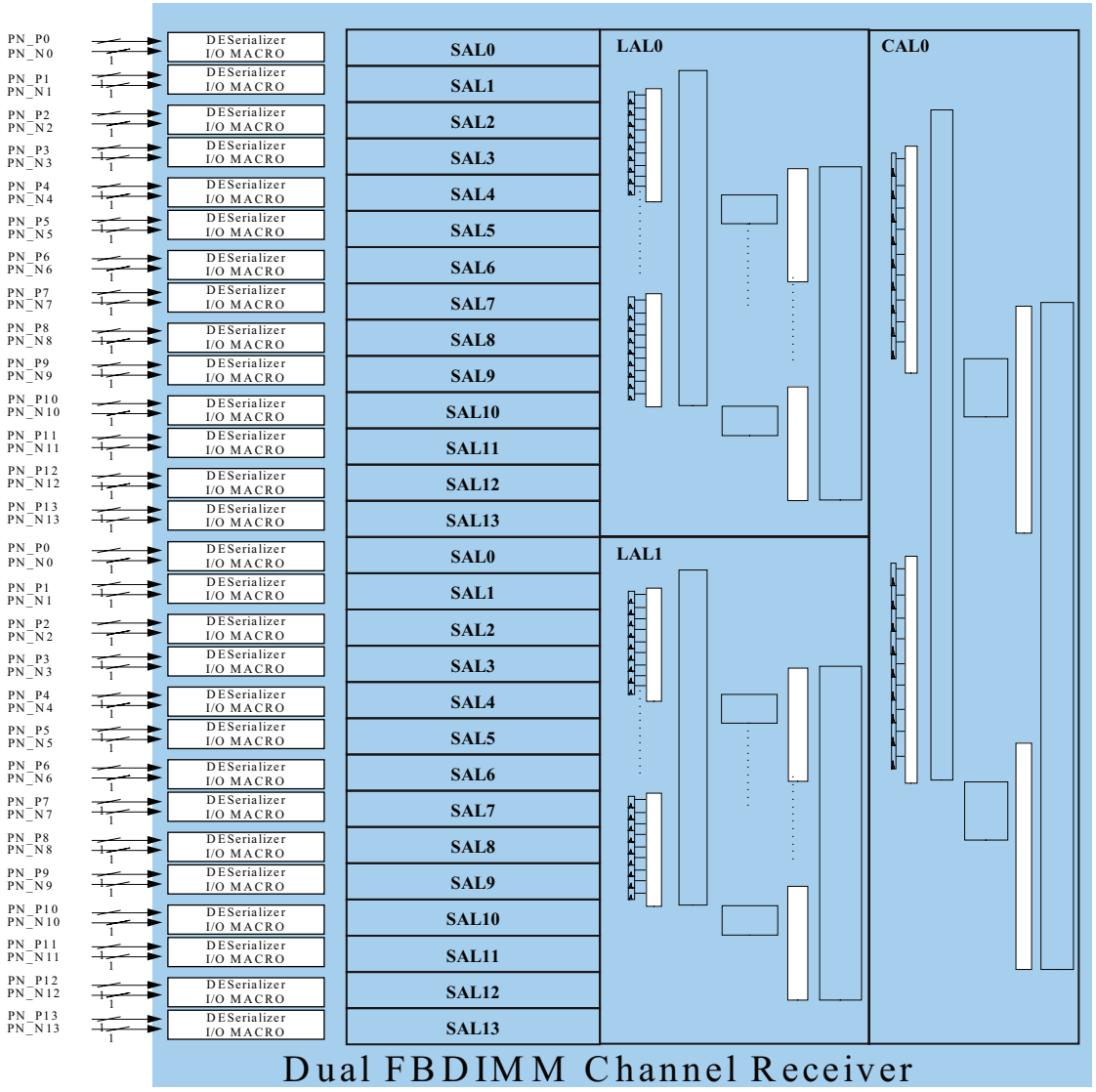
FIGURE 3-12 Memory Burst Write with AutoPrecharge and multiple banks activated



3.5.3 SERDES (I/O) Timing

The SERDES handles the physical layer of the FBD channel. The packets from MCU are converted into frames of bits sent out on the serial link. On the northbound serial lanes, the data returns from the memory link. Each frame going southbound is divided into 10 bit lanes and 12 bit times or Unit Interval (UI). The maximum bit lane frequency is 4.8GHz. Each frame going northbound is divided into 14 bit lanes and 12 bit times. The DRAM clock is 1/12 the link's frequency.

FIGURE 3-13 Dual FBDIMM Channel Receiver



On the receive side, each SERDES macro recover its high speed clock, convert the electrical Signals into a bit stream of logical 1s and 0s, and outputs a group (1 symbol) of 12 bits every 400MHz max. There are 3 fundamental microarchitectural issues for the host receiver:

- frame / symbol alignment logic (SAL) within a single bit lane
- frame / lane alignment logic (LAL) across all 14 northbound lanes needed so MCU data layer does not see the physical aspects of a single channel

- frame / channel alignment logic (CAL) across 2 northbound channels (across potentially all 28 northbound lanes) needed so MCU data layer does not see the physical aspects of 2 independent channels but sees instead dual channels operating logically in lockstep.

To minimize the number of wires between MCU and the SERDES logic, especially for supporting the dual channel case, half a frame is transferred every MCU's drl2clk clock cycle, which implies that drl2clk frequency must be at least 1/6 rather than 1/12 the link frequency.

3.5.3.1 Single Lane Symbol Alignment Logic

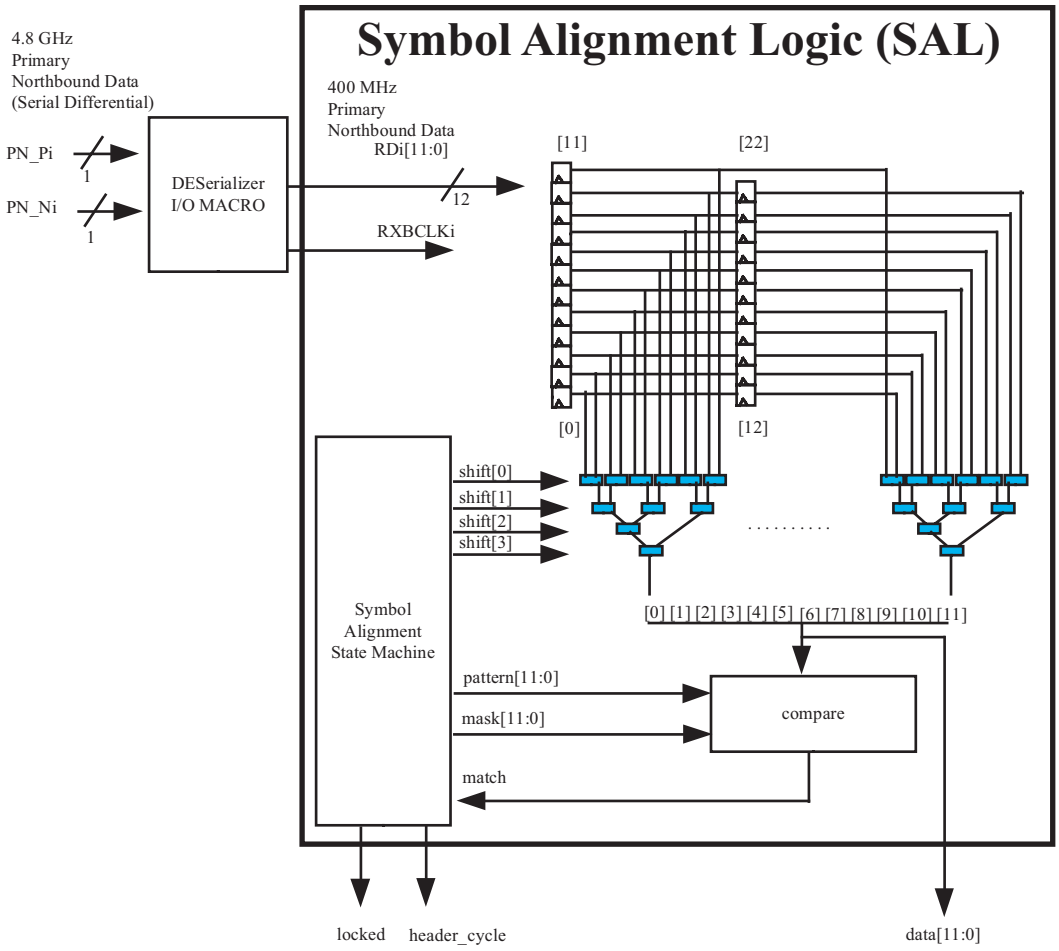
Assumes 1 SERDES macro handles 1 lane.

- Q1 : Will 1 fbd SERDES macro do "symbol" detection and lock during TS0 state -- meaning I'm expecting it to lock to the pattern (bus order lsb->msb) 0111_1111_1101 0101_0x0x_0x0x 0101_0101_0101 ....
- Q1a : If not, we would like the ability to have access to their bit (UI) alignment jog feature for 1,2,4,6,8 (or even 10) bits so we get symbol lock. (That feature is available in TI's PCI Express SERDES macro when its not programmed to do comma symbol detection.)
- Q2: how long for symbol lock? (Host only get a budget of 13 TS0 patterns from the northernmost AMB to recover clock, symbol lock and lane-deskew)

If we can't get the macro to do symbol lock, we'll need a symbol alignment logic (SAL) block. Basically, accumulate 2 12 bit symbols into flops and then program up a set of mux selects to hunt for the 0101_0101 0111_1111 1101_0101 pattern with a counter to make sure we can find it again exactly 1 TS0 pattern later. As a host we only have 13 TS0 patterns to get this and lane-deskew correct, so if we see the symbol again 3 times, the SAL block asserts symbol lock. This set of muxes + comparitors and state machine should be doable in 400 MHz. This SAL block is instantiated 14 times, 1 per lane.

FIGURE 3-14 shows the SAL block needed if the macro doesn't do symbol lock. The bit order for the 12 bit data is bit position 0 corresponds to oldest received bit on the serial link.

FIGURE 3-14 Symbol Alignment Logic

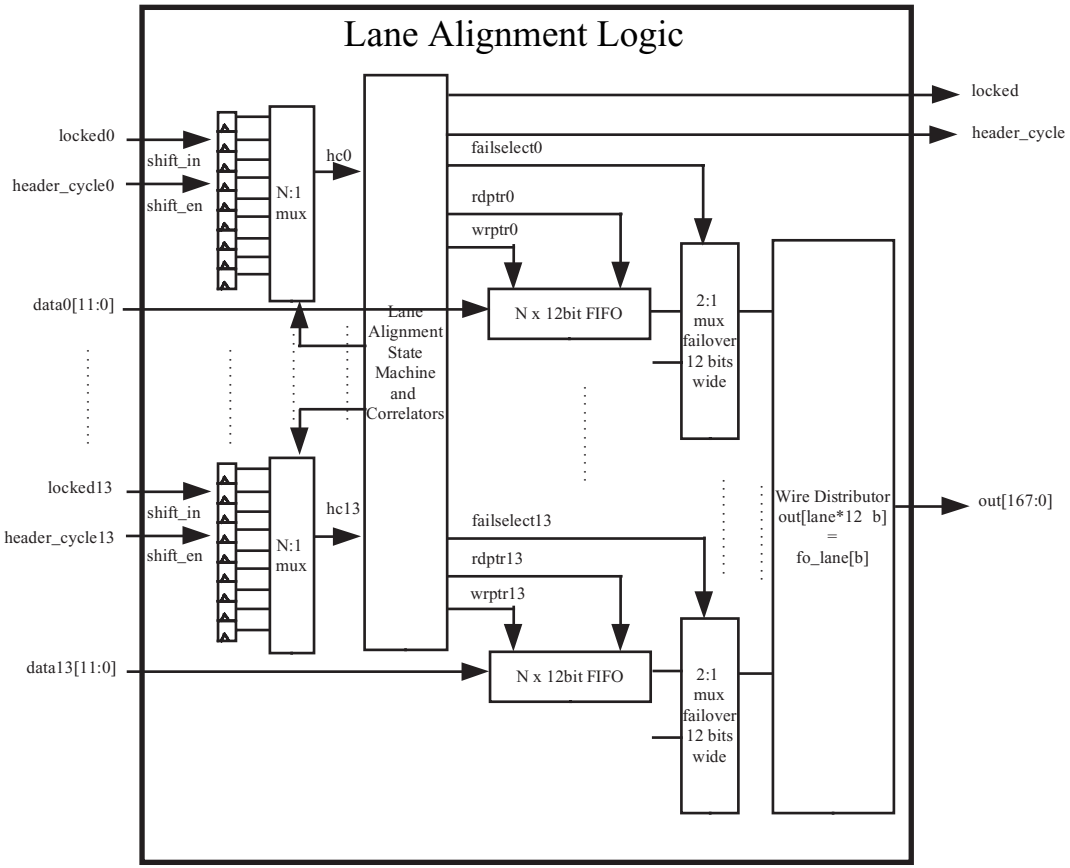


3.5.3.2 Frame Lane Alignment Logic across all 14 Northbound Lanes

Each bit lane has a static skew relative to another bit lane due to differences in drivers, receivers and traces between bit lanes. The FBDIMM link specification requires only the host perform the deskew on the northbound direction. Each AMB is required to deskew in the southbound direction.

Even when an AMB sends out the same symbol on all of the northbound lanes all initially aligned, a symbol from one SAL instance is not guaranteed to arrive at the same cycle as the same symbol from another SAL instance. A full frame must be collected and all 14 northbound lanes must be aligned to meaningfully interpret the original content. This lane alignment is done during link training. During training state 0 (TS0), identical frames are repeated many times and all lanes carry the same sequence of symbols. The header cycle of the frame is a uniquely identifiable symbol compared to the rest of the symbols in the frame. During TS0, the Lane Alignment Logic (LAL) queues up the symbols from each lane and search for the header cycle symbol. Once the skew between lanes have been determined consistently across multiple TS0 patterns, the LAL locks onto the wavefront and begins outputing frame aligned data for the full channel.

FIGURE 3-15 Lane Alignment Logic



Algorithm

```
lal_locked =0, lal_hc = 0, out=0, wrptr[0]=wrptr[1]=wrptr[2]=...=
wrptr[13] = 0, fastest_found = 14, lockcount=0, master_offset_counter
= 0; master_TS0pattern_counter = 0;
foreach cycle {
while (TS0state)  {
    if (lockcount<13) {
    foreach lane (i=0 to i=13) {
        if sal_locked[lane] {
           if sal_hc[lane] {
              if !fastest_found() {
                 /* assume lane is fastest */
                 /* set its rdptr to the deepest entry of the symbol
fifo */
                 rdptr[lane] = N;
                 set_lockseen[lane] = 1;
                 start_master_offset_counter(); /* counter
increments each cycle */
                 start_master_TS0pattern_counter(); /* counter
increments every TS0 pattern */
                 set_fastest_found(lane);
                 }
               else if !relationship_exists_to_fastest() {
                 if (master_offset_counter_value()  > N) /* something
is wrong... */
                     error_handle(SKEW_TOO_LARGE);
                 } else {
                     rdptr[lane] = rdptr[fastest_lane]-
master_offset_counter_value();
                     increment_lock_count();
                 }
                else if !same_relationship_to_fastest() {
                     error_handle(DYNAMIC_SKEW_TO_FASTEST);
                 }
             } /* if sal_hc[lane]
        } else { /* sal_locked is false */
                 if (lockseen[lane]) {
                    error_handle(SYMBOL_LOCK_UNLOCKED);
                 }
         } /* foreach lane */
      } /* if lockcount < 13 */
      else {
                 zero_offset_to_slowest(); /* for all i {rdptr[i]=
rdptr[i]-rdptr[slowest]}
               }
} /* while TS0 state */

lal_locked = (lock_count>=13) && (master_TS0pattern_counter > 2);
```

```
lal_header_cycle = lal_locked && (symbol(fastest_found,
rdptr[fastest_found]) == TS0_HEADER);
lal_out[167:0] = concat( failsymbol(0, rdptr[0], rdptr[1]),
failsymbol(1, rdptr[1], rdptr[2]), ... , symbol13(1, rdptr[13]));
```

The Lane alignment state machine has direct control of the write pointers and read pointers of 14 FIFOs. The SAL's lock signal is the write and shift enable. The state machine tracks the procession of each SAL's header_cycle signal over many cycle to determine the relative delays between the lanes. The first lane to have its SAL header_cycle assert is declared the fastest lane. It's symbol read pointer is set to the deepest entry (N). Each time a header cycle is detected on a locked lane, its symbol read pointer is set to the appropriate distance 'earlier' from the fastest's symbol read pointer. Once 13 out of the 14 lanes (enough to support failover) have achieved lock, the read pointer for all the lanes are subtracted such that the slowest lane's symbol read pointer is 0, while all the other lane remains the same distance away. This is for latency purpose only. A separate counter makes sure lane lock is not declared until multiple TS0 patterns have been elapsed with all 13 or 14 lanes locked.

For efficiency, each SAL's header_cycle signal is accumulated using flops to allow for arbitrary access for correlation purpose. Until lane alignment lock has occurred, the 168 bit data out is undeterministic. The lane alignment state machine also handles masking out the failed lane during training state 3.

Architecturally Complete Implementation

Architecturally, FBDIMM allows for up to 8 DIMMs and 4 logic analyzers on a single FBDIMM channel. All intermediate DIMMs transmitter are allowed to introduce 100ps + 2UI of skew. The last (southernmost) DIMM is allowed to introduce a longer lane skew for its northbound driver (100ps + 3UI). All receiver skew component have the same identical maximum (6UI).

Max lane-skew is

= (# AMBs -1) x (L_txskew2 + L_rxskew) + (L_txskew1 + L_rxskew)

= (# AMBs - 1) x (100ps + 2UI + 6UI) + (100ps + 3UI + 6UI)

= 11 x (100ps + 8UI) + 100ps + 9UI

= 1200ps + 97UI

= Worse case UI max lane skew @ 4.8Gbps, 1UI=208ps

= ~6UI + 97UI = ~9 frames

= Worse case UI max lane skew @ 4.8Gbps, 1UI=208ps

= ~6UI + 97UI = ~9 frames

The TS0 pattern is 12 frames long, so with a 9 frame lane-skew, and 9 frame deep buffer, there would not be an alias problem of erroneously locking one lane's TS0 pattern to a prior or next TS0 pattern on another lane. IF ANY TS0 patterns were dropped on any lanes by the designated AMB while it lane-deskews and the host locks the northbound lanes prior to that AMB locking its southbound lanes, then the true maximum allowable skew would be 6 frames long to prevent the situation of the fastest lane aliasing into a slower lane. A more robust (but much more costly) mechanism for the host would use TS1's unique end delimiters for the n-3, n-2, n-1, and n frames to correct any alias problem. After symbol lock, a budget of 2 or 3 TS0 patterns should be sufficient to assert lane-locked. If lock is not acquired after the total of 13 TSO patterns, the AMB_ID returned will be incorrect and MCU will need to look at lane-locked signal. If not locked, then it'll need to transition back to EI and do TS0 again.

Nine frames of lane skew can tolerate approximately 22 to 34 ns of delay due to trace mismatch. At about 61.5 ps/cm of trace velocity, that's about a mismatch of 50+ cm.

Having a 9 deep x 12 bit wide FIFO per lane seems excessive for the type of board design and blade system OpenSPARC T2 would go into. FBDIMM architecturally allows for very inexpensive board design and board manufacturing costs - relaxed board routing rules between lanes using cheap FR4 dielectric material - while allowing for a long latency system topology with DIMMs plugged on riser card and long chains. Architecturally FBDIMM allows a system with a N2's MCU is connected to 12 inches of board traces to a PCI Express connector to a memory riser board to a FBDIMM Connector to a set of AMBs DIMMs and long wires for hooking up a logic analyzer AMB module.

PC Board/System Dependent OpenSPARC T2 Implementation

If a more realistic board design and system tolopology is available, the cost of the deep buffer and logic to quickly search exhaustively through theoretically 914 permutation of ordering could be significantly reduced. A typical blade server does not have that much cubic space of freedom nor expected to need to access more memory on another blade via the backplane.

A sample analysis from slides from Intel Spring'03 IDF FBDIMM and RawCard simulation results suggests a better upper bound for the lane trace mismatch.

The short answer is:

1st order approximation of frames needed

frames = roundup (N * (7 * D + L) / 40)

N = settling time (# of wave propagations) of a tx/rcr pair.

D = max trace mismatch between a pair of DIMMs, in cm

L = max trace mismatch between host and northernmost DIMM.

Intel's "8 DIMM Layout" slide has the following:

- 0.4" DIMM to DIMM,

- shortest lead-in 1.2"

  (southbound channel 0)

- Longest lead-in 7.3"

  (northbound channel 3)

Using this as a recommended layout, .4" is practically 1cm.

Assuming N2's board has at least a 6 layer so there can be a clean back current return path and sufficient shielding against radiation to the outside and edges of the PCB; Assuming these 'critical' Signals avoid high impedance areas (vias, gaps, and zones in ground planes) and takes the path of least inductance (like avoid going vertical between planes as much as possible) and also assuming 45 degree turns instead of 90 degree turns to avoid changing the 'w' parameter of the wave guide.

The FR4 board dielectric has a permittivity of 4.5 at 1MHz (number used for DIMM card), but is 3.4 for the GHz operation in the channel (using a higher channel impedance of 85 ohm)

Propagation delay calculation:

Assuming the simple stripline transmission line model where the

inner metal traces are sandwiched by the pair of ground or VCC planes,

separated by a FR4 dielectric (er = ~3.4), wave velocity

ignoring the higher order effects of the full RLGC,

v = c / sqrt (er)

= 300um/ps / sqrt (3.4)

= 0.5423 * 300 um/ps

= 1627 um/ps (or 61.5 ps/cm)

Relative to the UI of 208ps (4.8GHz), v ~= 30% of a UI

I'm also going to assume a very slow driver to minimize the spike in driver-current and ringing. FBDIMM does not explicitly specify a maximum slew rate but they're implied by the edge transition rate and eye diagrams. So setting N to be a simple 1.

1 * .3 UI / cm = 0.3 UI skew per cm of trace mismatch.

7 channels between the 8 DIMMs @ 1 cm separate. It shouldn't be too dificult to lay out these 48 southbound traces on the OpenSPARC T2 board, all next to each other in parallel, and the DIMM connectors minimally spaced for a field technician to install the DIMMs. Bear in mind that requirement for each set of 2 differential conductors have very stringent skew mismatch requirement to avoid the collapse of the eye at the next receiver.

But let's suppose it's D=4 cm of trace mismatch each time we go from one DIMM to the next DIMM. This allows for going through the connector, through the AMB package pins to the actual transceivers and associated delay between the AMB secondary northbound receiver and AMB primary northbound driver) =>

4 * 7 * 0.3 UI = 8.4 UI

Now, the long leg -- between the northernmost DIMM and the host-- contributes the most skew. potentially 6" mismatch in Intel's case (although they were from different channels and opposite directions)

Call it L=16 cm trace mismatch

16 * 0.3 UI = 4.8 UI

Add them up and we are 13.2 UI. This is 2 frames only.

1st order approximation of frames needed

frames = roundup (N * (7 * D + L) / 40)

N = settling time (# of wave propagations) of a tx/rcr pair.

D = max trace mismatch between a pair of DIMMs, in cm

L = max trace mismatch between host and northernmost DIMM.

3.5.3.3 Channel Alignment Logic across all Two FBDIMM Channels.

The FBDIMM Link specification does not specify the skews between 2 different channels, nor does it specify that the lane skew numbers are stricly for the same channel. If it can be assumed that the numbers for lane deskewing applies also regardless of number of channels as long as the host guarantees that it locks the 2 southbound channels then deskewing across 2 northbound channels can be done in 2 ways:

1. Expand the Lane alignment logic to support 28 simultaneous channels. This approach introduces less memory latency and alignment buffers but makes the search algorithm more complex (search within 928 permutations)
2. Use the 2 independent LAL outputs and add 2 sets of 9 deep FIFO and a delay counter or set of flops to record the delay (maximum 9 cycles apart) between one channel achieving lane lock and the other channel achieving lane lock.

To minimize bus width for a dual channel option, rather than route 168x2 (334) data bits per direction between the IO and MCU when both channels are used, a 168 bit data bus per direction is used and MCU runs at 2x dram speed (800MHz max). In both dual and single channel mode, fbd_mcu_data[83:0] always contain the primary channel's 1st half-frame the 1st cycle and the primary channel's 2nd half-frame the 2nd cycle. In dual channel mode, fbd_mcu_data[167:84] always contain the secondary channel's 1st half-frame the 1st cycle and the secondary channel's 2nd half-frame the 2nd cycle. In single channel mode, fbd_mcu_data[167:84] will always contain the 2nd half-frame on both cycles.

3.6 Memory Latencies

3.6.1 Read Latency

TABLE 3-20 shows the stages in the memory read pipeline and their approximate latencies for a 4-4-4 800 MHz DDR SDRAM with worst-case bit-lane deskewing.

The total latency does not include the time in the l2clk domain or the time to synchronize the data between the l2clk and drl2clk domains (A and I). These latencies assume that the MCU is idle when it receives the read request and that the request is going to the last of eight FBDs in the channel.

TABLE 3-20 Memory Read Pipeline and Latency

Stage	Clock Domain	Latency
A. L2 issues read request and MCU acknowledges	l2clk (1.4 GHz)	
B. MCU schedules read command	drl2clk (800 MHz)	7.5 ns (6 cycles)
C. Read request transmitted on SB FBD channel	fbdclk (4.8 GHz)	7.0 ns (1 ns per DIMM)
D. Bit-lane deskew	sdram clock (drl2clk / 2 = = 400 MHz)	22.5 ns (9 frames worst case)
E. Read command issued to SDRAMs	sdram clock	22.5 ns (9 cycles: CL + AL + 1 for frame alignment)
F. Read data returned to MCU on NB FBD channel	fbdclk	7.0 ns (1 ns per DIMM)
G. Bit-lane deskew	drl2clk	22.5 ns (9 frames worst case)
H. MCU checks for CRC and ECC errors	drl2clk	3.75 ns (3 cycles)
I. Read data returned to L2	l2clk	
Total		92.75 ns

The read requests from the L2 cache are placed in the Read Request Queue (RRQ) which is checked every drl2clk for a transaction. When a transaction enters the RRQ, there are two cycles for arbitration, two to issue the Activate command, and two to issue the Read command. (two drl2clk cycles == one sdram cycle). The Activate and Read commands are transmitted on successive cycles on the high-speed FBD channel which runs at 12 times the sdram speed. The Activate and Read commands are driven to the SDRAMs one cycle after each reaches the FBD. After the data returns from the SDRAMs, the AMB places the data in the appropriate NB frame, 144 bits per frame, to return to the MCU. Once the MCU receives the read data, it checks for CRC errors in the frame and ECC errors in the data. If no errors are found, the data is returned to the L2 cache 128 bits per drl2clk cycle.

3.6.2 Write Latency

TABLE 3-21 shows the stages latencies in the memory write pipeline. The total latency does not include the time in the l2clk domain or the time to synchronize the data between the l2clk and drl2clk domains (A and B). These latencies assume that the MCU is idle when it receives the write request and that the request is going to the last of eight FBDs in the channel.

TABLE 3-21 Memory write pipeline and latency

Stage	Clock Domain	Latency
A. L2 issues write request and MCU acknowledges	l2clk (1.4 GHz)	
B. L2 sends write data	l2clk	
C. MCU schedules write data and write command	drl2clk (800 MHz)	7.5 ns (6 cycles)
D. Write data and command transmitted on SB FBD channel	fbdclk (4.8 GHz)	7 ns (1 cycle per DIMM)
E. Bit-lane deskew	sdram clock (drl2clk / 2 = = 400 MHz)	22.5 ns (9 frames worst case)
F. Write command issued to SDRAMs	sdram clock	2.5 ns (1 cycle)
G. AMB issues Idle frame on NB FBD channel	fbdclk	7 ns (1 cycle per DIMM)
H. Bit-lane deskew	drl2clk	22.5 ns (9 frames worst case)
I. MCU checks for Alert frame	drl2clk	1.25 ns (1 cycle)
Total		70.25 ns

When the L2 cache issues a write request, the MCU transfers the request from the l2clk domain to the drl2clk domain, places it in the Write Request Queue (WRQ), and sends an acknowledge back to the L2 cache. Once the L2 receives the acknowledge, it transmits the write data to the MCU 64-bits per cycle over 8 cycles, and the MCU stores the write data in the Write Data Queue. The MCU sends the write data and write command independently on the FBD channel; however, it must ensure that the write data reaches the write data fifo within the AMB early enough that the AMB can write command the timing requirements to the SDRAMs. If the write command and data are received with no CRC errors, an Idle frame (as opposed to an Alert frame) is sent on the NB channel. Once the MCU sees there is no Alert frame, and thus no error on the write, it can the release the WRQ entry for the write.

3.7 Multiple Clock Domains

The MCU has three clock domains - l2clk, drl2clk, and iol2clk - and also interfaces to high-speed SERDES IOs for the DDR channels. The l2clk is the main cpu clock whose frequency is a multiple of the system clock, iol2clk. The l2clk and drl2clk are synchronous but do not have an integer ratio between them. The drl2clk will run at the same frequency as the SDRAM. The l2clk frequency target is 1.4 GHz, and the system clock target is 350 MHz. The SERDES IOs have a data rate of 12x the DDR rate - 3.2 GHz for 266 MHz DDR FBDs, 4.0 GHz for 333 MHz DDR FBDs, and 4.8 GHz for 400 MHz FBDs. The SERDES must run at one of these rates within +/- 5%.

The clock inputs to MCU are from the Clock Control Unit (CCU). The transmitting (l2if_cmp_ddr_sync_en, l2if_cmp_io_sync_en) and receiving (l2if_ddr_cmp_sync_en, l2if_io_cmp_sync_en) synchronization pulses are delayed versions of outputs from the CCU which act as clock enable for synchronizing Signals between two clock domains. The CCU will generate one of each of these enable pulses per MCU clock cycle.

Example waveforms for two clock ratios and the synchronizing signals across two clock domains are shown in FIGURE 3-16, FIGURE 3-17, and FIGURE 3-18. More detail on clock domain synchronization and the supported clock ratios can be found in the CCU specification.

FIGURE 3-16 Odd Ratio (13:2) Clock from the On-chip PLL Block

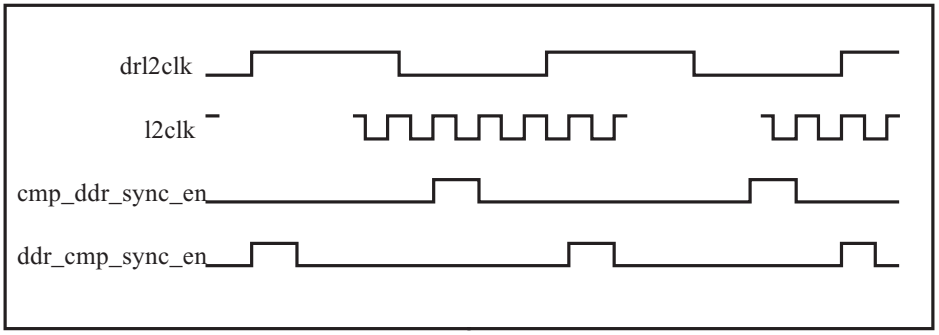


FIGURE 3-17 Even Ratio (12:2) Clock from the On-chip PLL Block

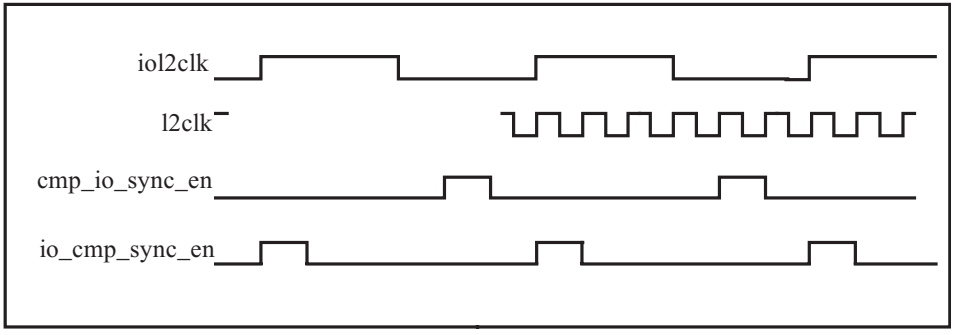
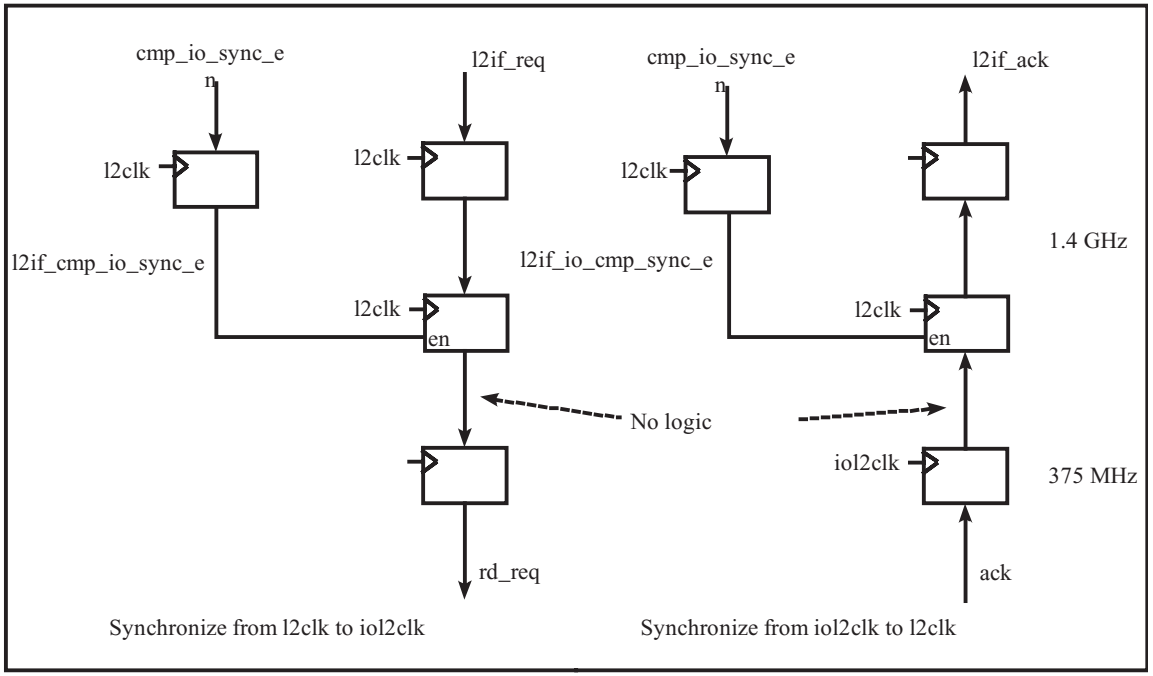


FIGURE 3-18 Example of Synchronizing between l2clk and iol2clS



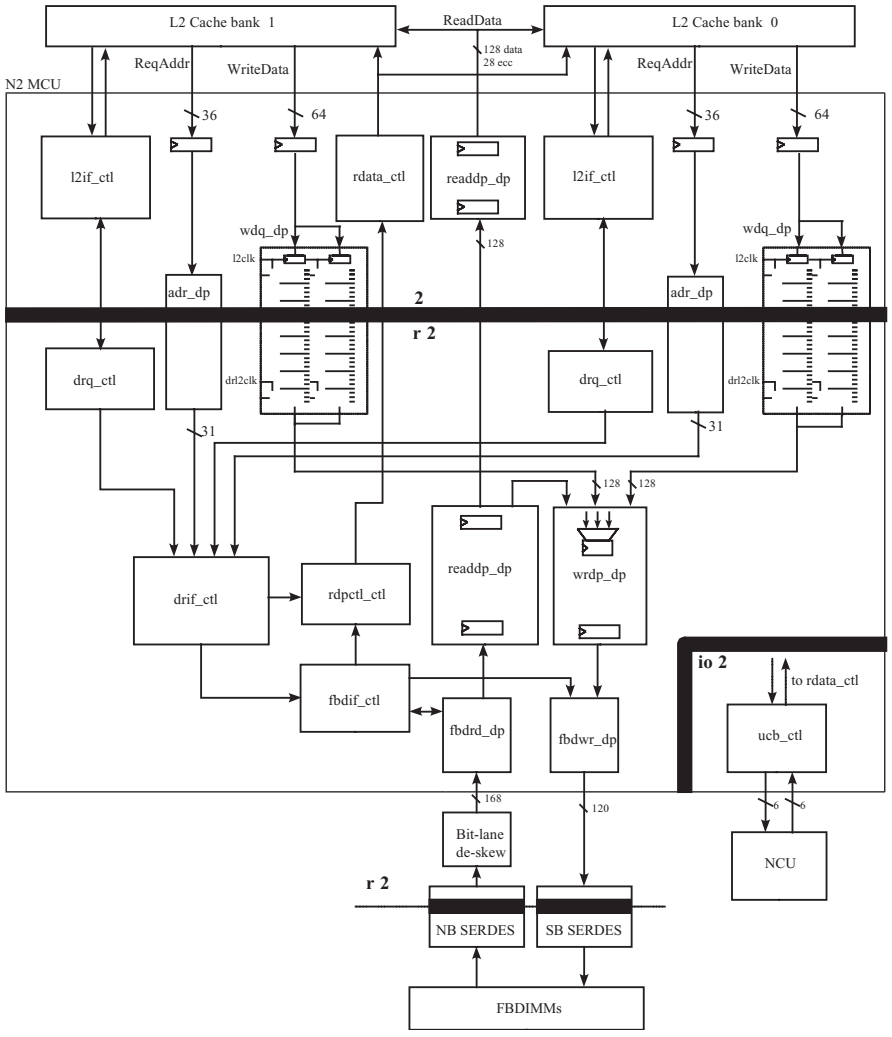
3.8 Functional Description

The MCU top level block diagram consists of the control logic and the datapaths to interface to the two L2 banks and the external DDR2 memory channel.The top level block diagram of the MCU is shown in FIGURE 3-19 with its 3 clock domains.

In FIGURE 3-18, requests come from the L2 cache banks to the L2 Cache Interface Control (L2IF_CTL) block. The incoming physical addresses are converted into DIMM addresses and stored in the Address Datapath (ADR_DP) block. For write requests, write data is stored in the Write Data Queue Datapath (WDQ_DP). The DRAM Request Queue Control (DRQ_CTL) block determines the order in which read and write transactions will go out to the DIMMs. The transactions are forwarded to the DRAM Interface Control (DRIF_CTL) block which generates the control Signals for the transactions going out to the DIMMs. The DRAM Write Datapath (WRDP_DP) block generates ECC for the write data going to the DIMMs. Read data returning from the DIMMs goes through the Read Data Datapath (READDP_DP) which checks ECC and corrects single-bit errors and regenerates

ECC for the data before it is returned to the L2 cache. The UCB logic unit provides the CSR interface for the MCU. Details of each of these blocks in given in the following sections.

FIGURE 3-19 MCU Block Diagram



3.8.1 MCU Datapaths

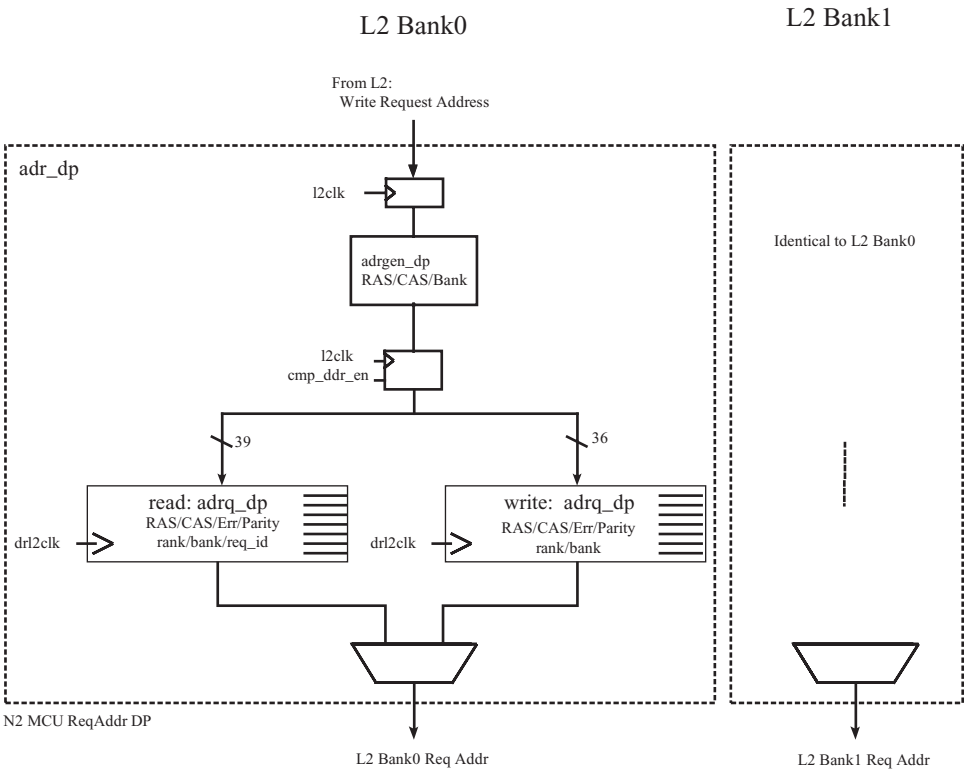
3.8.1.1 Request Address Datapath

The physical memory address of the L2 bank read requests and write requests from

the L2 cache are converted to the row and column addresses for the DIMMs in the system. The converted addresses are queued in the read request address queue or the write request address queue. These queues participate in arbitration for the Activate command cycle.

The block diagram of the request address datapath for the two L2 banks is shown FIGURE 3-20:

FIGURE 3-20 MCU Request Address Queue datapath



DDR2 Address Generation (ADRGEN_DP)

The Address Generation block converts the incoming physical address to an SDRAM address consisting of the row, column, and bank address bits, and, if they exist, rank selects. Also, parity is generated for the address, and an address error bit is generated if an address is accessed that is out of range for the installed DIMMs.

Inputs to the block are the physical address and the MCU configuration registers which give the number of row and column address bits, number of ranks, number of internal banks, and whether the rank selects should come from upper or lower physical address bits. All of the address information is stored in the Read or Write Request Address Queue, and the rank and internal bank information will also be sent to the MCU Request Queue Control (drq_ctl) module.

The converted DIMM address consists of the following 36 bits:

- RAS address (15b)
- CAS address (14b)
- SDRAM internal bank address (3b)
- DIMM rank select (2b)
- Request address error (1b)
- Address parity (1b)

Refer to section 5 for a description of how SDRAM address is converted from the physical address.

Read and Write Request Address Queues (ADRQ_DP)

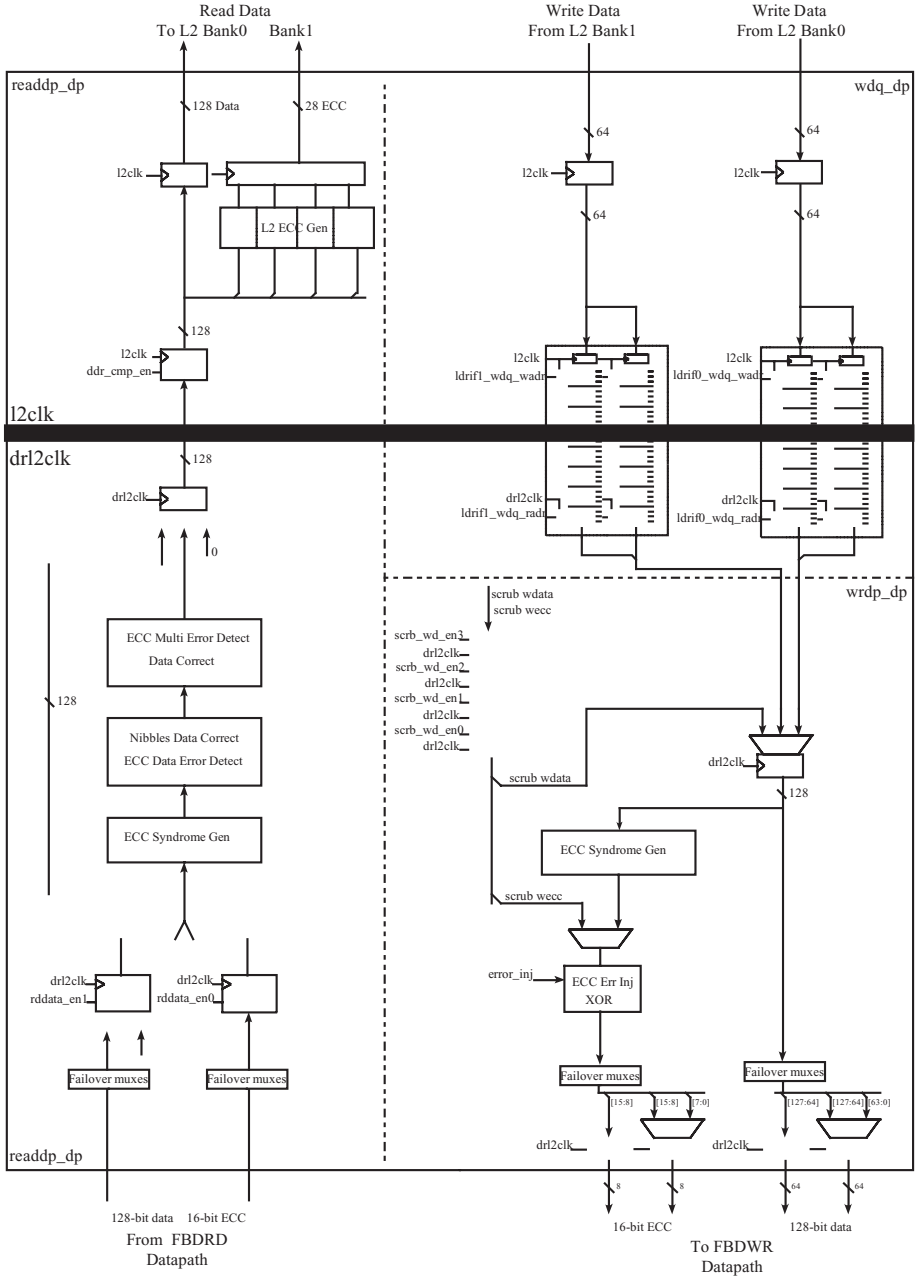
The Read and Write Request Address Queues store the converted DIMM addresses of the MCU memory requests. Each queue has eight entries of 36 bits. The 36 bits are the same as the converted DIMM address bits described in Section , "DDR2 Address Generation (ADRGEN_DP)" on page 3-52.

3.8.1.2 Read and Write Data Datapaths

The MCU read and write data datapaths block diagram is shown in FIGURE 3-21. These datapaths queue write data from the L2 cache to the DIMMs and read data returning from the DIMMs to the L2 cache.

The MCU read and write data datapath between the two L2 banks and the external memory channel consists of a write data queue per L2 bank (WDQ_DP), the write data datapath to the DRAM (WRDP_DP), and the read data return datapath to the L2 banks (READDP_DP). The MCU read and write data datapath supports 2 databus size interface modes to external memory - full size memory databus (128 bits data and 16 bits ECC) , and half size memory databus (64 bits data and 8 bits ECC).

FIGURE 3-21 Read and Write Datapaths Block Diagram



Write Data Queue Datapath (WDQ_DP)

The Write Data Queue stores the L2 write data to be written to the external memory. There are two Write Data Queues in each MCU module, one for each L2 cache bank that it communicates with. Each queue is composed of two dual port 1R/1W register files, each of which is 32 entries by 66 bits wide.

The L2 Cache Interface Control (L2IF_CTL) module controls write access to the WDQ_DP. The L2 write data is written to the write data queue at the rate of 64bits/cycle for 8 l2clk cycles to complete 1 L2 cache line (64 bytes). Write enables are used to enable writing to one register file at a time.

Read access to the WDQ is controlled by the DRAM Interface Control (DRIF_CTL) module in the drl2clk domain. The output ports of the register files are concatenated and read as a single 128-bit bus.

Write Data Datapath (WRDP_DP)

The Write Data datapath block sends write data to the IO pads. A multiplexer controlled by the DRIF_CTRL module selects between the two Write Data Queues and scrubbed data coming from the Read Data Return datapath.

16 bits of ECC is generated for each 128 bits of data, and 144 bits of data and ECC are sent to the SDRAM for four cycles in the normal configuration. In single-channel mode, the 144 bits data are multiplexed into 2 72-bit data packets and sent to the SDRAM over eight cycles.

Read Data Datapath (READDP_DP)

The 128 bits of data and 16 bits of ECC from the SDRAM are flopped in the drl2clk domain for ECC error detection and correction. In the single-channel mode, two 64-bit data packets are combined before ECC error detection and correction.

ECC is regenerated based on the 128 bits of data read from the SDRAMs and is compared with the ECC bits read from the SDRAMs. If a single bit error is detected, it is corrected in the data correction logic; however, if multiple errors are detected, no correction is done. The data are transferred from the dr2clk domain to the l2clk domain upon the assertion of the ddr_cmp_sync_en signal, or if the data is from a scrubbing request, it is sent to the Write Data datapath module.

All ECC errors on L2 cache reads and scrubbing reads are signaled to the L2 cache (mcu_l2t_scb_secc_err, mcu_l2t_scb_mecc_err, mcu_l2t_secc_err_r2, and mcu_l2t_mecc_err_r2) as well as being flagged in the MCU Error Status Register.

In the l2clk domain, the L2 cache ECC is generated on the 128 bits received from the drl2clk domain. The L2 cache uses a 7-bit Hamming code for to protect each 32-bit word, allowing single-bit error correction and double-bit error detection per 32 bits. (The algorithm for generating this L2 ECC is found in section A.4 of the OpenSPARC T2 Programmer's Reference Manual.) After ECC is generated, the 128 bits of data and 28 ECC bits are then sent back to the L2 cache.

3.8.1.3 FBD Write and Read Datapaths (FBDWR_DP, FBDRD_DP)

The datapath portion of the FBD Controller will stage write data or the B and C command portions of the frame and generate all of the CRC values. There are six CRC generators required for each FBD channel, four in the FBDWR_DP and two in the FBDRD_DP.

FIGURE 3-22 shows the FBD Write Datapath and the following four CRC generation blocks:

1. CRC[13:0] for command A portion of 10-bit mode SB frame, generated from 26 bits of data.
2. CRC[9:0] for command A portion of 10-bit failover mode SB frame, generated from 26 bits of data.
3. CRC[21:0] for command BC/data portion of 10-bit mode SB frame, generated from 72 bits of data.
4. CRC[9:0] for command BC/data portion of 10-bit failover mode SB frame, generated from 72 bits of data.

FIGURE 3-22 FBD Write Datapath

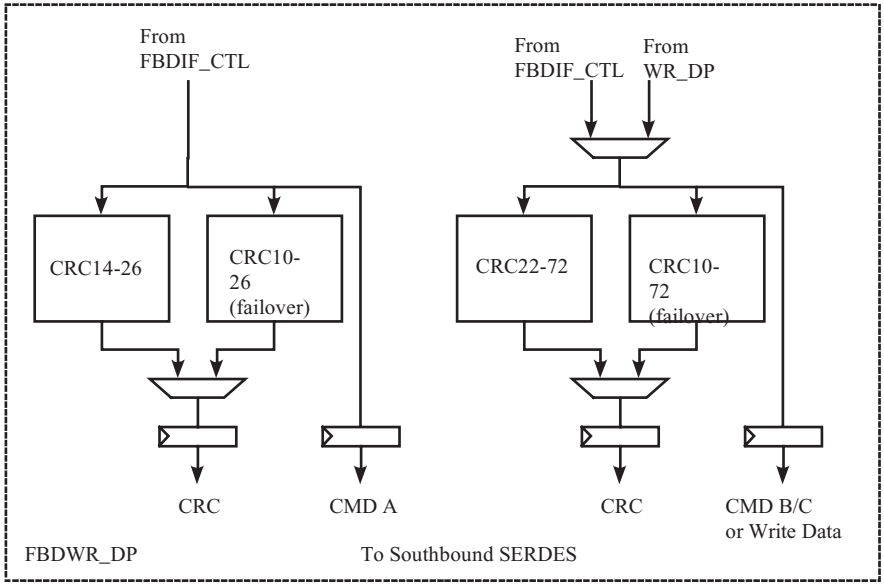
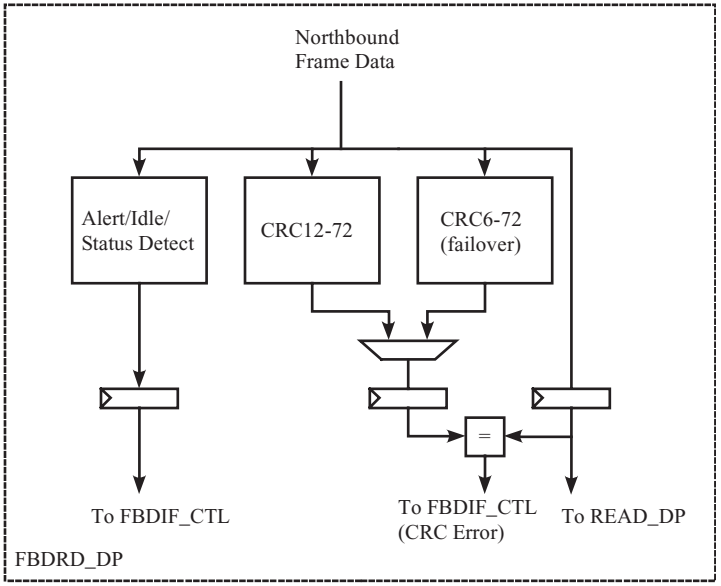


FIGURE 3-23 shows the FBD Read Datapath and the following two CRC generation blocks:

1. CRC[11:0] for read data of 14-bit mode NB frame, generated from 72-bits of data, compared to transmitted CRC.
2. CRC[5:0] for read data of 14-bit failover mode NB frame, generated from 72-bits of data, compared to transmitted CRC.

If there is a CRC error on a read frame, the FBDRD_DP Signals the DRIF_CTL and FBDIF_CTL blocks for error reporting and to try to recover from the error.

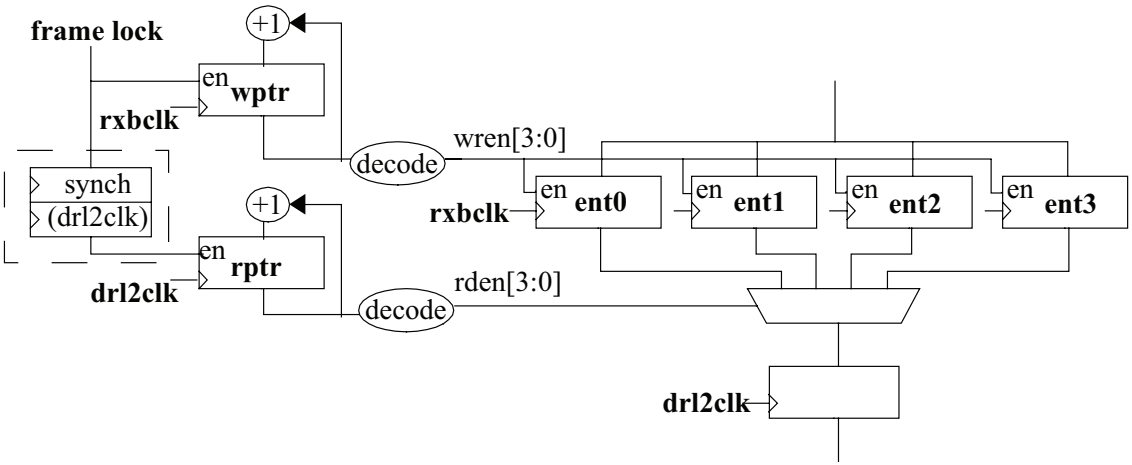
FIGURE 3-23 FBD Read Datapath



3.8.1.4 FSR to MCU Cross-Domain FIFO (FBD_DP)

The recovered clock from the SERDES runs at the same speed as the drl2clk; however, the two clocks are asynchronous. Therefore, an asynchronous FIFO is needed to pass data between these clock domains. A FIFO is required for each northbound bit lane since each generates its own clock. The FIFOs are implemented in flip-flops. The FIFOs are 12 bits wide and 4 entries deep. Once enabled, data will be written to the write port every cycle and read from the read port every cycle. The write pointer is in the recovered clock domain and is enabled once the MCU has seen 16 sync Signals from the SERDES receiver. The read pointer is in the drl2clk domain and is enabled by the same write pointer enable signal after being sent through a synchronizer. After both the read and write pointers are enabled, data placed into the FIFO should be stable for approximately 2 clocks drl2clk's before being read out. As long as the phase difference between the two clocks does not drift by +- 1 period, no data will be lost. If the phase drift is greater that +-1 period, data may get dropped in which case the MCU will detect an error and may have to retrain the link to reestablish the cross-domain relationship.

FIGURE 3-24 FBD Cross Domain Logic

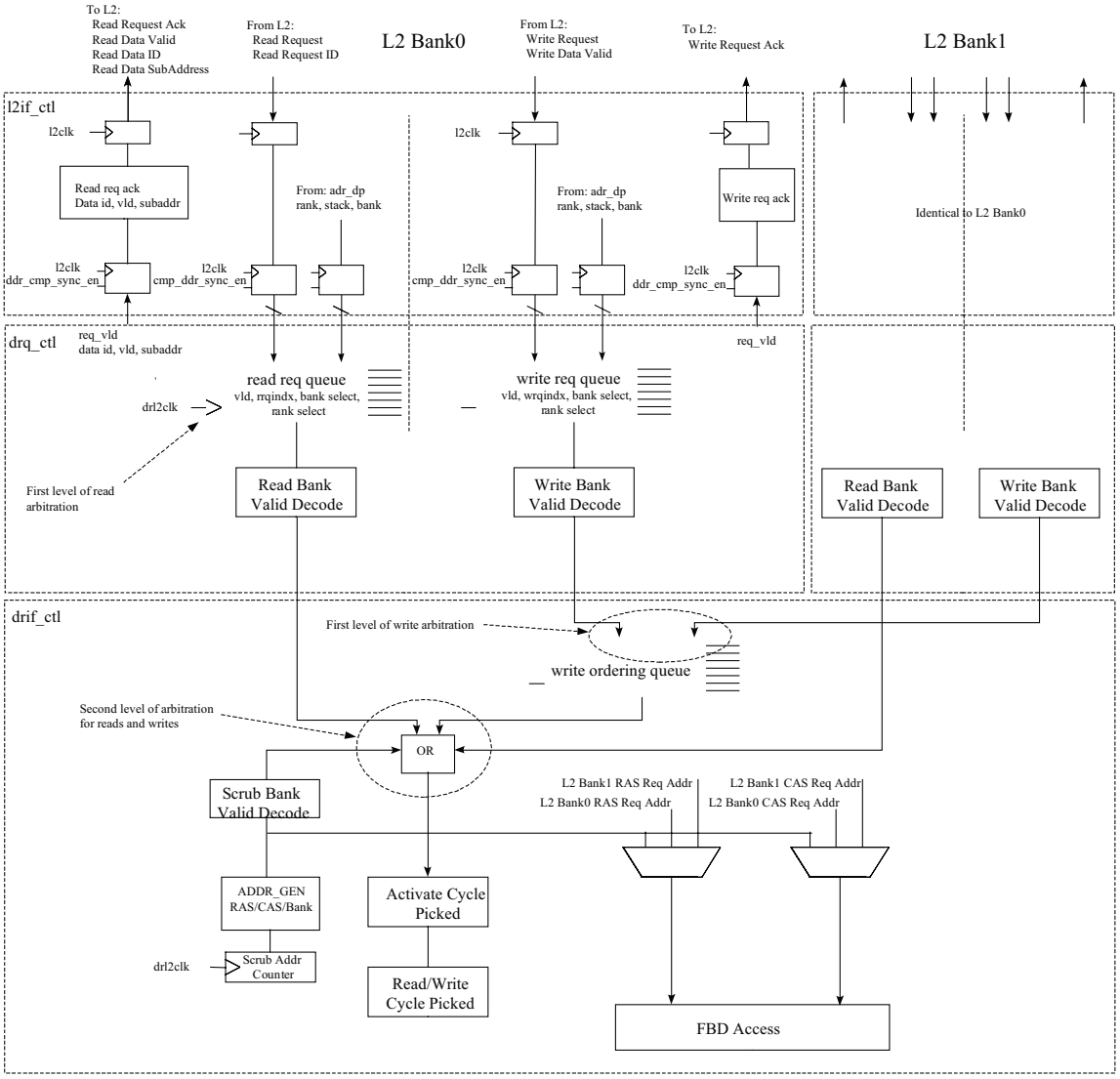


3.8.2 MCU Control Logic

The MCU control logic provides the interface between the two L2 banks and the external memory channel. The MCU control logic block diagram is shown in FIGURE 3-25:

Requests enter the L2IF_CTL blocks from the L2 cache banks. The requests are passed to the DRQ_CTL blocks with the cmp_ddr_sync_en Signals where they are placed in the Read or Write Request Queues. Write arbitration occurs between the two Write Request Queues, and the requests are then placed in the Write ordering queue. Read request arbitration occurs between the requests in one Read Request Queue. The second level of arbitration selects a request from the Write Ordering Queue or one of the Read Request Queues to issue to the FBDs. Details of the control flow are given in the following sections.

FIGURE 3-25 MCU Control Logic Block Diagram



3.8.2.1 MCU - L2 Cache Interface Control (L2IF_CTL)

The L2 Cache Interface Control module handles incoming read and write requests from the L2 cache. It controls the synchronization of these requests to the drl2clk domain, the writing of data to the Write Data Queues, and the return of read data to the L2 cache or scrubbed data back to the SDRAMs. Section 6 discusses the protocol

the MCU uses for communicating with the L2 cache.

When a read request comes in from the L2 cache, it is held in a staging register in the L2IF_CTL block and loaded with the signal l2if_cmp_ddr_sync_en. The acknowledge is always sent back one cycle after l2if_cmp_ddr_sync_en, and the request information is moved to the MCU Request Queue Control block at the rising edge of the next drl2clk.

Write requests are handled similarly except that the L2 cache does not limit the number of write requests it issues, so the MCU must handle this. The L2IF_CTL block receives information from the DRQ_CTL block telling how many Write Request Queue entries are free. If all eight entries are used, the ninth request is held in the L2IF_CTL block until one frees up, at which time the write request information is passed to the DRQ_CTL. Once the acknowledge is sent to the L2 cache, the L2 cache will start sending the write data 64 bits at a time, and the L2IF_CTL will control the loading of data into the Write Data Queue.

3.8.2.2 MCU Request Queue Control (DRQ_CTL)

Upon receiving a read or write request from an L2 bank, the MCU Request Queue Control (DRQ_CTL) logic generates the write enables to enqueue the incoming request in the read or write request address queue. The information required for request arbitration is stored in the read or write request queue within the DRQ_CTL block. Every drl2clk cycle, the arbitration logic must look at all entries in the read and write request queues in parallel to determine the next request to be scheduled. A request's entry in the read or write request queue will be invalidated upon the completion of the memory access.

The Read and Write Request Queues are implemented in registers as collapsing queues. The newest entry is placed at the tail of the queue; however, the oldest entry is not necessarily the first to be removed. The arbitration algorithm decides which entry to select, and when it is removed, the remaining queue entries which entered after it collapse to fill the empty entry.

Read Request Queue (RRQ)

The following information from the incoming L2 read request is stored in the 8-entry read request queue:

1. L2 read request valid - 1b
2. Index of DIMM address in the read request address queue - 3b
3. DIMM rank select - 2b
4. DIMM internal bank select - 3

This information is used by the arbitration logic for request scheduling.

Write Request Queue (WRQ)

The following information from the incoming L2 write request is stored in the 8-entry write request queue:

1. L2 write request valid - 1b
2. Index of DIMM address in the write request address and write data queues - 3b
3. DIMM rank select - 2b
4. DIMM bank select - 3b

This information is used by the arbitration logic for request scheduling.

3.8.2.3 Write Ordering Queue (WOQ)

The Write Ordering Queue controls the issuing of write data to the FBDs. The WOQ uses round-robin arbitration to choose between the two write request queues when selecting a request to issue. Within a WRQ, the WOQ looks for a transaction going to a different FBD than where the previous two transactions were issued. (This helps the MCU be able to schedule multiple write requests later.) If a transaction for a different FBD is not found, the first request in the WRQ is selected. The data corresponding to the selected request is sent to be buffered in the AMB and the request is placed in the WOQ. This ensures that write requests will be sent to the FBDs in the same order that the write data was sent.

When a write request is issued to an FBD, the read pointer for the WOQ is incremented; however, the information for the request remains in the WOQ data structure. Another pointer (the outstanding write transaction (owt) pointer) points to the head of all requests that have been sent to the FBDs but have not been verified as having completed. When a write is issued, the MCU waits that time that a read would take. Once this time elapses, the owt pointer is incremented; however, the transaction is not considered complete until an Idle Frame or a Status Frame has been seen after this point. A third pointer , the woq error pointer, points to the head of requests that should have completed but for which the MCU has not yet seen an Idle or Status Frame. If an error occurs, all writes from this woq error pointer to the tail of the WOQ will be reissued. If no error occurs, the woq error pointer is updated to the location of the owt pointer once an Idle or Status Frame is received.

The WOQ has 16 entries and holds the same information as the WRQ; however, each entry has an additional bit that points to the source L2 bank.

3.8.2.4 MCU - DDR2 Interface Control (DRIF_CTL)

The MCU arbitrates among 3 memory access request sources - memory refresh requests, memory scrubbing requests, and L2 cache memory access requests. The priority among the 3 request types is as follows:

1. Memory refresh
2. Memory scrubbing requests
3. L2 cache read and write requests

Memory Refresh Request

DDR2 SDRAMs require a refresh of all rows within the memory every 7.8s. The MCU has a programmable refresh counter, clocked with drl2clk, to keep track of the refresh time interval. At every refresh interval, a memory refresh request is issued successively to each rank present.

With a drl2clk of 400 MHz, the programmable refresh counter value is calculated as follows:

refresh counter value = 7.8s / 2.5ns = 3120 (0xC30)

Memory Scrubbing Request

At intervals defined by the DRAM Scrub Frequency Register, scrub read requests are issued to the SDRAMs. The purpose of these requests is to detect and correct transient memory errors. More detail on the scrubbing procedure is given in the RAS section of the document.

First-level Write Request Arbitration

Because write data is buffered in the AMBs on the FBDs, it is advantageous to send the write data to the DIMMs early and arbitrate for the write requests later. When write requests are available in the write request queue, the write scheduler will issue the write data in frames to the FBDs whenever free time slots are available. When a write request is selected, its data is sent out to the FBDs, and the request is placed in the Write Ordering Queue for the second level of arbitration. When the write commands are eventually issued, the ordering of write transactions to a given FBD must be maintained; however, write transactions to different FBDs can bypass each other.

First-level Read Arbitration

Each Request Queue Control block (DRQ_CTL) sends read memory requests to the DRIF_CTL arbiter, and the Write Ordering Queue within the DRIF_CTL provides write requests to the arbiter. The DRIF_CTL uses a first-come-first-served algorithm at the first level of arbitration for reads, selecting the oldest read request in the Read Request Queue whose bank is available.

Second-level Read and Write Arbitration

At the second level of arbitration, the DRQ_CTL does a round-robin selection of requests from the two DRQ_CTL read request queues. Reads have highest priority and will be scheduled when ready. A maximum of one read per frame can be scheduled. One or two write requests from the write ordering queue can be scheduled simultaneously with the read if they target different FBDs than the read. If there are no read read requests ready, up to three write requests can be scheduled if they all target different FBDs.

Each read or write transaction must be issued over two consecutive cycles. On the first cycle, an Activate command is issued to activate a bank and row within an FBD. On the following cycle, the read or write command is issued with the bank and column addresses. The MCU uses the DDR2 SDRAM Posted CAS feature, so the read or write commands are delayed internal to the SDRAMs by the Additive Latency (AL) value programmed in the SDRAMs' extended mode register.

An Auto-Refresh command is issued when the MCU state machine transitions to the refresh state. Transactions to the rank being refreshed are blocked until the refresh completes.

Requests issued to the DIMMs are arbitrated every drl2clk cycle and are prioritized as follows, in order of descending priority:

Command slot A:

1. Scrub Read Activate requests
2. Read Activate requests from read request queues
3. Write Activate requests from write ordering queue

Command Slot B:

1. Autorefresh
2. Write data
3. Write Activate request

Command Slot C:

1. Power Down mode exit
2. Power Down mode enter
3. Write data
4. Write Activate request

Write Starvation Prevention

A write starvation counter ensures that writes do not get starved out. There is a separate write starvation counter in each DRQ_CTL block. Initially, reads have priority, and each write starvation counter is incremented whenever there are 8 pending write requests total (between the DRQ and the WOQ). It is reset whenever there are less than 8 pending writes or when a write request is issued from the WOQ. If either starvation counter reaches 64 (meaning there has been 8 pending writes for 64 consecutive cycles), then starvation mode is entered. Once in starvation mode, write activation requests from the write ordering queue are given priority over read activate requests (i.e. the priorities of 2 and 3 for Command Slot A above are reversed). During starvation mode, the starvation counter that reached 64 is decremented each cycle, and once it reaches 0, starvation mode is exited, and reads are again given priority.

Scheduling Writes in Command Slots B and C

When there are no higher priority requests for command slots B and C, write requests can be sent out. These write requests can only be sent to DIMMs whose read-to-write delay has been satisfied, and they must be scheduled so that no write data collisions occur on the FBD data buses. Also, in order to simplify scheduling, these requests cannot be selected on the same cycle as a read or write in command slot A.

Read-after-Write Hazards

It is possible for the MCU to receive a read request to a physical address that matches the address of a write request pending in the write request queue. If this occurs, the MCU must ensure that the write completes first. (Whenever there is an address match between a read and a write, it is guaranteed that the write has preceeded the read. Write-after-read hazards are handled by the L2 cache itself.)

When a read transaction wins arbitration, its address is compared against all of the valid entries in the write request queue for the L2 bank from which the read request came. If there is a match, the read request is not sent out, and the write request

queue entry that matches the read is flagged. Writes are given priority until the flagged write request queue entry is issued to the FBDs. After the write request is issued, reads are given priority.

Scheduling State Machines

In a fully populated FBD channel, there can be 8 dual rank dimms for a total of 16 ranks. Each rank supports DIMMs with up to 8 banks. Thus, there can be 128 banks per channel.

There are 16 state machines to keep track of available banks. Each is dedicated to a set of DRAM banks depending on the configuration. The equations below show which state machine will be used for a given address:

4-bank DRAMs, double-sided DIMMs: state_machine = {dimm[0], rank, bank[1:0]};

4-bank DRAMs, single-sided DIMMs: state_machine = {dimm[1:0], bank[1:0]};

8-bank DRAMs, double-sided DIMMs: state_machine = {rank, bank[2:0]};

8-bank DRAMs, single-sided DIMMs: state_machine = {dimm[0], bank[2:0]};

ECC Error Handling

The DRIF_CTL block handles errors that have occurred. Information on the read request that caused the error is received from the RDPCTL_CTL block and stored in a register FIFO. The FIFO has eight entries to hold all outstanding reads that may generate ECC errors. Once an error is detected, the DRIF_CTL stops issuing requests, and after all outstanding requests have completed, it begins retrying each transaction from the error FIFO. Refer to Section 12.4 for an explanation of the error handling scheme.

3.8.2.5 FBD Interface Control (FBDIC_CTL)

The FBD Interface Control block is responsible for FBD channel initialization, channel error detection, and frame encoding and decoding. All of the channel configuration registers also reside in this block.

At power-on, software is responsible for sequencing the FBD controller through the initialization sequence. The FBD channel initialization sequence is described in Section 3.3.1, “FBD Channel Initialization” on page 3-11..

After initialization is complete and the AMBs are in the L0 state, software must program various registers in the AMBs using channel commands, and when it is ready to accept SDRAM commands, the FBDIC_CTL Signals the DRIF_CTL. The

FBDIC_CTL will encode the channel and SDRAM commands using the frame formats described in section 4.2.1 using the CRC data provided by the FBD_DP block. This information is sent to the FBD SERDES IO block to be transmitted to the FBDs.

The Latency Queue (latq) informs the FBDIC_CTL when read data is returning. When a request is issued, a timestamp is placed in the latq (current time plus channel latency). When the current time matches the timestamp at the head of the latq, the read data (or other reponse, such as a Status Frame) is expected. If a CRC error occurs or an Alert or Idle Frame is detected, the DRIF_CTL is signaled so that it can retry the read. If an error is detected on the second read, the FBDIC_CTL module tries to recover with a Soft Channel Reset or a Fast Reset, if necessary. If neither of these revives the FBD channel, software is signalled to handle the recovery.

If a Status Frame shows an error has occurred or if an Alert Frame is received on the NB channel, this indicates an error has occurred on the SB channel. Again, a Soft Channel Reset, and possibly a Fast Reset, will be issued, afterwhich software will signalled if the condition persists.

3.8.2.6 MCU Read Datapath Control (RDPCTL_CTL)

The Read Data Path Control module controls the portion of the READDP_DP within the drl2clk domain and prepares the data valid and error Signals to be returned to the L2 cache banks. Error logging is also performed in this block.

The following information is received from the DRIF_CTL block to keep track of the read requests outstanding to the FBDs:

- L2 bank - 1 bit
- read/write - 1 bit
- starting quadword - 2 bits
- read request id - 3 bits
- location in read or write request queue - 3 bits
- out-of-bound address error on read - 1 bit

The information is stored in a FIFO implemented in registers. The FIFO depth is 16 and only holds outstanding reads. Reads are freed from this FIFO when a read transaction's data returns correctly.

If a CRC error occurs on the Southbound channel, all transactions in the RDPCTL_CTL FIFO must be retried in their original order after the channel is reset. If a CRC or ECC error occurs on the Northbound channel, only the transaction with an error must be retried.

If a CRC error occurs on the Southbound channel, all transactions in the RDPCTL_CTL FIFO must be retried in their original order after the channel is reset. If a CRC or ECC error occurs on the Northbound channel, only the transaction with an error must be retried.

3.8.2.7 MCU Read Data Control (RDATA_CTL)

The Read Data Control block sends the data valid, qword id and read request id Signals to the L2 cache banks, and responds to the L2 cache dummy read requests. It also generates the address generation control Signals for the ADRGEN_DP blocks in the Address Datapath block and acts as a bridge between the IO and MCU clock domains for CSR reads and writes.

3.8.3 UCB CSR Interface

The Unit Control Block (UCB) provides the CSR interface to the MCU. The NCU communicates with the UCB module through a 4-bit bus. For register writes, the UCB assembles the 4-bit packets received into a 32-bit address and a 64-bit data word, and conversely for reads, breaks the 64-bit read data into 4-bit packets to send back to the NCU.Interconnect Built-In Self Test (IBIST) Engine

The FBDIMM standard requires an IBIST engine within the MCU that will stress the FBDIMM channel electrical connections. When the FBDIMM channel initialization reaches the Testing stage, the IBIST engine will take control of the channel after the TS1 header is issued.

The following registers are implemented by the MCU:

SBFIBPORTCTL: 0x84_0000_0E80

Since the MCU will always be a master on the southbound port, bit 1 and bits 6 through 23 of this register are not used by the MCU and will be read only. Bit [1] will be 1'b1, and bits [23:6] will be 18'h00000.

SBFIBPGCTL: 0x84_0000_0E84

SBFIBPATTBUF1: 0x84_0000_0E88

SBFIBTXMSK: 0x84_0000_0E8C

SBFIBTXSHFT: 0x84_0000_0E94

SBFIBPATTBUF2: 0x84_0000_0EA0

SBFIBPATT2EN: 0x84_0000_0EA4

SBFIBINIT: 0x84_0000_0EB0

SBIBISTMISC: 0x84_0000_0EB4

NBFIBPORTCTL: 0x84_0000_0EC0

Since the MCU will always be a slave on the northbound port, Bits 0, 1, 22, and 23 will not be used by the MCU and will be read only. Bit [1] will be 1'b0, and bits [23:22] will be 2'h0.

NBFIBPGCTL: 0x84_0000_0EC4

NBFIBPATTBUF1: 0x84_0000_0EC8

NBFIBRXMSK: 0x84_0000_0ED0

NBFIBRXSHFT: 0x84_0000_0ED8

NBFIBRXLNERR: 0x84_0000_0EDC

NBFIBPATTBUF2: 0x84_0000_0EE0

NBFIBPATT2EN: 0x84_0000_0EE4

The following registers are not implemented:

SBFIBRXMSK

SBFIBRXSHFT

SBFIBRXLNERR

NBFIBTXMSK

NBFIBTXSHFT

NBFIBINIT

NBIBISTMISC

3.9 SDRAM Power Reduction and Reduced-Configuration Operating Modes

The SDRAMs in a system consume a large portion of the power budget and ways to limit the power consumption in certain configurations or at certain times. A single-channel mode is available that allows one DIMM per MCU channel, power throttling limits the number of SDRAM transactions over a period of time, and the SDRAMs can also be put in self refresh modes.

3.9.1 Single Channel Mode

Normally, the memory will be configured in dual-channel mode. In order to reduce system power, a single-channel mode has been added which supports one DIMM per channel. In this mode, 72 bits of data and ECC are driven externally per memory cycle. The burst length for this mode is 8 to maintain the 64-byte cache line size per memory transaction.

To enable single channel mode, the single channel mode register, address 0x84_0000_0148 must be set to 1. Also, for proper operation, the Trrd (0x84_0000_0080) and Trc (0x84_0000_0080) must be increased by 2.

3.9.2 MCU Programmable Power Throttle

There are two registers per controller that control power throttling. The DRAM Open Bank Max Register designates the maximum number of DRAM bank openings that can occur in a time period. The time period is determined by the DRAM Programmable Time Counter Register whose value is a count of DRAM clock cycles. There is a counter that counts the number of DRAM banks that are opened. If this counter exceeds the maximum number of open banks, the DRAM controller is blocked from issuing anymore DRAM accesses until the counter is reset. A second counter counts DRAM clock cycles. When this counter is greater than or equal to the programmable time counter value, both this counter and the DRAM open bank counter are reset to zero.

The registers in the four controllers should be programmed to the same values. The chip will still operate correctly if they are programmed differently, but there may performance penalties (e.g. if one controller stops much earlier than the others) and power may not be as effectively controlled. Also, a mechanism is needed to ensure that all of the controllers are working with in the same time window of DRAM clock cycles. This will ensure that all controllers stop and start at approximately the same

time. When any of the Programmable Time Counter Registers is written, a reset signal will be sent to the other three controllers to reset their DRAM clock cycle counters.

3.9.3 SDRAM Self-Refresh Mode

The DDR2 SDRAMs support Self-Refresh mode allows the OpenSPARC T2 MCU to be reset without data loss in the SDRAMs.

For Self-Refresh mode, when the clock control unit Signals the MCU to enter this mode, the MCU waits until all requests have completed and then issues a Self-Refresh Entry command to the FBDs. In order to leave this mode, the clock and other external control must be stable for at least one clock cycle. The MCU issues a Self-Refresh Exit command and waits 200 cycles before returning to normal operation.

3.9.4 FBD L0s State

Some AMBs provide a low-power state. When an AMB receives a Sync frame with the 'Enter L0s' bit set, it transitions to the L0s state for a time period determined by its L0s_Duration register. This register value is between 32 and 42, and must be less than the minimum interval between Sync frames defined in the AMB's Sync Train Interval register. Once the timer for the L0s state expires, the AMB transitions back to the L0 state. After exiting the L0s state, the first command that the host must issue is another Sync frame in order to ensure that the AMB clocks remain locked. With this Sync frame, it is also possible to put the AMBs back into the L0s state for another low-power interval.

The MCU must be programmed to decide when to transition to this state. This is enabled by setting bit [6] of the L0s Duration Register. When this mode is enabled and there are no pending transactions when a Sync frame is being sent out, the el0s bit will be set in the Sync frame.

There is Thermal_Trip information returned in NB status frames which indicates that a thermal threshold has been exceeded and that power throttling may be required. This information is held in the Thermal Trip Status Register that software can check to decide when to enable L0s mode.

3.9.5 Power Down Mode

Power Down mode is a low power mode for the DRAMs. The MCU can optionally use this mode. When a transaction enters the MCU, a counter for the destination DIMM is incremented. When the transaction completes, DIMM counter is decremented. When any counter goes from 0 to 1, an Exit Power Down command will be sent to the corresponding DIMM. When a counter goes from 1 to 0, an Enter Power Down command will be sent to the DIMM.

3.9.6 Partial Bank Mode

Partial bank mode is a mode where less than 8 L2 banks are used in the system. In 4-bank mode, 2 MCUs are used and in 2-bank mode, 1 MCU is used. In these modes, the addressing to the DIMMs is changed. In 4-bank mode, the MCU left-shifts address bits [39:7] by 1 bit before applying the normal address decoding. In 2-bank mode, the MCU left-shifts address bits [39:7] by 2 bits before applying the normal address decoding.

When in partial bank mode, the MCU will detect out-of-bound errors on a smaller address range for a given configuration as compared to a system with all MCUs enabled. In 4-bank mode, the system will have half of the memory of a full system with the same configuration and one-fourth the memory when in 2-bank mode. The MCUs must be configured with twice the memory in 4-bank mode and 4 times the memory in 2-bank mode to provide the same address space as a full system. If the full system uses the maximum memory configuration, then the tests must reside in the lower half or quarter of memory in order to execute properly in a 4-bank or 2-bank configured system, respectively.

3.10 RAS Features

3.10.1 SDRAM ECC

The data sent to the DRAMs is protected by SEC-DED error correction. Galois field multiplication techniques are used to generate 16 bits of ECC in this block for each 128 bits of data.

3.10.2 Memory Scrubbing

Memory scrubbing refers to the regeneration of ECC for data in memory and the correction of single-bit errors and detection of double-bit errors. When scrubbing is enabled, at the end of the time interval defined by the DRAM Scrub Frequency Register, a memory scrub request is issued to the DIMMs. The scrubbing requests have priority over L2 cache requests. First, a scrubbing read request is issued to the DIMMs. When the scrubbing read data returns, the error detection and correction logic is used on the data. ECC is regenerated and compared with the ECC data read from memory. If an error is detected, a single-bit and double-bit error is flagged in the DRAM Error Status Register as well as being signalled to the L2 cache; then the MCU generates additional requests to the SDRAMs to collect more information on the error which is detailed in the following section. After the scrubbing transaction completes, the L2 cache requests are able to proceed.

Once a scrubbing request is sent, the time interval counter is reset and begins counting down again, and the scrub address is incremented to the next memory location.

3.10.3 Data Poisoning

Data poisoning involves marking known corrupt data in memory with bad ECC so that any later access will get an ECC error. MCU memory poisoning is performed by flipping ECC check bits 15, 9, 5 and 0. This will generate a failing syndrome of 0x8221 which, when encountered on a read, will most likely indicate poisoned data. The L2 cache asserts l2b_mcu_data_mecc which causes the MCU to corrupt the ECC for the corresponding 64-bit data word.

3.10.4 ECC Error Handling

When an error occurs on a scrub read or an L2 cache read request, the MCU will flag the error and then try to determine if the error is a hard error or a transient error. After the error occurs, the MCU will first perform another Read and log its ECC status. If the second read does not have an uncorrectable error, the corrected read data is written back to the SDRAM, and a third read is issued, and its status is also logged. Only the status from the first read will be sent to the L2 cache bank. One of the cores must perform a register read to check the status of the subsequent reads.

3.10.5 FBD Channel Errors

There are several ways that errors may be detected on the SB or NB channels. Alerts and Status Frames show when CRC errors have occurred on the SB channel. CRC errors on data frames and corrupted Idle or Status Frames show errors on the NB channel. When a channel error occurs on the SB channel, all transactions not guaranteed to have completed before the problem was detected must be reissued. When an error occurs on the NB channel, only the transaction with an error must be reissued.

1. Alert Frame: The MCU will stop issuing transactions and will issue a Soft Channel Reset (SCR) frame to attempt to reset the state of the AMBs and try to determine which AMB has detected an error. If errors persist, the MCU will issue a Fast Reset. If errors still persist after the Fast Reset, the MCU will log an Unrecoverable error in the MCU ESR, log an Alert Frame error in the MCU Syndrome Register and assert mcu_l2t0_scb_mecc_err to the L2. If at any point in the error processing the MCU is able to recover from the error condition, the MCU will instead assert mcu_l2t0_scb_secc_err to the L2 and set the Recoverable error bit in the ESR. Any outstanding reads or subsequent reads must still be returned to the L2 after the error processing is completed. If the channels are not working, these reads will be seen as Unrecoverable CRC errors.

2. Status Frame with Alert asserted: This indicates that the asserting AMB has detected an error on the SB channel. If this is the only error detected, the MCU will wait for the next Status Frame. If no other error occurs before or in the next Status Frame, the MCU will flag an Alert Asserted error in the syndrome register, set the Recoverable error bit in the MCU ESR, and assert mcu_l2t0_scb_secc_err to the L2. If any other error type occurs before or in the next Status Frame, processing proceeds as for that error condition.

3. Status Frame Parity Error: If there is a parity error in a Status Frame, the MCU will wait for the next Status frame. If the error persists, the MCU will attempt a Soft Channel Reset and if necessary, a Fast Reset. If the error persists the MCU will flag a Status Frame Error in the syndrome register, log a Status Parity error in the MCU Syndrome Register and assert mcu_l2t0_scb_mecc_err to the L2. If at any point in the error processing tthe MCU is able to recover from the error condition, the MCU will instead assert mcu_l2t0_scb_secc_err to the L2 and set the Recoverable error bit in the ESR. Any outstanding reads or subsequent reads must still be returned to the L2 after the error processing is completed. If the channels are not working, these reads will be seen as Unrecoverable CRC errors.

4. CRC Error on Read Data: The MCU will discard the data and retry the transaction. If it fails again, the MCU will issue a Fast Reset of the FBD channel. If the error persists after the Fast Reset, the MCU will flag an Unrecoverable error in the ESR, flag a CRC error in the Syndrome register and assert mcu_l2t_mecc_err_r3 to the L2 along with the bad data. If there is no CRC error on any of the retries, the MCU will flag a Recoverable Error in the ESR and assert mcu_l2t0_scb_secc_err once to the L2 and then send the correct read data.

When a Soft Channel Reset command is issued to the FBDIMMs, it will be followed by a CKE command (to enable all CKEs) and a precharge all command to put the DRAMs in a legal operating state. In order for the CKE commands to be issued correctly, the FBD Per Rank CKE Register must be set correctly and the CKE bit in the DIMM Initialization Register must be set.

When two channels are operating in lock-step, the MCU will perform error handling as if the same error occurred on both channels.

3.10.6 Interrupts

The MCU has two interrupt types that it can send to the NCU based on certain MCU errors. The error types are Correctable ECC Error Count and Recoverable FBD Channel Error Count . When one of these errors is generated, the MCU will send a single cycle pulse to the NCU in the iol2clk domain. Either of FBD errors will also generate a syndrome which is stored in the MCU Syndrome register if no other FBD error is pending.

The NCU also has a mechanism for generating these error types and an Unrecoverable FBD Channel Error. When one of these Signals is asserted, the MCU will inject that type of error within its logic to verify that the error detection and reporting logic is operating correctly.

Correctable ECC Error Count Interrupt

When the MCU Error Count Register reaches zero, the MCU will generate an interrupt on mcu_ncu_ecc, asserting it for one iol2clk cycle. No syndrome is reported, and no more of this type of interrupt will be generated until the software writes to the Error Count Register to enable interrupt generation.

If ncu_mcu_ecci is asserted, the MCU will inject a single correctable error on the lowest ECC bit on the next read packet. If the Error Count Register is already zero, nothing happens. Otherwise, the error count will be decrement by one. If the MCU Error Count Register goes to zero, then the Correctable ECC Error Interrupt will be generated. This ECC error should also be reported to the L2 regardless of the value of the Error Count Register.

Recoverable FBD Channel Error Count Interrupt

A Recoverable FBD Channel Error Interrupt will be generated whenever the Recoverable FBD Channel Error Count Regiser reaches zero. The MCU will assert mcu_ncu_fbr to the NCU for one iol2clk cycle. Once the count value is zero, no more interrupts of this type will be generated until software writes a non-zero value to the count register.

If ncu_mcu_fbri is asserted from the NCU to the MCU, the MCU will inject an error within the FBD channel. The source of the error will be determined by the Injected Error Source Register. The MCU will handle the error as if it had actually occurred in hardware. If the Recoverable FBD Channel Error Count Regiser reaches zero, then mcu_ncu_fbr will be asserted to the NCU.

Unrecoverable FBD Channel Error Interrupt Injection

Thee is no error generated to the NCU for Unrecoverable FBD Channel Errors. These types of errors will only be indicated through the L2 cache.

If ncu_mcu_fbui is asserted from the NCU to the MCU, the MCU will inject an error within the FBD channel. The source of the error will be determined by the Injected Error Source Register. The MCU will handle the error as if it had actually occurred in hardware.

3.11 Test Features

3.11.1 DFT Features

3.11.1.1 Debug Reset

During Debug Reset, the MCU needs to keep the FBD links active so that runs can be reproducible. By keeping them active, the channel latencies and the SERDES to MCU asynchronous crossings will not change between test runs.

In order to achieve this, a small part of the MCU must be protected from warm reset, and the clock to it needs to remain active during reset. For each MCU, there is an additional clock stop signal, tcu_mcu*_fbd_clk_stop, and a common test mode signal, tcu_mcu_testmode. The subblock mcu_fdout_ctl contains the logic that remains active during the Debug Reset. Its clock is taken directly from dr_gclk, not from the output of a cluster header. The tcu_mcu*_fbd_clk_stop signal goes directly to the L1 headers in mcu_fdout_ctl, and these clock stop Signals are not asserted during the Debug Reset. The tcu_mcu_testmode signal is used to qualify the scan and reset Signals. tcu_scan_en, tcu_aclk and tcu_bclk are ANDed with tcu_mcu_testmode in mcu_fdout_ctl. tcu_mcu_testmode is asserted for Power-On Reset and for a normal Warm Reset, but not for Debug Reset. tcu_mcu_testmode is also used to control a mux which bypasses the protected flops on the scan chain when it is not asserted but inclues them when not asserted.

3.11.2 Deterministic Test Mode (DTM)

Deterministic Test Mode is a mode in which all of the IO units operate at the same speed so that test runs may be reproducible on a tester. The mode is entered when the ccu_serdes_dtm is set to 1'b1. In this mode different data will be sent by the MCU to the debug bus. Also, since the tester will be "sourcing" transactions, the MCU will act as a slave device during channel initialization and must respond to transactions on the NB channel in order to enter a predictable state.

3.11.2.1 Debug Signals

For DTM, the MCU will provide CRC data from the southbound FBD frames to the Debug unit. Each frame contains 22 bits of CRC, so since there are two channels per MCU, there will be 44 bits of CRC data per MCU. The MCU will take the CRC from the two channels and XOR them; however, either channel's CRC can be masked off before the XOR by setting the appropriate bit in the Debug Trigger Enable Register. This data will then be sent to the UCB module to be multiplexed with the normal-mode debug Signals. One additional bit needs to be sent to the Debug unit since the current debug bus signal count is 21 bits. The MCU will use the ccu_serdes_dtm signal to select the CRC data for sending to the Debug unit.

These debug Signals will be taken from the drl2clk domain to the iol2clk domain; however, they will not need to be synchronized since they will only be used for DTM where the drl2clk and iol2clk will be driven from the same source.

3.11.2.2 Initialization for Testing

When in DTM mode, the northbound FBD channel still needs to be initalized by northbound TS0 patterns. Therefore, the tester must send enough TS0 patterns to achieve bit lock, frame lock and lane deskew within the MCU. After these are achieved, the tester must then cause the MCU to transition to the L0 state in order to enable southbound transactions and to enable recognition of northbound transactions.

After reset, the MCU's FBD initialilzation state machine will be in the Disable state, and the DTM state machine will begin in the IDLE state. The MCU's initialization state machine must be in the Disable state before DTM testing begins. The tester will send TS0 patterns to train the northbound link. The DTM state machine will transition to the TS0 state once it sees the TS0 patterns on the NB channel. This will also cause the MCU state machine to transition to the TS0 state, and the MCU will begin sending southbound TS0 patterns. Once in the TS0 state, the DTM state machine will wait until it sees at least 4 northbound frames containing all 0's. At this point, the MCU's initalization state machine will be transitioned to the L0 state, and the DTM state machine will return to IDLE. Once the MCU enters the L0 state, operation will proceed as in normal system mode.

Since the Polling state will be bypassed during the initialization for DTM, the channel latency register must be programmed to match the channel latency used for generating the test vectors.

In order to achieve operating rates that the tester can support, the RXTX_RATE field of the SERDES Configuration Bus Register must be set to Half Rate (2'b01) or Quarter Rate (2'b10) as required. The required link rates for supported operating frequencies are given in the DTM section of the CCU spec.

3.11.3 SERDES Blunt-End Loopback

SERDES Blunt-End Loopback within the MCU is controlled by the Loopback Mode Control Register. This register controls both links within an MCU. When bit 1 is set, data received on a northbound FBD channel will be placed on the corresponding southbound channel. Since there are 14 northbound lanes and only 10 southbound lanes, bit 0 selects which northbound lanes map to which southbound lanes. If bit 0 is 1'b0, then northbound lanes 0 through 9 are mapped to the southbound lanes. If bit 0 is 1'b1, then northbound lanes 4 through 13 are mapped to the southbound lanes.

Since the data on the northbound channel must be synchronized from the recovered clock domain into the MCU's drl2clk domain, TS0 patterns must be sent on the northbound channel in order to achieve frame lock. Once frame lock is achieved, the cross domain FIFO will be enabled, and data will be forwarded from the northbound channel to the southbound channel.

This increase in size would change the MCU's Y dimension to about 1750 µm due to the addition of the FBDWR_DP and FBDRD_DP. (The size of the MCU as of June 10, 2004 is 1744µm x 847µm.) The X dimension would have to be reduced and the aspect ratios of the control blocks would have to change in order to use up any whitespace created.

A break down of the area changes per block is given in the following sections.

3.12 MCU Level I/O

TABLE 3-22 MCU Level I/O

Signal Name	I/O	Description
Clocks, Reset, Etc.		
iol2clk	I	System bus clock
drl2clk	I	DRAM clock
l2clk	I	CPU domain clock
mcu_ce	I	DRAM module clock enable
ccu_mcu_ddr_cmp_sync_en	I	Dram to Cmp clock synchronization
ccu_mcu_cmp_ddr_sync_en	I	Cmp to Dram clock synchronization
ccu_mcu_io_cmp_sync_en	I	Sys to Cmp clock synchronization
ccu_mcu_cmp_io_sync_en	I	Cmp to Sys clock synchronization
clspine_mcu_selfrsh	I	Enter hardware self-refresh mode
rst_por_	I	Power-on reset signal
rst_wmr_	I	Warm reset signal
mcu_pt_sync_in0 mcu_pt_sync_in1 mcu_pt_sync_in2	I	Incoming power throttling counter synchronizing Signals
mcu_pt_sync_out	O	Outgoing power throttling counter synchronizing signal
mcu_id[1:0]	I	MCU ID for error reporting
mcu_clk_en	I	Clock enable to synchronize MCU clock domain to external DRAM clock
Test		
scan_in	I	Scan in
scan_out	O	Scan out
tcu_aclk	I	
tcu_bclk	I	
tcu_soc_cmp_clk_stop	I	Clock stop signal for l2clk domain
tcu_soc5dr_clk_stop	I	Clock stop signal for drl2clk domain
tcu_soc6io_clk_stop	I	Clock stop signal for iol2clk domain
tcu_pce_ov	I	Clock enable override signal

TABLE 3-22 MCU Level I/O *(Continued)*

Signal Name	I/O	Description
tcu_dectest	I	
tcu_scan_en	I	Scan enable
tcu_se_scancollar_in	I	Scan enable for memory input flops
tcu_se_scancollar_out	I	Scan enable for memory output flops
tcu_array_wr_inhibit	I	Inhibit memory array updates
tcu_array_bypass	I	Bypass memory array
tcu_mbist_bisi_en	I	Enable MBIST engine
tcu_mcu_mbist_start	I	Start MBIST sequence
mcu_tcu_mbist_done	O	MBIST done
mcu_tcu_mbist_fail	O	MBIST fail
tcu_mcu_mbist_scan_in	I	MBIST module scan in
mcu_tcu_mbist_scan_out	O	MBIST module scan
Debug		
mcu_dbg1_rd_req_in_0[3:0]	O	Read request received from L2 bank 0
mcu_dbg1_rd_req_in_1[3:0]	O	Read request received from L2 bank 1
mcu_dbg1_rd_req_out[4:0]	O	Read data returned to L2 bank
mcu_dbg1_wr_req_in_0	O	Write request received from L2 bank 0
mcu_dbg1_wr_req_in_1	O	Write request received from L2 bank 1
mcu_dbg1_wr_req_out[1:0]	O	Number of writes retired
mcu_dbg1_mecc_err	O	Multiple nibble ECC error
mcu_dbg1_secc_err	O	Single nibble ECC error
mcu_dbg1_fbd_err	O	FBD channel error
mcu_dbg1_err_mode	O	MCU in error processing mode
mcu_dbg1_err_event	O	Debug error event when debug trigger is enabled
NCU Interface		
ncu_mcu_data[3:0]	I	NCU to MCU module CSR bus
ncu_mcu_stall	I	Stall signal from NCU for outgoing transactions
ncu_mcu_vld	I	Incoming CSR data valid
ncu_mcu_ecci	I	Inject Correctable Error Count
ncu_mcu_fbri	I	Inject FBDIMM Channel Recoverable Error

TABLE 3-22 MCU Level I/O *(Continued)*

Signal Name	I/O	Description
ncu_mcu_fbui	I	Inject FBDIMM Channel Unrecoverable Error
mcu_ncu_data[3:0]	O	MCU module to NCU CSR bus
mcu_ncu_stall	O	Stall incoming transactions
mcu_ncu_vld	O	Outgoing CSR data valid
mcu_ncu_ecc	O	Correctable Error Count Interrupt
mcu_ncu_fbr	O	FBDIMM Channel Recoverable Error Interrupt
ncu_mcu_pm	I	Enables partial-bank mode
ncu_mcu_ba01	I	L2 banks 0 and 1 are enabled in partial-bank mode
ncu_mcu_ba23	I	L2 banks 2 and 3 are enabled in partial-bank mode
ncu_mcu_ba45	I	L2 banks 4 and 5 are enabled in partial-bank mode
ncu_mcu_ba67	I	L2 banks 6 and 7 are enabled in partial-bank mode
MCU-L2 Interface		
l2b0_mcu_data_mecc_r5 l2b1_mcu_data_mecc_r5	I	Signal to inject ecc errors in write data
l2b0_mcu_data_vld_r5 l2b1_mcu_data_vld_r5	I	Data valid signal from L2 cache
l2b0_mcu_wr_data_r5[63:0] l2b1_mcu_wr_data_r5[63:0]	I	Data from L2 cache
l2t0_mcu_addr_39to9[39:7] l2t0_mcu_addr_5 l2t1_mcu_addr_39to9[39:7] l2t1_mcu_addr_5	I	L2 cache transaction address
l2t0_mcu_rd_dummy_req l2t1_mcu_rd_dummy_req	I	Dummy read request from L2
l2t0_mcu_rd_req l2t1_mcu_rd_req	I	L2 cache read request signal
l2t0_mcu_rd_req_id[2:0] l2t1_mcu_rd_req_id[2:0]	I	L2 cache read request ID
l2t0_mcu_wr_req l2t1_mcu_wr_req	I	L2 cache write request signal
mcu_l2t0_data_vld_r0 mcu_l2t1_data_vld_r0	O	L2 cache read data valid signal
mcu_l2t0_rd_ack mcu_l2t1_rd_ack	O	Read request acknowledge signal to L2 cache
mcu_l2t0_scb_mecc_err mcu_l2t1_scb_mecc_err	O	MCU scrub multiple ECC error indication to L2 cache

TABLE 3-22 MCU Level I/O *(Continued)*

Signal Name	I/O	Description
mcu_l2t0_scb_secc_err mcu_l2t1_scb_secc_err	O	MCU scrub single ECC error indication to L2 cache
mcu_l2t0_wr_ack mcu_l2t1_wr_ack	O	Write request acknowledge signal to L2 cache
mcu_l2t0_wr_addr_err mcu_l2t1_wr_addr_err	O	Write address error signal to L2 cache
mcu_l2t0_qword_id[1:0] mcu_l2t1_qword_id[1:0]	O	Quadword of data being returned from MCU
mcu_l2t0_mecc_err_r3 mcu_l2t1_mecc_err_r3	O	MCU multiple ECC error indication to L2 cache
mcu_l2t0_rd_req_id_r0[2:0] mcu_l2t1_rd_req_id_r0[2:0]	O	Read request ID for L2 cache read data
mcu_l2t0_secc_err_r3 mcu_l2t1_secc_err_r3	O	DRAM single ECC error indication to L2 cache
mcu_l2b_data_r3[127:0]	O	Read data to L2 cache
mcu_l2b_ecc_r3[27:0]	O	ECC data for read data to L2 cache
MCU-FBD IO Interface		
mcu_fsr0_data[119:0] mcu_fsr1_data[119:0]	O	Southbound FBD Channel Data
mcu_fsr0_cfgpll_enpll mcu_fsr1_cfgpll_enpll	O	Enable PLLs for FBD Channels
mcu_fsr01_cfgpll_lb[1:0]	O	PLL Loopback for Channels 0 and 1
mcu_fsr01_cfgpll_mpy[3:0]	O	PLL Multiplier for Channels 0 and 1
mcu_fsr0_cfgrx_enrx mcu_fsr1_cfgrx_enrx	O	Enable SERDES receivers
mcu_fsr0_cfgrx_align mcu_fsr1_cfgrx_align	O	Enable Alignment detection for FBD SERDES
mcu_fsr0_cfgrx_los[1:0] mcu_fsr1_cfgrx_los[1:0]	O	Enable Loss-of-Signal (Electrical Idle) detection
mcu_fsr0_cfgrx_invpair[13:0] mcu_fsr0_cfgrx_invpair[13:0]	O	Invert RXP and RXN per bit
mcu_fsr01_cfgrx_eq[3:0]	O	Enable and configure adaptive equalizer
mcu_fsr01_cfgrx_cdr[2:0]	O	Configure clock/data recovery algorithm
mcu_fsr01_cfgrx_term[2:0]	O	Set input termination
mcu_fsr0_cfgtx_entx mcu_fsr1_cfgtx_entx	O	Enable SERDES transmitters

TABLE 3-22 MCU Level I/O *(Continued)*

Signal Name	I/O	Description
mcu_fsr0_cfgtx_enidl mcu_fsr1_cfgtx_enidl	O	Enable Electrical Idle on Transmitter
mcu_fsr0_cfgtx_invpair[9:0] mcu_fsr1_cfgtx_invpair[9:0]	O	Invert TXP and TXN per bit
mcu_fsr01_cfgtx_enftp	O	Enable fixed phose on TXBCLKIN
mcu_fsr01_cfgtx_de[3:0]	O	Set transmitter output de-emphasis
mcu_fsr01_cfgtx_swing[2:0]	O	Set transmitter output swing
mcu_fsr01_cfgtx_cm	O	Adjust common mode.
fsr0_mcu_rxbclk[13:0] fsr1_mcu_rxbclk[13:0]	I	Clocks for Northbound FBD Channels
fsr0_mcu_data[167:0] fsr1_mcu_data[167:0]	I	Northbound FBD Channel Data
fsr0_mcu_stspll_lock fsr1_mcu_stspll_lock	I	SERDES PLLs are locked
fsr0_mcu_stsrx_sync[13:0] fsr1_mcu_stsrx_sync[13:0]	I	Header alignment signal from SERDES Receivers
fsr0_mcu_stsrx_losdtct[13:0] fsr1_mcu_stsrx_losdtct[13:0]	I	Electrical Idle signal from SERDES Receivers

3.13 MCU Registers

MCU register definitions are detailed in the OpenSPARC T2 Programmer's Reference Manual. This document will only provide a list of registers and a cross reference to their locations in the *OpenSPARC T2 Programmer's Reference Manual.*

3.13.1 Control and Status Registers

TABLE 3-23 Control and Status Registers

Register Offset	Register Name	OpenSPARC T2 PRM Section
0x00000000	CAS Address Width Register	25.10.1
0x00000008	RAS Address Width Register	25.10.2
0x00000010	CAS Latency Register	25.10.3
0x00000018	Scrub Frequency Register	25.10.4
0x00000020	Refresh Frequency Register	25.10.5
0x00000038	Refresh Counter Register	25.10.6
0x00000040	Scrub Enable Register	25.10.7
0x00000080	RAS to RAS Different Bank Delay Register	25.10.8
0x00000088	RAS to RAS Same Bank Delay Register	25.10.9
0x00000090	RAS to CAS Delay Register	25.10.10
0x00000098	Write to Read CAS Delay Register	25.10.11
0x000000A0	Read to Write CAS Delay Register	25.10.12
0x000000A8	Internal Read to Precharge Delay Register	25.10.13
0x000000B0	Active to Precharge Delay Register	25.10.14
0x000000B8	Precharge Command Period Register	25.10.15
0x000000C0	Write Recovery Period Register	25.10.16
0x000000C8	Autorefresh to Active Period Register	25.10.17
0x000000D0	Mode Register Set Command Period Register	25.10.18
0x000000E0	Internal Write to Read Command Delay Register	25.10.19
0x000000E8	Precharge Wait Register During Power Up	25.10.20
0x00000108	DIMM Stacked Register	25.10.21
0x00000110	Extended Mode 2 Register	25.10.22
0x00000118	Extended Mode 1 Register	25.10.23
0x00000120	Extended Mode 3 Register	25.10.24
0x00000128	8 Bank Mode Register	25.10.25
0x00000138	Branch Disabled Register	25.10.27
0x00000140	Select Low Order Address Bits Register	25.10.28
0x000001A0	DIMM Initialization Register	25.10.29

TABLE 3-23 Control and Status Registers *(Continued)*

Register Offset	Register Name	OpenSPARC T2 PRM Section
0x00000208	Mode Register Write Status Register	25.10.33
0x00000210	Initialization Status Register	25.10.34
0x00000218	DIMMs Present Register	25.10.35
0x00000220	Fail-Over Status Register	25.10.36
0x00000228	Fail-Over Mask Register	25.10.37

3.13.1.1 Changes to DIMM Initialization Register- 0x84_0000_01A0

The DIMM Initialization Register for OpenSPARC T2 has the following format:

TABLE 3-24 DRAM Initialization Register

Field	Bit position	Initial value	R/W	Description
RSVD	[62:2]	0x0	RO	Reserved
CKE	[1]	0x0	RW	Enabled CKE to DIMMs
INIT	[0]	0x1	RW	Set to 1 during software initialization of DRAMs; cleared by software when done.

3.13.1.2 Single Channel Mode Regiser - 0x84_0000_0148

TABLE 3-25 Single Channel Mode Register

Field	Bit position	Initial value	R/W	Description
RSVD	[62:1]	0x0	RO	Reserved
MODE	[0]	0x0	RW	Enable use of 1 FBD channel for memory transactions. Burst length becomes 8.

3.13.1.3 Four Activate Window Register

TABLE 3-26 Four Activate Window Register

Field	Bit position	Initial value	R/W	Description
RSVD	[62:5]	0x0	RO	Reserved
MODE	[4:0]	0xA	RW	tFAW. Number of cycles in which 4 Activate commands may be issued to a DIMM. Preserved on warm reset.

3.13.2 Error Registers

TABLE 3-27 Error Registers

Register Offset	Register Name	OpenSPARC T2 PRM Section
0x00000280	Error Status Register	12.12.1
0x00000288	Error Address Register	12.12.2
0x00000290	Error Injection Register	12.12.3
0x00000298	Error Counter Register	12.12.4
0x000002A0	Error Location Register	12.12.5

3.13.2.1 Changes to Error Status Regiser - 0x84_0000_0280

The MCU Error Status Register has 3 additional bits:

TABLE 3-28 MCU Error StatusRegister

Field	Bit Position	Initial Value	R/W	Description
MEB	[56]	0x0	R/W1C	Multiple Out-of-Bound Errors
FBU	[55]	0x0	R/W1C	FBDIMM Channel Unrecoverable Error
FBR	[54]	0x0	R/W1C	FBDIMM Channel Recoverable Error

3.13.2.2 Error Retry Register - 0x84_0000_02a8

TABLE 3-29 Error Entry Register

Field	Bit Position	Initial Value	R/W	Description
VALID	[63]	0x0	RW	Error Retry Register is valid
RSVD	[62:50]	0x0	RO	Reserved
SYNDROME2	[49:34]	0x0	RW	Syndrome from second retry read
TYPE2	[33:32]	0x0	RW	Result of second retry read
RSVD	[31:18]	0x0	RO	Reserved
SYNDROME1	[17:2]	0x0	RW	Syndrome from second retry read
TYPE1	[1:0]	0x0	RW	Result of second retry read: 00: No read 01: No error 10: Correctable Error 11: Uncorrectable Error

3.13.3 Power Management Registers

TABLE 3-30 Power Management Registers

Register Offset	Register Name	OpenSPARC T2 PRM Section
0x00000028	Open Bank Max Register	26.3.1
0x00000048	Programmable Time Counter Register	26.3.2

3.13.3.1 Power Down Mode Register - 0x84_0000_0238

This register enables the use of Power Down mode for power savings. When enabled, an FBD will be placed in Power Down mode when there are no pending or outstanding transactions to that FBD.

TABLE 3-31 Power Down Mode Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:1]	0x0	RO	Reserved
ENABLE	[0]	0x0	RW	Enable use of Power Down mode if 1.

3.13.4 Performance Registers

TABLE 3-32 Performance Registers

Register Offset	Register Name	OpenSPARC T2 PRM Section
0x00000400	Performance Control Register	10.3
0x00000408	Performance Counter Register	10.3

3.13.5 Changes to Debug Trigger Enable Register

TABLE 3-33 Debug Trigger Enable Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:6]	0x0	RO	Reserved
DTM_ATSPEED	[5]	0x0	RW	If set, Debug bus sends normal mode data in DTM mode
DTM_MASK1	[4]	0x0	RW	If set, mask off CRC data from Channel 1 going to Debug bus
DTM_MASK0	[3]	0x0	RW	If set, mask off CRC data from Channel 0 going to Debug bus
DBG_EN	[2]	0x0	RW	Enable error events to Debug unit
MASK_ERR	[1]	0x0	RW	Mask LFSR related errors on NB FBD links
KP_LNK_UP	[0]	0x0	RW	Keep FBD links up during Warm Reset

3.13.6 State Registers for FBD Branch

FBD controller register address space will be a subset of the MCU's address space, differentiated by some upper bits.

3.13.6.1 Channel State Register - 0x84_0000_0800

TABLE 3-34 Channel State Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:8]	0x0	RO	Reserved
MDISABLE	[7]	0x0	RW	Disable AMB data merging for TS2 patterns
AMBID	[6:3]	0x0	RW	Target AMB for training sequences
STATE	[2:0]	0x0	RW	State in initialization sequence 0x0 = Disable, 0x1 = Calibrate, 0x2 = Training, 0x3 = Testing, 0x4 = Polling, 0x5 = Config, 0x6 = L0

3.13.6.2 Fast Reset Flag - 0x84_0000_0808

TABLE 3-35 Fast Reset Flag

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:4]	0x0	RO	Reserved
SYNC_IER	[3]	0x0	RW	Enables use of IER bit in Sync command. IER will be issued in last Sync frame before a Channel Reset.
SYNC_R	[2:1]	0x0	RW	Indicates which status register will be received from AMBs
FASTRESET	[0]	0x0	RW	Causes MCU to enter use Fast Reset sequence for channel initialization

3.13.6.3 Channel Reset (Initialization) Flag - 0x84_0000_0810

TABLE 3-36 Channel Reset (Initialization) Flag

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
FBDINITERR	[1]	0x0	RW	Set to 1 if an error occurred during the FBD Channel initialization.
FBDINIT	[0]	0x0	RW	Causes FBD Channel to be initialized. Uses fast reset sequence if Fast Reset Flag is set, otherwise performs full initialization (including Calibration). Reset to 0 when initialization is complete.

3.13.6.4 TS1 Southbound to Northbound Mapping Register - 0x84_0000_0818

TABLE 3-37 TS1 Southbound to Northbound Mapping Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:4]	0x0	RO	Reserved
IBRX_CHNL	[3]	0x0	RW	Selects which NB channel will be checked by the IBIST Receive engine
MAPPING	[2:0]	0x0	RW	Determines how targetted AMB maps data from SB bit lanes to NB bit lanes

3.13.6.5 TS1 Test Paramater Register - 0x84_0000_0820

TABLE 3-38 TS1 Test Paramater Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:24]	0x0	RO	Reserved
PARAM	[23:0]	0x0	RW	AMB test parameters for TS1 sequence

3.13.6.6 TS3 Failover Configuration Register - 0x84_0000_0828

TABLE 3-39 TS3 Failover Configuration Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:16]	0x0	RO	Reserved
SBCONFIG1	[15:12]	0xF	RW	Indicates which southbound lanes for channel 1 will be used.
NBCONFIG1	[11:8]	0xF	RW	Indicates which northbound lanes for channel 1 will be used.
SBCONFIG0	[7:4]	0xF	RW	Indicates which southbound lanes for channel 0 will be used.
NBCONFIG0	[3:0]	0xF	RW	Indicates which northbound lanes for channel 0 will be used.

3.13.6.7 Electical Idle Detected Register - 0x84_0000_0830

TABLE 3-40 Electrical Idle Detected Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:28]	0x0	RO	Reserved
ELECTIDLE1	[27:14]	0x3ff	RO	Electrical Idle detected from bit lanes
ELECTIDLE0	[13:0]	0x3ff	RO	Electrical Idle detected from bit lanes

3.13.6.8 Disable State Period Register - 0x84_0000_0838

TABLE 3-41 Disable State Period Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:10]	0x0	RO	Reserved
COUNT	[9:0]	0xFF	RW	Counter value for Disable state. Once Disable state is entered, a counter will count to this value. Once it reaches it, the Disable_Done bit in the Disable State Period Done Register will be set.

3.13.6.9 Disable State Period Done Register - 0x84_0000_0840

TABLE 3-42 Disable State Period Done Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:1]	0x0	RO	Reserved
DONE	[0]	0x0	RW	Indicates that counter for Disable state period has completed counting.

3.13.6.10 Calibrate State Period Register - 0x84_0000_0848

TABLE 3-43 Calibrate State Period Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:20]	0x0	RO	Reserved
COUNT	[19:0]	0x0	RW	Counter value for Calibrate state. Once Calibrate state is entered, a counter will count to this value. Once it reaches it, the Calibrate_Done bit in the FBD Status Register will be set.

3.13.6.11 Calibrate State Period Done Register - 0x84_0000_0850

TABLE 3-44 Calibrate State Period Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:1]	0x0	RO	Reserved
DONE	[0]	0x0	RW	Indicates that counter for Disable state period has completed counting.

3.13.6.12 Training State Minimum Time Register - 0x84_0000_0858

TABLE 3-45 Training State Minimum Time Registers

Field	Bit position	Initial value	R/W	Description
RSVD	[63:16]	0x0	RO	Reserved
COUNT	[15:0]	0xFF	RW	Minimum number of frames for Training state before starting to check for Done.

3.13.6.13 Training State Done Register - 0x84_0000_0860

TABLE 3-46 Training State Done Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
TIMEOUT	[1]	0x0	RW	Set when timeout period has elapsed before Done has been asserted.
DONE	[0]	0x0	RW	Set when Training state has completed.

3.13.6.14 Training State Timeout Register - 0x84_0000_0868

TABLE 3-47 Training State Timeout Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:8]	0x0	RO	Reserved
PERIOD	[7:0]	0xFF	RW	Number of frames for Training state to complete after minimum number of frames have elapsed.

3.13.6.15 Testing State Done Register - 0x84_0000_0870

TABLE 3-48 Testing State Done Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
TIMEOUT	[1]	0x0	RW	Set when timeout period has elapsed before Done has been asserted.
DONE	[0]	0x0	RW	Set when Testing state has completed.

3.13.6.16 Testing State Timeout Register - 0x84_0000_0878

TABLE 3-49 Testing State Timeout Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:8]	0x0	RO	Reserved
PERIOD	[7:0]	0xFF	RW	Number of frames for Testing state to complete.

3.13.6.17 Polling State Done Register - 0x84_0000_0880

TABLE 3-50 Polling State Done Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
TIMEOUT	[1]	0x0	RW	Set when timeout period has elapsed before Done has been asserted.
DONE	[0]	0x0	RW	Set when Polling state has completed.

3.13.6.18 Polling State Timeout Register - 0x84_0000_0888

TABLE 3-51 Polling State Timeout Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:8]	0x0	RO	Reserved
PERIOD	[7:0]	0xFF	RW	Number of frames for Polling state to complete.

3.13.6.19 Config State Done Register - 0x84_0000_0890

TABLE 3-52 Config State Done Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
TIMEOUT	[1]	0x0	RW	Set when timeout period has elapsed before Done has been asserted.
DONE	[0]	0x0	RW	Set when Config state has completed.

3.13.6.20 Config State Timeout Period Register - 0x84_0000_0898

TABLE 3-53 Config State Timeout Period Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:8]	0x0	RO	Reserved
PERIOD	[7:0]	0xFF	RW	Number of frames for Config state to complete.

3.13.6.21 Per Rank CKE Register - 0x84_0000_08A0

Writing this register or the CKE bit of register 0x84_0000_01A0 sends a CKE command to the FBDIMMs. Each bit corresponds to a rank in a fully populated FBDIMM system. Bit 0 is for DIMM0, rank0; bit 1 is for DIMM0, rank1; etc. The enable bits are qualified by the number of DIMMs and whether they are stacked

before the CKE command is issued to the DIMMs.

TABLE 3-54 Per Rank CKE Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:16]	0x0	RO	Reserved
D7R1	[15]	0x1	RW	CKE enable for DIMM 7 Rank 1
D7R0	[14]	0x1	RW	CKE enable for DIMM 7 Rank 0
D6R1	[13]	0x1	RW	CKE enable for DIMM 6 Rank 1
D6R0	[12]	0x1	RW	CKE enable for DIMM 6 Rank 0
D5R1	[11]	0x1	RW	CKE enable for DIMM 5 Rank 1
D5R0	[10]	0x1	RW	CKE enable for DIMM 5 Rank 0
D4R1	[9]	0x1	RW	CKE enable for DIMM 4 Rank 1
D4R0	[8]	0x1	RW	CKE enable for DIMM 4 Rank 0
D3R1	[7]	0x1	RW	CKE enable for DIMM 3 Rank 1
D3R0	[6]	0x1	RW	CKE enable for DIMM 3 Rank 0
D2R1	[5]	0x1	RW	CKE enable for DIMM 2 Rank 1
D2R0	[4]	0x1	RW	CKE enable for DIMM 2 Rank 0
D1R1	[3]	0x1	RW	CKE enable for DIMM 1 Rank 1
D1R0	[2]	0x1	RW	CKE enable for DIMM 1 Rank 0
D0R1	[1]	0x1	RW	CKE enable for DIMM 0 Rank 1
D0R0	[0]	0x1	RW	CKE enable for DIMM 0 Rank 0

3.13.6.22 L0s Duration - 0x84_0000_08A8

TABLE 3-55 L0s Duration

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:7]	0x0	RO	Reserved
ENABLE	[6]	0x0	RW	Enables use of L0s mode when MCU is idle.
COUNT	[5:0]	0x2A	RW	Determines the number of frames that the branch will be in the L0s state. Legal values are 0x20 to 0x2A. Values below 0x20 will be treated as 0x20 and values above 0x2A will be treated as 0x2A.

3.13.6.23 Sync Frame Frequency Register - 0x84_0000_08B0

TABLE 3-56 Sync Frame Frequency Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:6]	0x0	RO	Reserved
FREQ	[5:0]	0x2A	RW	Frequency at which sync frames are issued on the FBDIMM channels

3.13.6.24 Channel Read Latency Register - 0x84_0000_08B8

TABLE 3-57 Channel Read Latency Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:16]	0x0	RO	Reserved
LATENCY1	[15:8]	0xFF	RW	Read latency for channel 1. Determined during Polling state.
LATENCY0	[7:0]	0xFF	RW	Read latency for channel 0. Determined during Polling state.

3.13.6.25 Channel Capability Register - 0x84_0000_08C0

TABLE 3-58 Channel Capability Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:10]	0x0	RO	Reserved
CAPABIL1	[9:5]	0x0	RO	Channel capabilities for selected AMB in channel 1. Only valid during Polling state.
CAPABIL0	[4:0]	0x0	RO	Channel capabilities for selected AMB in channel 0. Only valid during Polling state.

3.13.6.26 Loopback Mode Control Register - 0x84_0000_08C8

TABLE 3-59 Loopback Mode Control Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
MODE	[1:0]	0x0	RW	Loopback Mode: 0x: Loopback Mode disabled 10: Place low-order NB data on SB bus 11: Place high-order NB data on SB bus

3.13.6.27 SERDES Configuration Bus Register - 0x84_0000_08D0

TABLE 3-60 SERDES Configuration Bus Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:25]	0x0	RO	Reserved
RXTX_RATE	[29:28]	0x0	RW	Receiver/Transmitter Operating Rate
TX_CM	[27]	0x0	RW	Transmitter Common Mode
TX_SWING	[26:24]	0x1	RW	Transmitter Output Swing
TX_DE	[23:20]	0x0	RW	Transmitter De-emphasis
TX_ENFTP	[19]	0x0	RW	Tranmitter Enable
RX_TERM	[18:16]	0x0	RW	Receiver Termination
RSVD	[15]	0x0	RO	Reserved
RX_CDR	[14:12]	0x0	RW	Receiver Clock/Data Recovery Algorithm
RX_EQ	[11:8]	0x0	RW	Receiver Adaptive Equalizer Configuration
RSVD	[7:6]	0x0	RO	Reserved
PLL_MPY	[5:2]	0x0	RW	PLL Multiplier
PLL_LB	[1:0]	0x0	RW	Loop bandwidth

3.13.6.28 SERDES Tranmitter and Receiver Differential Pair Inversion Register - 0x84_0000_08D8

TABLE 3-61 SERDES Transmitter and Receiver Differential Pair Inversion Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:48]	0x0	RO	Reserved
TX1_INVPAIR	[47:38]	0x0	RW	Invert Channel 1 TXPi/TXNi if bit is 1.
TX0_INVPAIR	[37:28]	0x0	RW	Invert Channel 0 TXPi/TXNi if bit is 1.
RX1_INVPAIR	[27:14]	0x0	RW	Invert Channel 1 RXPi/RXNi if bit is 1.
RX0_INVPAIR	[13:0]	0x0	RW	Invert Channel 0 RXPi/RXNi if bit is 1.

3.13.6.29 SERDES Test Configuration Bus Register - 0x84_0000_08E0

TABLE 3-62 SERDES Test Configuration Bus Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:32]	0x0	RO	Reserved
FSR1_TX_ENTEST	[31]	0x0	RW	Enable testing of FSR1 Transmit ports
FSR0_TX_ENTEST	[30]	0x0	RW	Enable testing of FSR0 Transmit ports
FSR1_RX_ENTEST	[29]	0x0	RW	Enable testing of FSR1 Receive ports
FSR0_RX_ENTEST	[28]	0x0	RW	Enable testing of FSR0 Receive ports
FSR1_INVPATT	[27]	0x0	RW	FSR1 Invert Polarity
FSR1_RATE	[26:25]	0x0	RW	FSR1 Operating Rate
FSR1_ENBSPLS	[24]	0x0	RW	FSR1 Receiver pulse boundary scan
FSR1_ENBSRX	[23]	0x0	RW	FSR1 Receiver boundary scan
FSR1_ENBSTX	[22]	0x0	RW	FSR1 Transmitter boundary scan
FSR1_LOOPBACK	[21:20]	0x0	RW	FSR1 Loopback
FSR1_CLKBYP	[19:18]	0x0	RW	FSR1 Clock bypass
FSR1_ENRXPATT	[17]	0x0	RW	FSR1 Enable Rx patterns
FSR1_ENTXPATT	[16]	0x0	RW	FSR1 Enable Tx patterns
FSR1_TESTPATT	[15:14]	0x0	RW	FSR1 Test pattern
FSR0_INVPATT	[13]	0x0	RW	FSR0 Invert Polarity
FSR0_RATE	[12:11]	0x0	RW	FSR0 Operating Rate
FSR0_ENBSPLS	[10]	0x0	RW	FSR0 Receiver pulse boundary scan
FSR0_ENBSRX	[9]	0x0	RW	FSR0 Receiver boundary scan
FSR0_ENBSTX	[8]	0x0	RW	FSR0 Transmitter boundary scan
FSR0_LOOPBACK	[7:6]	0x0	RW	FSR0 Loopback
FSR0_CLKBYP	[5:4]	0x0	RW	FSR0 Clock bypass
FSR0_ENRXPATT	[3]	0x0	RW	FSR0 Enable Rx patterns
FSR0_ENRXPATT	[2]	0x0	RW	FSR0 Enable Tx patterns
FSR0_TESTPATT	[1:0]	0x3	RW	FSR0 Test pattern

3.13.6.30 SERDES PLL Status Register - 0x84_0000_08E8

TABLE 3-63 SERDES PLL Status Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:6]	0x0	RO	Reserved
FSR1_STSPLL	[5:3]	0x0	RO	PLL Lock Status for FSR1 macros
FSR0_STSPLL	[2:0]	0x0	RO	PLL Lock Status for FSR0 macros

3.13.6.31 SERDES Test Status Register - 0x84_0000_08F0

TABLE 3-64 SERDES Test Status Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:48]	0x0	RO	Reserved
FSR1_TX_TESTFAIL	[47:38]	0x0	RO	Test Status for FSR1 Transmit ports
FSR0_TX_TESTFAIL	[37:28]	0x0	RO	Test Status for FSR0 Transmit ports
FSR1_RX_TESTFAIL	[27:14]	0x0	RO	Test Status for FSR1 Receive ports
FSR0_RX_TESTFAIL	[13:0]	0x0	RO	Test Status for FSR0 Receive ports

3.13.6.32 Configuration Register Access Address Register - 0x84_0000_0900

TABLE 3-65 Configuration Register Access Address Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:16]	0x0	RO	Reserved
CHANNEL	[15]	0x0	RW	Channel of Configuration Register Access.
AMB	[14:11]	0x0	RW	AMB ID of Configuration Register Access.
DATA	[10:2]	0x0	RW	Address for Configuration Register read or write.
RSVD	[1:0]	0x0	RO	Reserved

3.13.6.33 Configuration Register Access Data Register - 0x84_0000_0908

TABLE 3-66 Configuration Register Access Data Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:32]	0x0	RO	Reserved
DATA	[31:0]	0x0	RW	Data for Configuration Register read or write. Writing to this register generates a Configuration Register Write on the FBD Channel; reading from this register generates a Configuration Register Read on the FBD Channel.

3.13.6.34 FBD Thermal Trip Status Register - 0x84_0000_0A00

TABLE 3-67 FBD Thermal Trip Status Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:48]	0x0	RO	Reserved
TTRIP1_11	[47:46]	0x0	RO	Thermal Trip information for AMB 11, Channel 1
TTRIP1_10	[45:44]	0x0	RO	Thermal Trip information for AMB 10, Channel 1
TTRIP1_9	[43:42]	0x0	RO	Thermal Trip information for AMB 9, Channel 1
TTRIP1_8	[41:40]	0x0	RO	Thermal Trip information for AMB 8, Channel 1
TTRIP1_7	[39:38]	0x0	RO	Thermal Trip information for AMB 7, Channel 1
TTRIP1_6	[37:36]	0x0	RO	Thermal Trip information for AMB 6, Channel 1
TTRIP1_5	[35:34]	0x0	RO	Thermal Trip information for AMB 5, Channel 1
TTRIP1_4	[33:32]	0x0	RO	Thermal Trip information for AMB 4, Channel 1
TTRIP1_3	[31:30]	0x0	RO	Thermal Trip information for AMB 3, Channel 1
TTRIP1_2	[29:28]	0x0	RO	Thermal Trip information for AMB 2, Channel 1
TTRIP1_1	[27:26]	0x0	RO	Thermal Trip information for AMB 1, Channel 1
TTRIP1_0	[25:24]	0x0	RO	Thermal Trip information for AMB 0, Channel 1
TTRIP0_11	[23:22]	0x0	RO	Thermal Trip information for AMB 11, Channel 0
TTRIP0_10	[21:20]	0x0	RO	Thermal Trip information for AMB 10, Channel 0
TTRIP0_9	[19:18]	0x0	RO	Thermal Trip information for AMB 9, Channel 0
TTRIP0_8	[17:16]	0x0	RO	Thermal Trip information for AMB 8, Channel 0
TTRIP0_7	[15:14]	0x0	RO	Thermal Trip information for AMB 7, Channel 0

TABLE 3-67 FBD Thermal Trip Status Registers *(Continued)*

Field	Bit Position	Initial Value	R/W	Description
TTRIP0_6	[13:12]	0x0	RO	Thermal Trip information for AMB 6, Channel 0
TTRIP0_5	[11:10]	0x0	RO	Thermal Trip information for AMB 5, Channel 0
TTRIP0_4	[9:8]	0x0	RO	Thermal Trip information for AMB 4, Channel 0
TTRIP0_3	[7:6]	0x0	RO	Thermal Trip information for AMB 3, Channel 0
TTRIP0_2	[5:4]	0x0	RO	Thermal Trip information for AMB 2, Channel 0
TTRIP0_1	[3:2]	0x0	RO	Thermal Trip information for AMB 1, Channel 0
TTRIP0_0	[1:0]	0x0	RO	Thermal Trip information for AMB 0, Channel 0

3.13.6.35 MCU Syndrome Register - 0x84_0000_0C0

TABLE 3-68 MCU Syndrome Registers

Field	Bit Position	Initial Value	R/W	Description
VALID	[63]	0x0	RW	Error Status is Valid
RSVD	[62:30]	0x0	RO	Reserved
ALERT1	[29:18]	0x0	RW	AMB's on Channel 1 with Status Alert Asserted
ALERT0	[17:6]	0x0	RW	AMB's on Channel 0 with Status Alert Asserted
SOFTRESET	[5]	0x0	RW	Soft Channel Reset Performed on Channel
FASTRESET	[4]	0x0	RW	Fast Reset Performed on Channel
SOURCE	[3:0]	0x0	RW	Source(s) of error (multiple may be asserted: 0xXXX1: CRC Error 0xXX1X: Alert Frame 0xX1XX: Status Alert Asserted 0x1XXX: Status Frame Parity Error

3.13.6.36 Injected Error Source Register - 0x84_0000_0C08

TABLE 3-69 Injected Error Source Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:2]	0x0	RO	Reserved
SOURCE	[1:0]	0x0	RW	Source for Injected Error: 0x0 - CRC Error 0x1 - Alert Frame 0x2 - Status Alert Asserted 0x3 - Status Frame Parity Error

3.13.6.37 MCU FBR Count Register - 0x84_0000_0C10

TABLE 3-70 MCU FBR Count Registers

Field	Bit Position	Initial Value	R/W	Description
RSVD	[63:17]	0x0	RO	Reserved
COUNTONE	[16]	0x0	RW	Hardware behaves as if count was always one, i.e. it will always generate an interrupt
COUNT	[15:0]	0x0	RW	Number of recoverable errors before a recoverable error interrupt will be sent to NCU

3.14 Other Registers

3.14.1 Self-Refresh Registers

The Clock Control Register in the Clock Control Unit (CCU), described in Section 11.1 of the OpenSPARC T2 Programmer's Reference Manual, controls self-refresh mode for the MCU upon assertion of warm reset.

CHAPTER 4

Test Control Unit (TCU)

This document defines the architecure for the Test Control Unit (TCU) for OpenSPARC T2 . It is intended for RTL design engineers of other OpenSPARC T2 blocks, verification engineers, and DFT engineers. The document contains the functional description, and some level of implementation detail for the TCU, test control unit.

This chapter contains the following sections:

4.1 Introduction

The TCU is the OpenSPARC T2 Test Control Unit and provides access to the chip test logic. It also participates in Reset, EFuse programming, clock stop/start sequencing, and chip debug. The TCU including JTAG is completely stuck-fault testable via ATPG manufacturing scan.

4.1.1 Features

The features available for debug or test, implemented in OpenSPARC T2 and supported by the TCU are as follows:

- ATPG or Manufacturing scan for stuck-fault testing.
- TAP and Boundary Scan (JTAG) - support for IEEE 1149.1 and 1149.6
- JTAG scan for scan chain loading and unloading.
- JTAG shadow scan - allows for inspection of specific registers while part is running in system.
- Support for Macrotest.
- JTAG UCB - Allows CREG access via instructions sent to the NCU which will then intermix the transaction with normal requests. The NCU will then take the results and pass them back to the TCU which can then send out TDO.
- EFuse - Control and programming
- Transition fault testing - This is done on the tester while PLLs are locked; slower domains may be driven via pins directly.
- MBIST - Memory Built-in Self Test; tests Array bit cells and write/read mechanisms. BISI (Built-in Self Initialization) allows arrays to be initialized.
- LBIST - Logic BIST, implemented in cores.
- Reset - Handshaking with RST unit to control scan flop reset and clock stop/start sequencing.
- L2 Access - via JTAG through the SIU.
- Debug Support.
- Support for SerDes - ATPG, STCI, boundary scan

4.2 JTAG

The JTAG block resides in the TCU. This interface will be used to access not only standard JTAG services but also implementation of specific debug features. The JTAG architecture is designed to be compliant with the IEEE 1149.1 Standard.

JTAG provides these features:

1. Access to JTAG ID code
2. Implementation of JTAG public instructions (see Note)
3. Ability to load/unload chip scan chains as a single chain, or individually
4. Initiation and control of Shadow scan
5. Initiation and control of Boundary scan
6. Control of MBIST or BISI
7. Control of LBIST
8. Interface to Chip UCB
9. Interface to E-Fuse Unit
10. Interface to various debug features
11. Control of clock domains (starting, stopping)
12. Write/read access to L2

The following JTAG pins are implemented for OpenSPARC T2:

- TDI
- TDO
- TMS
- TRST_L
- TCK

Note – The JTAG unit implements all instructions specified as mandatory in the 1149.1 and 1149.6 standards along with a number of private instructions that help the debugger to access specific debug features. However, not all I/O on OpenSPARC T2 support the HIGHZ and SAMPLE instructions. HIGHZ and SAMPLE are not supported on SerDes I/O. In addition, some non-SerDes I/O do not implement HIGHZ correctly.

4.2.1 Instruction Register

The instruction register provides 8 bits to access up to 256 instructions. On the rising edge of TCK in the capture-IR state, the instruction register shift portion is updated to the IDCODE instruction. The instruction register update portion loads the IDCODE instruction on the falling edge of TCK in the reset state, or when TRST_L goes low.

TABLE 4-1 JTAG Instruction Register

Instruction Decode	Value on Reset
7:0	IDCODE Instr: 8b 0000 0001

4.2.2 Reset State and TRST_L

The TRST_L pin provides an asynchronous reset for the JTAG state machine and associated registers. When TRST_L is activated (low), the TMS pin should be held high and it is recommended that TCK be held off. When TRST_L goes low:

- the TAP state machine is put in the test-logic-reset state
- the Instruction Register is set to the IDCODE instruction
- All data registers internal to the JTAG block are reset to their default states

After TRST_L is deasserted it is recommended to keep TCK off until JTAG is to be used, and then allow TCK to run with TMS be held high for a few cycles to allow the reset state inside JTAG to stabilize before entering the Run-Test-Idle state.

Synchronous resetting of the TAP is done by entering the test-logic-reset state via control of TMS and TCK. This does not necessarily reset private data registers to their default states.

4.2.3 Instruction Summary

Unimplemented or undefined instructions will default to the BYPASS instruction.

TABLE 4-2 JTAG Public Instructions

Instruction	Encoding	Description
TAP_BYPASS	0xFF	Mandatory; selects BYPASS REGISTER
TAP_EXTEST	0x00	Mandatory; selects BOUNDARY SCAN REGISTER
TAP_IDCODE	0x01	Optional per standard; selects IDCODE DR

TABLE 4-2 JTAG Public Instructions

Instruction	Encoding	Description
TAP_CLAMP	0x04	Optional per standard
TAP_EXTEST_PULSE	0x05	Mandatory for 1149.6
TAP_EXTEST_TRAIN	0x06	Mandatory for 1149.6

TABLE 4-3 JTAG Private Instructions

Instruction	Encoding	Description
TAP_SAMPLE_PRELOAD	0x02	Mandatory; shared encoding allowed per standard; selects BOUNDARY SCAN REGISTER - SerDes I/O do not support SAMPLE part of this instruction
TAP_HIGHZ	0x03	Optional per standard - SerDes I/O do not support HIGHZ, and some DBG_DQ I/O have weak pullup/down resistors.
UNDEFINED	0x07	--
TAP_CREG_ADDR	0x08	Stores address to be used for system access to control reg (ASI/IO mapped)
TAP_CREG_WDATA	0x09	Stores data to be used for system access to control reg
TAP_CREG_RDATA	0x0A	Captures data from system access
UNDEFINED	0x0B	--
TAP_NCU_WRITE	0x0C	Initiates write to system control register
TAP_NCU_READ	0x0D	Initiates read from system control register
TAP_NCU_WADDR	0x0E	Combination of TAP_CREG_ADDR and TAP_NCU_WRITE
TAP_NCU_WDATA	0x0F	Combination of TAP_CREG_WDATA and TAP_NCU_WRITE
TAP_NCU_RADDR	0x10	Combination of TAP_CREG_ADDR and TAP_NCU_READ
UNDEFINED	0x11-0x12	--
TAP_MBIST_CLKSTPEN	0x13	Enables clock stop for mbist via cycle counter
TAP_MBIST_BYPASS	0x14	Select engines to by excluded from MBIST operation; using mbist_bypass data register
TAP_MBIST_MODE	0x15	Specify serial/parallel, diag. mode or bist/bisi modes via mbist_mode data reg
TAP_MBIST_START	0x16	Initiate MBIST
UNDEFINED	0x17	--
TAP_MBIST_RESULT	0x18	Query 2-bit done/fail register: and/or of all MBIST engines
TAP_MBIST_DIAG	0x19	Run MBIST on one array; MBIST engine & arrays are data reg
TAP_MBIST_GETDONE	0x1A	Query 48-bit done data register, one bit per MBIST engine

TABLE 4-3 JTAG Private Instructions

Instruction	Encoding	Description
TAP_MBIST_GETFAIL	0x1B	Query 48-bit fail data register, one bit per MBIST engine
TAP_DMO_ACCESS	0x1C	Set DMO Mode - enables DMO logic and package pins
TAP_DMO_CLEAR	0x1D	Clears DMO Mode
TAP_DMO_CONFIG	0x1E	Access 48-bit DMO configuration register
TAP_MBIST_ABORT	0x1F	Stop any MBIST activity and reset MBIST controls
UNDEFINED	0x20-0x27	--
TAP_FUSE_READ	0x28	Shift out 32 bits selected by ROW_ADDR; selects EFUSE DR
TAP_FUSE_BYPASS_DATA	0x29	Provides user-data directly to EFU; selects EFUSE DR
TAP_FUSE_BYPASS	0x2A	Starts EFU control using bypass data provided by user
TAP_FUSE_ROW_ADDR	0x2B	Shift in 7-bit row address for EFU access; selects EFU ROW ADDRESS DR
TAP_FUSE_COL_ADDR	0x2C	Shift in 5-bit column address for EFU programming; selects EFU COLUMN ADDRESS DR
TAP_FUSE_READ_MODE	0x2D	Configures EFU with 3 bits for EFU access; selects EFU READ MODE DR
TAP_FUSE_DEST_SAMPLE	0x2E	Samples EFU destination redundancy value from the destination specified
TAP_FUSE_RVCLR	0x2F	Access 7-bit redundancy value clear register
TAP_SPCTHR0_SHSCAN	0x30	Samples thread 0 for all available cores
TAP_SPCTHR1_SHSCAN	0x31	Samples thread 1 for all available cores
TAP_SPCTHR2_SHSCAN	0x32	Samples thread 2 for all available cores
TAP_SPCTHR3_SHSCAN	0x33	Samples thread 3 for all available cores
TAP_SPCTHR4_SHSCAN	0x34	Samples thread 4 for all available cores
TAP_SPCTHR5_SHSCAN	0x35	Samples thread 5 for all available cores
TAP_SPCTHR6_SHSCAN	0x36	Samples thread 6 for all available cores
TAP_SPCTHR7_SHSCAN	0x37	Samples thread 7 for all available cores
TAP_L2T_SHSCAN	0x38	Samples specified error registers in the 8 L2 Tags
UNDEFINED	0x39-0x3F	--
TAP_CLOCK_SSTOP	0x40	Soft Stop of clocks; cores only
TAP_CLOCK_HSTOP	0x41	Hard Stop of clocks
TAP_CLOCK_START	0x42	Start clocks
TAP_CLOCK_DOMAIN	0x43	Specify entry clock domain for stopping/starting clocks

TABLE 4-3 JTAG Private Instructions

Instruction	Encoding	Description
TAP_CLOCK_STATUS	0x44	2-bit status indicating if clock stop/start routine finished
TAP_CLKSTP_DELAY	0x45	7-bits; Specify up to 128 cycle delay between successive clk_stop Signals
TAP_CORE_SEL	0x46	8-bit register to specify target SPC cores for clock operations.
UNDEFINED	0x47	--
TAP_DE_COUNT	0x48	Access 32-bit Debug Event Counter
TAP_CYCLE_COUNT	0x49	Access 64-bit Reset/Cycle Counter
TAP_TCU_DCR	0x4A	Access 4-bit TCU Debug (event) Control Register
UNDEFINED	0x4B	--
TAP_CORE_RUN_STATUS	0x4C	Access 64-bit CMP core-running-status reg.
TAP_DOSS_ENABLE	0x4D	Access 64-bit disable overlap/single step mode enable register
TAP_DOSS_MODE	0x4E	Specify either disable overlap or single step mode; [1]=enable, [0]=single step if set to '1', disable overlap if set to '0'
TAP_SS_REQUEST	0x4F	Pulse single step request signal; need to go through update-dr
TAP_DOSS_STATUS	0x50	8-bit status for disable overlap or single step completion
TAP_CS_MODE	0x51	Specify cycle-step mode. 1-bit register. set to '1' to enable; uses Cycle Counter for cycle-step operation.
TAP_CS_STATUS	0x52	Read 1-bit status indicating cycle stepping has completed.
UNDEFINED	0x53-0x57	--
TAP_L2_ADDR	0x58	Load L2 Address (to be written to or read from)
TAP_L2_WRDATA	0x59	Load L2 Write Data
TAP_L2_WR	0x5A	Initiate write to L2: WRDATA to ADDR
TAP_L2_RD	0x5B	Initiate read from L2 at ADDR and receive L2 data
UNDEFINED	0x5C-0x5F	--
TAP_LBIST_START	0x60	Initiate Logic BIST
TAP_LBIST_BYPASS	0x61	Bypass Logic BIST for specified cores; 1 bit per core
TAP_LBIST_MODE	0x62	Control program mode; paralle/serial modes
TAP_LBIST_ACCESS	0x63	Place one Logic BIST controller between TDI-TDO
TAP_LBIST_GETDONE	0x64	Determine if Logic BIST is done across all selected cores
TAP_LBIST_ABORT	0x65	Abort any Logic BIST currently in progress
UNDEFINED	0x66-0x7F	--

TABLE 4-3 JTAG Private Instructions

Instruction	Encoding	Description
TAP_SERSCAN	0x80	Access internal scan chains; selects INTERNAL SCAN FLOPS as DATA REGISTER
TAP_CHAINSEL	0x81	Select all or one of 32 chains for serial scan mode using CHAIN SELECT DR
TAP_MT_ACCESS	0x82	Enables Macro Test mode for JTAG Scan
TAP_MT_CLEAR	0x83	Clears Macro Test mode
TAP_MT_SCAN	0x84	Similar to TAP_SERSCAN but drives TCK onto clock tree for capture pulses during RTI state
UNDEFINED	0x85-0x87	--
TAP_TP_ACCESS	0x88	Enables Test Protect mode to block inputs to TCU and other blocks such as RST, CCU, and DMU
TAP_TP_CLEAR	0x89	Clears Test Protect mode
UNDEFINED	0x8A-0x8F	--
TAP_STCI_ACCESS	0x90	Enables STCI mode for SerDes Test Configuration Interface Bus
TAP_STCI_CLEAR	0x91	Clears STCI mode for SerDes Test Configuration Interface Bus
UNDEFINED	0x92-0x9F	--
TAP_JTPOR_ACCESS	0xA0	Enables JTAG access window during POR sequence
TAP_JTPOR_CLEAR	0xA1	Clears JTAG access window during POR sequence
TAP_JTPOR_STATUS	0xA2	JTAG access window status: returns ‘1’ if window is active
TAP_SCKBYP_ACCESS	0xA3	Enables Bypass for SCK counter in NCU
TAP_SCKBYP_CLEAR	0xA4	Clears Bypass for SCK counter in NCU
UNDEFINED	0xA5-0xFE	--

4.2.4 Data Registers

The data registers accessible via JTAG are listed in Table 4-4 on page 9. The least significant bit (lsb: bit 0, the rightmost bit) is always closest to tdo. An update of

TABLE 4-4 JTAG Data Registers

Data Register	Width	Capture	Update	Description & JTAG Instructions
Boundary Scan	~ 3 x I/O's	Yes	Yes	I/O pin boundary scan cells; see private instructions
Bypass	1	Yes	No	Selected via bypass instr. or any undefined instr.
ID Code	32	Yes	No	See following section; TAP_IDCODE
Chain Select	6	No	Yes	Active during JTAG Serial Scan; TAP_CHAINSEL
Serial (Internal) Scan	~1200K	No	via scan	Chip scan chains; TAP_SERSCAN, TAP_MT_SCAN
Macrotest Enable	1	No	Yes	Enables write/read control of arrays during serial scan; set with TAP_MT_ACCESS, clear with TAP_MT_CLEAR
Test Protect Enable	1	No	Yes	Enables Test Protect mode; set with TAP_TP_ACCESS, clear with TAP_TP_CLEAR
EFUSE	32	Yes	No	Contents of 1 row in EFuse array, per Row Addr.; TAP_FUSE_READ
EFUSE Bypass_Data	32	No	Yes	Data for BYPASSING EFuse Array; TAP_FUSE_BYPASS_DATA
EFUSE Row_Address	7	Yes	Yes	Select one row in the EFuse array; TAP_FUSE_ROW_ADDR
EFUSE Column_Address	5	Yes	Yes	EFUSE Column, only for programming; TAP_FUSE_COL_ADDR
EFUSE Read_Mode	3	Yes	Yes	See EFUSE document; TAP_FUSE_READ_MODE
EFUSE Dest_Sample	32	Yes	No	See EFUSE Document; TAP_FUSE_DEST_SAMPLE
EFUSE RVCLEAR	7	No	Yes	TAP_FUSE_RVCLR bit[6]=enable; bits[5:0]=RV_ID
MBIST Result	2	Yes (1)	No	Bit 1=1 when MBIST Done, Bit 0=1 if MBIST failed; TAP_MBIST_RESULT
MBIST Bypass	48	Yes	Yes	MBIST: Specify engines to bypass during MBIST; TAP_MBIST_BYPASS
MBIST Done	48	Yes (1)	No	MBIST engine Done status bits; TAP_MBIST_GETDONE
MBIST Fail	48	Yes (1)	No	MBIST engine Fail status bits; TAP_MBIST_GETFAIL

TABLE 4-4 JTAG Data Registers

Data Register	Width	Capture	Update	Description & JTAG Instructions
MBIST Diag	Variable	No	via scan	All registers in an MBIST engine. Updated via scan only. TAP_MBIST_DIAG
MBIST Mode	4	No	via scan	Select serial or parallel modes; bisi; user mode TAP_MBIST_MODE
CREG Address	40	No	Yes	40-bit address for system control register; TAP_CREG_ADDR, TAP_NCU_WADDR, TAP_NCU_RADDR
CREG Write_Data	64	No	Yes	64-bit data to be written to system control register; TAP_CREG_WDATA, TAP_NCU_WDATA
CREG Read_Data	65	Yes	No	65-bit data read from system control register; TAP_CREG_RDATA; due to sentinel bit, scan-out data is blocked to TDO during shiftDR
Core Shadow_Scan	8*len	Yes	via scan	Shadow scan for all available cores concatenated, spc0 to spc7; TAP_SPCTHR0_SHSCAN-TAP_SPCTHR7_SHSCAN
L2TAG Shadow_Scan	8*len	Yes	via scan	Shadow scan for all l2 tags concatenated, l2t0 to l2t7; TAP_L2T_SHSCAN
Clock Domain	32	Yes (1)	Yes	Specify starting points for turning clocks on or off; TAP_CLOCK_DOMAIN; bits [31:24] reserved, should be loaded to zeros.
Clock Status	2	Yes (1)	No	TAP_CLOCK_STATUS bits = 00 --> clock sequencer is running bits = 01 --> clock sequencer has started all clocks bits = 10 --> clock sequencer has stopped all clocks bits = 11 --> should not happen; indeterminate
Clock Stop Delay	7	Yes (1)	Yes	Delay between successive clk_stop's to clk domains; TAP_CLKSTP_DELAY
Core Select	8	Yes (1)	Yes	Enables clock stop to target cores; TAP_CORE_SEL
Debug Event Counter	32	Yes (2)	Yes	Decrementing counter to delay debug action by counting debug events; TAP_DE_COUNT
Cycle Counter	64	Yes (2)	Yes	Decrementing counter triggered by debug event; upper word is Reset Counter; TAP_CYCLE_COUNT
TCU Debug Control	4	Yes (1)	Yes	Control reg. for TCU debug events; TAP_TCU_DCR
(CMP) Core Run Status	64	Yes (1)	No	Thread (CMP) running status register; TAP_CORE_RUN_STATUS
DOSS Enable	64	Yes	Yes	Disable Overlap (do) and Single Step (ss) enable bits, per thread; TAP_DOSS_ENABLE

TABLE 4-4 JTAG Data Registers

Data Register	Width	Capture	Update	Description & JTAG Instructions
DOSS Mode	2	Yes	Yes	Controls disable overlap or single step modes; TAP_DOSS_MODE
DOSS Status	8	Yes (1)	No	Indicates completion of disable overlap or single step per SPC; TAP_DOSS_STATUS
Cycle Step Enable	1	Yes (1)	Yes	Enable cycle step mode; TAP_CS_MODE
Cycle Step Status	1	Yes (1)	No	Indicates cycle step is complete; TAP_CS_STATUS
L2_Addr	65	No (3)	Yes	64-bit address to L2 for write or read; bit 0 ignored; TAP_L2_ADDR
L2_Write_Data	65	No (3)	Yes	64-bit data to write to L2; bit 0 ignored; TAP_L2_WRDATA
L2_Read_Data	65	Yes (1)	No	64-bit data received from L2 per ADDR; bit 0 indicates data is valid, bits 64:1 are data; TAP_L2_RD
LBIST Bypass	8	Yes	Yes	One bit per core, to bypass an engine set to '1'; TAP_LBIST_BYPASS
LBIST Mode	2	No	via Scan	bit[1]: enable user (program) mode bit[0]: 0=serial, 1=parallel TAP_LBIST_MODE
LBIST Access		No	via Scan	Place one Logic BIST controller between TDI-TDO; TAP_LBIST_ACCESS
LBIST Done	8	Yes (1)	No	Read status of all enabled Logic BIST controllers; TAP_LBIST_GETDONE
DMO Config	48	Yes (2)	Yes	Access 48-bit DMO Configuration register; TAP_DMO_CONFIG
JTAG POR Status	1	Yes (1)	No	Access 1-bit status for JTAG POR Access; TAP_JTPOR_STATUS

"Yes" means there is an update register loaded during UpdateDR; an update of "via scan" means there is no separate update register.

For notes (1) and (2) in the Capture field of Table 4-4 please see Section 4.2.7 on page 14.

Note (3): L2_Addr and L2_Write_Data registers cannot be observed during shiftDR due to a bug in the RTL - see errata for details.

4.2.4.1 Boundary Scan

The boundary scan data register is selected by EXTEST, SAMPLE/PRELOAD, HIGHZ and CLAMP and is defined by the BSDL (Boundary Scan Description Language) file for OpenSPARC T2. It is also selected by the 1149.6 instructions TAP_EXTEST_TRAIN and TAP_EXTEST_PULSE and is part of the internal scan register when selected by manufacturing scan.

The HIGHZ instruction is not supported by the SerDes I/O, and also some of the DBG_DQ pins have weak pullup or pulldown resistors. So the HIGHZ instruction is not fully supported by OpenSPARC T2. See the JTAG errata section for details.

The SAMPLE instruction (encoded with PRELOAD) is not supported by the SerDes I/O, but the PRELOAD instruction is supported.

4.2.4.2 Bypass Register

This is a one-bit register. The bypass register loads "0" on the rising edge of TCK in the capture-DR state when the bypass register is selected. All non-specified instructions cause the bypass register to be selected, so that it is placed between TDI and TDO.

4.2.4.3 ID Code Register

The ID Code register is a 32-bit read-only register defined as:

TABLE 4-5 ID Code Register

Version Field	Part Number Field	Manufacturing ID Field	lsb
[31:28]	[27:12]	[11:1]	[0]
Initially 0x0; updated per BSDL change	0x2aaa	0x03e	1

The ID Code register is always placed between TDI and TDO when the select-DR state is reached directly after the test-logic-reset state with no intervening instruction register programming. The lsb is closest to the TDO as required by the standard.

4.2.4.4 CMP Data Registers

Access for all CMP registers will be via UCB (TAP_CREG_ and TAP_NCU_ instructions).

Threads in each core are virtual cores; for those CMP registers specifying physical cores each physical core is assigned 8 bits in a 64-bit register; allowed values are 8'b11111111 and 8'b00000000 The assigned 8 bits are [63:56] = core 7; [55:48] = core 6; [47:40] = core 5; [39:32] = core 4; [31:24] = core 3; [23:16] = core 2; [15:8] = core 1; [7:0] = core 0.

4.2.5 JTAG SCK Bypass

To get around the SSI lock loss issue in OpenSPARC T2 (see bug 97461) , NCU implements a down counter that would decrement to zero starting from the first iol2clk after flush reset completion (part of POR and Warm resets). After this counter expires (reaches 0 count), NCU will send out the first fetch on SSI interface by asserting SSI_MOSI. By this time the FPGA would have relocked against the SSI_SCK coming from OpenSPARC T2.

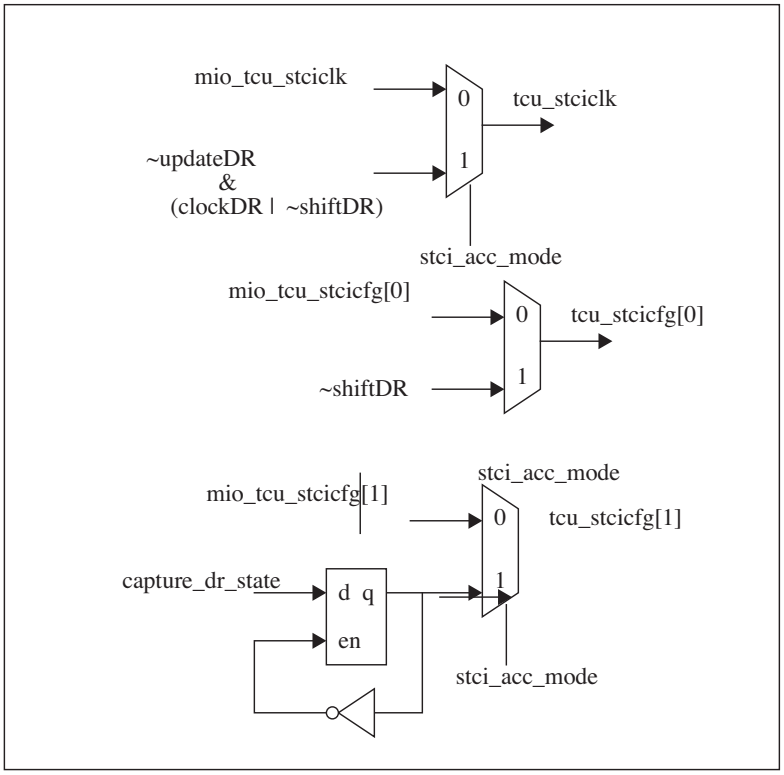
To remove this 5-6 msec wait on the tester, TCU supports a bit that is programmable through JTAG on the tester between deasssertion of TRST_L and assertion of POR2 by the reset block. This bit preserves its value on POR2 and WARM resets and is sent by the TCU to the NCU as a signal tcu_sck_bypass . When this signal is a 1, (bit programmed to 1'b1 for the tester), the NCU bypasses the counter to assert SSI_MOSI (thereby eliminating the 5-6 msec wait time on the tester). If this signal is a 0, (POR1 reset state of the bit in TCU) then the NCU honors the counter and waits till the counter expires before asserting SSI_MOSI. This would be the configuration in the system.

The TAP_SCKBYP_ACCESS and TAP_SCKBYP_CLEAR instructions are used to set and reset, respectively, the tcu_sck_bypass signal sent from TCU to NCU.

4.2.6 JTAG Access to SerDes STCI

JTAG provides access to the SerDes STCI bus. There are four inputs, STCICLK, STCICFG[1:0], and STCID, and one output STCIQ. To enable JTAG access to STCI, the JTAG instruction TAP_STCI_ACCESS should be executed. During STCI JTAG Access, STCICLK and STCICFG are driven as shown in FIGURE 4-1, while STCID is connected to TDI and STCIQ to TDO. To clear JTAG access to STCI, use TAP_STCI_CLEAR or reset the TAP state machine. The updateDR, clockDR andshiftDR are as specified in the IEEE 1149.1 spec., while capture_dr_state is active during the Capture DR TAP state.

FIGURE 4-1 SerDes STCI Bus Control



4.2.7 JTAG Errata

Erratum #1 JTAG accesses to some registers in CMP clock domain may result in erratic read values.

Symptom: JTAG reads of some registers which exist in CMP clock domain may result in erratic read data, if the registers are being updated when the read occurs.

Description: There are several JTAG instructions that can sample data from registers contained in the CMP clock domain. These are sampled with TCK in the JTAG block without synchronization. Consequently, if those registers are changing in the cmp domain, the read results will be indeterminate. The indeterminism results from the

asynchronous nature of TCK with respect to the CMP clock. It is permissible to read these registers directly after they are written with JTAG, before their correpsonding activity is initiated, to verify contents were correctly written.

For the first class of registers the data will generally be stable when reading, or at least changing very infrequently. Thus, it is recommended that when reading these registers, back to back reads be performed so that the contents can be compared; if the same, then the read is successful.

JTAG Registers in this class are indicated with a (1) in the Capture column of Table 4-4:

- Clock Domain[31:0]
- Core Run Status[63:0]
- MBIST Done[47:0]
- MBIST Fail[47:0]
- MBIST Result[1:0]
- LBIST Done[7:0]
- L2_Read_Data[64:0]
- Clock Stop Delay[6:0]
- Clock Status[1:0]
- Core Select[7:0]
- TCU Debug Control[3:0]
- DOSS Status[7:0]
- Cycle Step Enable[0]
- Cycle Step Status[0]
- JTAG POR Status[0]

For the second class of registers, the data may be changing every cycle. In this case it is recommended that the user refrain from reading these until the contents are stable - this can be determined by examination of the associated status registers.

Registers in this class are indicated with a (2) in the Capture column of Table 4-4

- Debug Event Counter[31:0]
- Cycle Counter[63:0]
- DMO Config[47:0]

Workaround: JTAG users should either 1) read the registers in the first group multiple times until two stable values are obtained, or 2) read the status registers associated with the registers in the second group to make sure hardware is not updating them.

Erratum #2 JTAG view of some CSR registers is not

Symptom: When accessing certain JTAG registers that are also accessible via SW as CSR's, there is a possibility that the JTAG view of the register can be inconsistent with the SW view. The SW view will always be correct, but the JTAG view may be incorrect.

Description: The TCU does not properly synchronize the update signal for writes to certain JTAG registers when written by SW. There are two copies of certain JTAG registers, one in the TCK clock domain inside JTAG, and one in the IO clock domain which services UCB access by SW. These registers are supposed to be coherent, but if SW writes to one of these registers it may be possible for the JTAG register to miss the data and be incoherent, or even to update with indeterminate data.

The registers affected by this condition (with their bit positions relative to JTAG) are:

- MBIST Mode[3:0]
- MBIST Bypass[47:0]
- MBIST Abort[0]
- LBIST Mode[1:0]
- LBIST Bypass[7:0]

One way that these registers can become incoherent is when SW writes its copy, the logic in JTAG also tries to write the same data into the JTAG register. But, the write pulse is not synchronized and may be missed depending on the relative frequencies of TCK and the IO clock. Hence, it is possible that the JTAG register will not be updated, or it may be updated (corrupted) with indeterminate data.

A second way that these registers can become incoherent is if the TCK is not running. In order to maintain coherency, the TCK clock must be running. However, the JTAG clock - TCK - is not required to run for functional operation, so there is no guarantee that TCK is even active when SW writes its register. So the JTAG version of the register is not updated.

Workaround: Given these problems, and since it is always the JTAG view which may be incorrect, the JTAG user should be aware that if SW writes these registers then the JTAG view may be incorrect.

Erratum #3 HIGH-Z Boundary Scan instruction is not supported

Symptom: Pins associated with DBG_DQ bus in the MIO have weak pullups or pulldowns, and thus do not go to a high-z state when instructed to do so. In addition, SerDes pins do not support the high-z state.

Description: The JTAG TAP_HIGHZ instruction is intended to put all output pins in a high-z state (tristate) and is typically used during manufacturing board test to prevent overdriving Signals. Due to the use of SerDes macros which do not support the high-z capability, and the use of pullups and pulldowns in the DBG_DQ bus pins, many of the OpenSPARC T2 output pins can not go to a high-z state even when instructed to do so by the TCU.

Workaround: Do not use the TAP_HIGHZ instruction. Support for the JTAG TAP_HIGHZ instruction has been moved from the public to the private section of the boundary scan description language file (BSDL); this is legal since HIGHZ is optional per the IEEE 1149.1 standard. The actual JTAG instruction, TAP_HIGHZ, has been kept in place but is now a private instruction. This was done since the RTL still supports the TAP_HIGHZ instruction. Moving it to private means it is not a supported public instruction and should not be used.

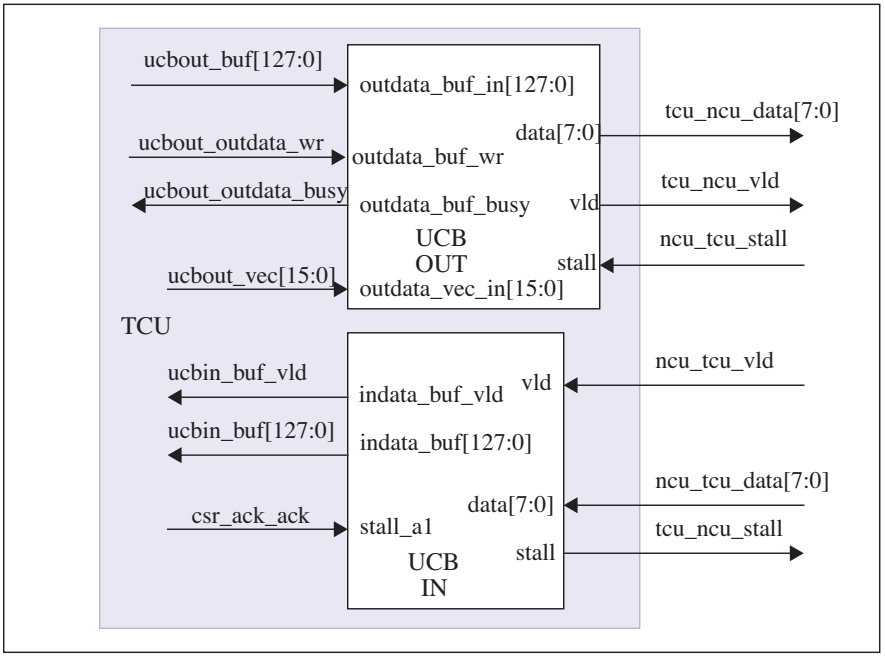
4.3 UCB Interface

The Unit Control Bus interface is a protocol for transmission of packets via the NCU between units. It is implemented inside the TCU and allows access via JTAG to IO mapped registers, and some ASI registers. A register's address and data in the case of writes are loaded via JTAG into holding registers in the TCU. The TCU then uses its UCB interface to communicate to the NCU which puts the new transaction (packet) into the data flow. The interface allows both reading and writing. On OpenSPARC T2, UCB access through the crossbar to the l2 and cores is not available so access to the L2 is done via a separate interace between the TCU and the SIU.

For a WRITE, a 40-bit address and 64 bits of data must be provided by JTAG to the UCB. For a READ, a 40-bit address is needed, with the data received from the NCU captured into a register in the TCU. To implement a READ, a sentinel bit is used since the exact timing of the read return is not deterministic. The system is only allowed to have 1 read outstanding at one time. There is no protection built in against this, adherence is left to the user.

4.3.1 UCB Simple Block Diagram

FIGURE 4-2 UCB Interface Inside the TCU



4.3.2 JTAG Instructions used to Access the UCB

The following descriptions are excerpts from the OpenSPARC T1 DFT specification and the OpenSPARC T1 DFT User's Guide but have been ported to OpenSPARC T2.

TAP_CREG_ADDR

Load System Address: Causes a 40-bit address register to become accessible from TDI. The target system address is loaded during shift-DR. On Update-DR a transfer occurs from the TCK domain to a 40-bit holding register in the IO CLK domain.

TAP_CREG_WDATA

Load System Write Data: Causes a 64-bit data register to become accessible from TDI, into which the data for the specified system address is loaded during shift-DR. On Update-DR a transfer occurs from the TCK domain to a 64-bit holding register in the IO CLK domain.

TAP_CREG_RDATA

Load System Read Data: Causes a 65-bit data register to become accessible from TDO. The 65th bit is used as a sentinel to allow driver software to synchronize with the read operation. While the read is outstanding the sentinel bit remains zero. Once the NCU has returned valid data then the read is complete and the sentinel bit is set to one. To use this, the JTAG is kept in ShiftDR and TCK is clocked until the TDO reads a "1", this indicates the sentinel bit has been set. When the sentinel bit becomes one, the next 64 bits shifted out are the valid read data.

The TCU can only issue a single access at a given time to the NCU. The user is responsible for ensuring that this is the case. Note too that the TCU does not report erroneous reads made to the NCU. Therefore, the driver software should time out on a read, assuming an error if this occurs.

TAP_NCU_WRITE

Initiate Write Transaction: Causes a write transaction to be initiated on Update-IR.

TAP_NCU_READ

Initiate Read Transaction: Causes a read transaction to be initiated on Update-IR

TAP_NCU_WADDR

Load System Address and Initiate Write Transaction: Causes a 40-bit address register to become accessible from TDI. The target system address is loaded during shift-DR. On Update-DR a transfer occurs from the TCK domain to a 40-bit holding register in the IO_CLK domain. In the cycle after the transfer is complete the contents of the address register is forwarded to the UCB interface and a write transaction is initiated. This instruction is a combination of TAP_CREG_ADDR and TAP_NCU_WRITE.

TAP_NCU_WDATA

Load Write Data and Initiate Write Transaction: Causes a 64-bit data register to become accessible from TDI, into which the data for the specified system address is loaded during shift-DR. On Update-DR a transfer occurs from the TCK domain to a 64-bit

holding register in the IO_CLK domain. In the cycle after the transfer is complete the contents of the address register and data register are forwarded to the UCB interface to initiate a write transaction. This instruction is a combination of TAP_CREG_WDATA and TAP_NCU_WRITE.

TAP_NCU_RADDR

Load System Address and Initiate Read Transaction: Causes a 40-bit address register to become accessible from TDI. The target system address is loaded during shift-DR. On Update-DR a transfer occurs from the TCK domain to a 40-bit holding register in the IO_CLK domain. In the cycle after the transfer is complete the contents of the address register is forwarded to the UCB interface and a read transaction is initiated. This instruction is a combination of TAP_CREG_ADDR and TAP_NCU_READ.

4.3.3 Expected Data and Address Format

The data to be written is 64 bits in length. A 40 bit address is also loaded into the ucb address register.

4.3.4 TCU as a Slave for UCB

The OpenSPARC T1 implementation provided only that TCU be a master for UCB interactions. To support debug requirements for OpenSPARC T2, the TCU will also act as a slave for UCB. The interface remains the same, the only changes will be in the TCU. For joint access between JTAG and UCB the result is indeterminate. The list of registers accessible via SW inside the TCU is provided in "TCU Local CSR Assignments" on page 94.

Reading local TCU CSR's via JTAG UCB protocol is not supported; local TCU CSR's should be accessed directly via the appropriate JTAG instructions. Note that the register bit ordering may not be consistent between both methods.

TCU is not designed to handle burst read requests, that is, a read request cannot be followed immediately by another read request, otherwise the second one may be dropped and no read data will be returned and the thread issuing the second request may hang. Users should program the second read request after the data for the first one has returned. In the case of multiple threads accessing TCU CSR registers, some mechanism (such as a semaphore lock) should be used to guarantee only one thread accesses any TCU CSR register at a given time. See "UCB Erratum" on page 21.

Note – Read requests to internal non-existent TCU CSRs (base address 85_0000_0000) cause TCU to respond with a READ_ACK instead of a READ_NACK. This means that TCU responds with garbage data. The requesting thread doesn't hang. Writes to undefined CSR addresses within TCU appear to complete but do not write to any real TCU registers; this is expected behavior. Reads appear to complete also but the READ_ACK is not correct behavior since TCU should respond with a READ_NACK. This behavior will not be fixed. A workaround is to never allow software to request data from undefined TCU CSRs. Software should take care to access only valid TCU register addresses.

4.3.5 UCB Erratum

Erratum #3 TCU UCB hangs on reads from SPARC core

Symptom: A thread hangs while reading a TCU CSR.

Description: Multiple threads can access CSR's inside the TCU and cause the NCU to send back-to-back reads to the TCU. As the TCU sends data back in response to these requests, the NCU may become overloaded and stall the TCU. If the stall lasts for more than one cycle, this will cause the TCU to drop any request that the NCU is sending at the time of the stall.

The TCU is not designed to handle burst read requests where the NCU has to stall the TCU for more than one cycle to receive the data the TCU is returning.

The problem is that the TCU does not provide a buffer for catching incoming requests when stalled, and drops the incoming request. The TCU does not hang, but the thread that issued the request will hang since its request is not serviced by TCU.

This issue only appears with multiple threads. A single thread cannot issue back-to-back read requests since it will always wait for return data before issuing the next request. Only multiple threads can send read requests which appear back-to-back to the NCU and TCU.

Also, this issue requires several back-to-back reads to cause enough activity for the NCU to stall the TCU. Simulation shows that for 4 back-to-back reads, the NCU stalls the TCU on the 3rd request. Consequently, the TCU drops the 4th request.

Workaround: Users should program the second TCU CSR read request to wait until after the data for the first read request has returned. In the case of multiple threads accessing TCU CSR registers, some mechanism (such as a semaphore lock) should be used to guarantee only one thread accesses any TCU CSR register at a given time.

4.4 L2 Access via SIU

4.4.1 JTAG L2 Access Registers

It is possible to write and read the L2 addresses while the chip is running using JTAG. The L2_Addr register is accessed via TAP_L2_ADDR; the L2_Write_Data register is accessed via TAP_L2_WRDATA; and the L2_Read_Data register is accessed via TAP_L2_RD as described below. The L2_Write_data and L2_Read_Data registers are the same physical register.

TABLE 4-6 L2 Access Registers

Register	JTAG Instr.	Bits[64:1]	Bit[0]
L2_Addr[64:0]	TAP_L2_ADDR	bit[64]=1: jtag access bits[63:57] = 000 0001 for read request bits[63:57] = 000 0010 for write request bits[56:41] = Unused bits[40:1] = Physical address (8 byte boundary)	Ignored
L2_Write_Data[64:0]	TAP_L2_WRDATA	bits[64:1] = 8-bytes of Data to write to L2a	Ignored
L2_Read_Data[64:0]	TAP_L2_RD	bits[64:1] = 8-bytes of Data returned from L2	1 = Data Valid

The TCU to L2 interface thru SIU is 4B aligned. The SIU will force bit 2 = 0.

4.4.2 Write

To write the L2 an address and data must be loaded via JTAG using TAP_L2_ADDR and TAP_L2_WRDATA, followed by TAP_L2_WR. When the TAP_L2_WR instruction is active, the run-test-idle state (0xC) of the TAP state machine is used to transfer the address and data to the L2 and at least 128 TCK clocks must be cycled while in RTI state for the transfer to complete. The RTI state should be avoided except for the actual transfer of data, and once entered should not be reentered during the write operation.

4.4.3 Read

A Read is accomplished by loading an address using TAP_L2_ADDR followed by a TAP_L2_RD. When the TAP_L2_RD instruction is active, only 64 TCK clocks need be cycled while in RTI to transfer the address to the L2. Then, repeated passes through capture-DR and shift-DR should be used to retrieve the data returned by the L2. Valid data is indicated during TAP_L2_RD at TDO in the shift-DR state by the presence of a leading '1' (bit[0] of the 65-bit L2_Read_Data register), otherwise another pass through capture-DR should be implemented, without intervening visits to run-test-idle. **Note**: bit 0 is not the same as the "sentinel bit" of the creg access. **Note**: The RTI state should be avoided except for the actual transfer of data, and once entered should not be reentered during the read operation.

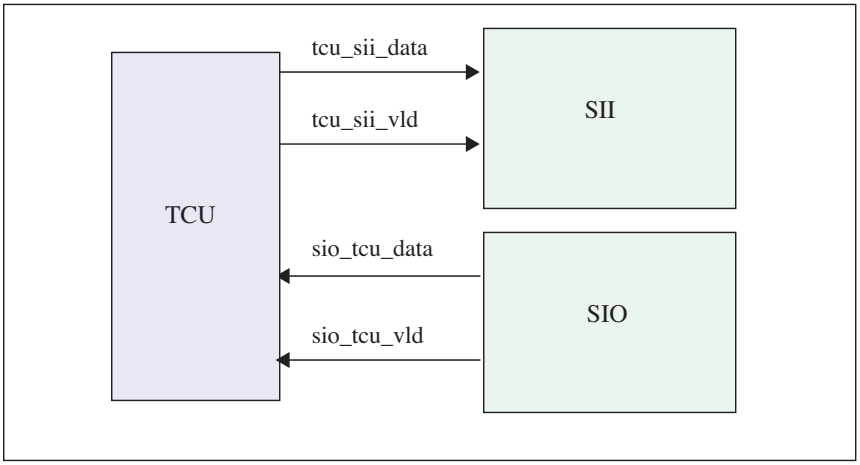
Further details on the Addr (Header) and Data (Payload) can be found in the SOC RAS specification. Only one write or read may be outstanding at any time. Also, since non-JTAG logic is used the POR reset sequence should be performed before using this feature (or at least the POR1 section of the reset sequence).

4.4.4 Diagram

The Signals used between TCU and SIU are:

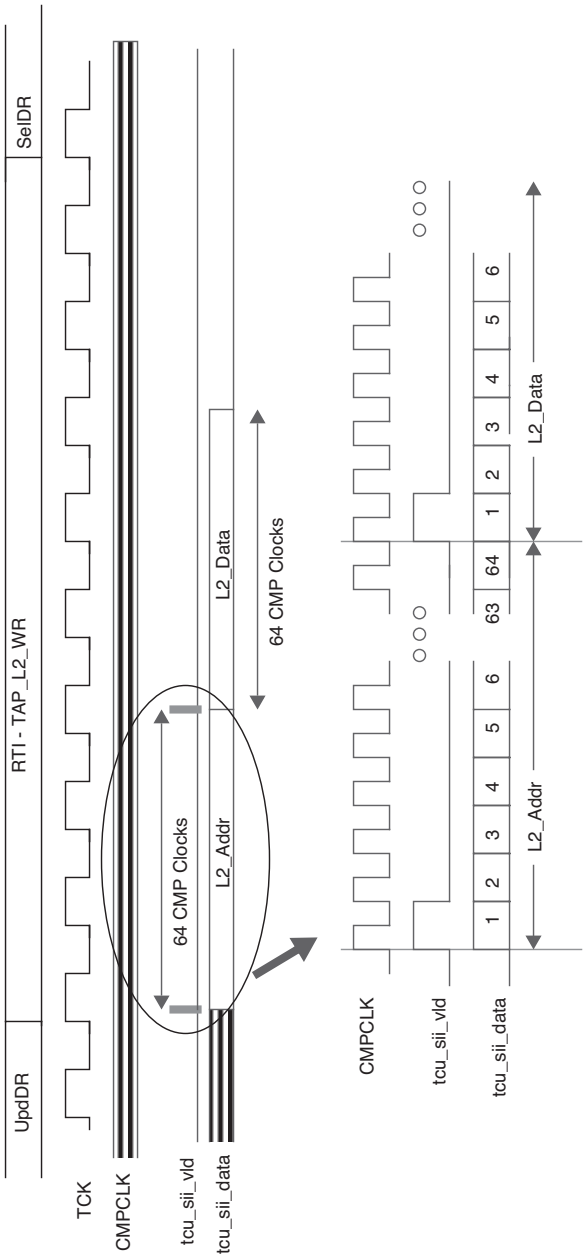
- tcu_sii_data - Sends L2_Addr[64:1] as address followed by L2_Write_Data[64:1] as data to SIU (Data only present for Write, absent for Read). Bit 1 is sent first for both address and data. Output from TCU.
- tcu_sii_vld - Pulsed when bit 1 of L2_Addr or L2_Write_Data goes onto tcu_sii_data. Output from TCU.
- sio_tcu_data - Input to TCU containing data returned from a Read request, bit 0 first.
    - sio_tcu_vld - Input to TCU, pulsed when bit 0 is on sio_tcu_data

FIGURE 4-3 TCU Interface with SIU



A sample waveform is shown in FIGURE 4-4.

FIGURE 4-4 JTAG Write to L2 via SIU - Waveform



4.5 Scan

4.5.1 Manufacturing Scan

There are 32 scan chains in the chip available in parallel during manufacturing (ATPG) scan mode. In manufacturing scan mode they will be accessed through the 32 scan in and 32 scan out pins. These 64 pins will be shared with functional pins; the dedicated testmode pin is used to configure the pins as scan inputs and scan outputs.

Clocks in this mode are provided by the tester (the PLL is not used) but will be multiplexed onto the clock domain trees outside of the TCU, under control of the pll_bypass and testmode pins. In order to allow tester control of clock domains individually during ATPG test, most clock domains will have separate pin control. Serdes is a special issue; the serdes logic will have some of their configuration Signals sourced from the pins. Also, even though the TCU might provide a scan clock the serdes will still need 2 test clocks from the pins for transmit and receive timing information in the manufacuring test modes that TI requires.

TCU participates passively in manufacturing scan; during manufacturing scan the JTAG logic and the TCU itself is included in one of the 32 scan chains and is testable via ATPG patterns.

TABLE 4-7 Manufacturing Parallel Scan Chains

Chain	Contents	TCU Input	TCU Output
0	SPC 0 internal chain 0	spc0_tcu_scan_in[0]	tcu_spc0_scan_out[0]
1	SPC 0 internal chain 1	spc0_tcu_scan_in[1]	tcu_spc0_scan_out[1]
2	SPC 1 internal chain 0	spc1_tcu_scan_in[0]	tcu_spc1_scan_out[0]
3	SPC 1 internal chain 1	spc1_tcu_scan_in[1]	tcu_spc1_scan_out[1]
4	SPC 2 internal chain 0	spc2_tcu_scan_in[0]	tcu_spc2_scan_out[0]
5	SPC 2 internal chain 1	spc2_tcu_scan_in[1]	tcu_spc2_scan_out[1]
6	SPC 3 internal chain 0	spc3_tcu_scan_in[0]	tcu_spc3_scan_out[0]
7	SPC 3 internal chain 1	spc3_tcu_scan_in[1]	tcu_spc3_scan_out[1]
8	SPC 4 internal chain 0	spc4_tcu_scan_in[0]	tcu_spc4_scan_out[0]
9	SPC 4 internal chain 1	spc4_tcu_scan_in[1]	tcu_spc4_scan_out[1]
10	SPC 5 internal chain 0	spc5_tcu_scan_in[0]	tcu_spc5_scan_out[0]

TABLE 4-7 Manufacturing Parallel Scan Chains

Chain	Contents	TCU Input	TCU Output
11	SPC 5 internal chain 1	spc5_tcu_scan_in[1]	tcu_spc5_scan_out[1]
12	SPC 6 internal chain 0	spc6_tcu_scan_in[0]	tcu_spc6_scan_out[0]
13	SPC 6 internal chain 1	spc6_tcu_scan_in[1]	tcu_sp6_scan_out[1]
14	SPC 7 internal chain 0	spc7_tcu_scan_in[0]	tcu_spc7_scan_out[0]
15	SPC 7 internal chain 1	spc7_tcu_scan_in[1]	tcu_spc7_scan_out[1]
16	CCX[0], SII	soca_tcu_scan_in	tcu_soca_scan_out
17	CCX[1], MCU0	socb_tcu_scan_in	tcu_socb_scan_out
18	MCU 1:2, SIO	socc_tcu_scan_in	tcu_socc_scan_out
19	DMU	socd_tcu_scan_in	tcu_socd_scan_out
22	NCU, MCU3	socg_tcu_scan_in	tcu_socg_scan_out
23	L2B 0:7	soch_tcu_scan_in	tcu_soch_scan_out
24	L2T 0:1, L2D 0:1	soc0_tcu_scan_in	tcu_soc0_scan_out
25	L2T 2:3, L2D 2:3	soc1_tcu_scan_in	tcu_soc1_scan_out
26	L2T 4:5, L2D 4:5	soc2_tcu_scan_in	tcu_soc2_scan_out
27	L2T 6:7, L2D 6:7	soc3_tcu_scan_in	tcu_soc3_scan_out
30	TCU, DB1, DB0, MIO, EFU, RST, CCU, BScan	soc6_tcu_scan_in	tcu_soc6_scan_out
31	SerDes Macros	srd_tcu_atpgq	tcu_srd_atpgd

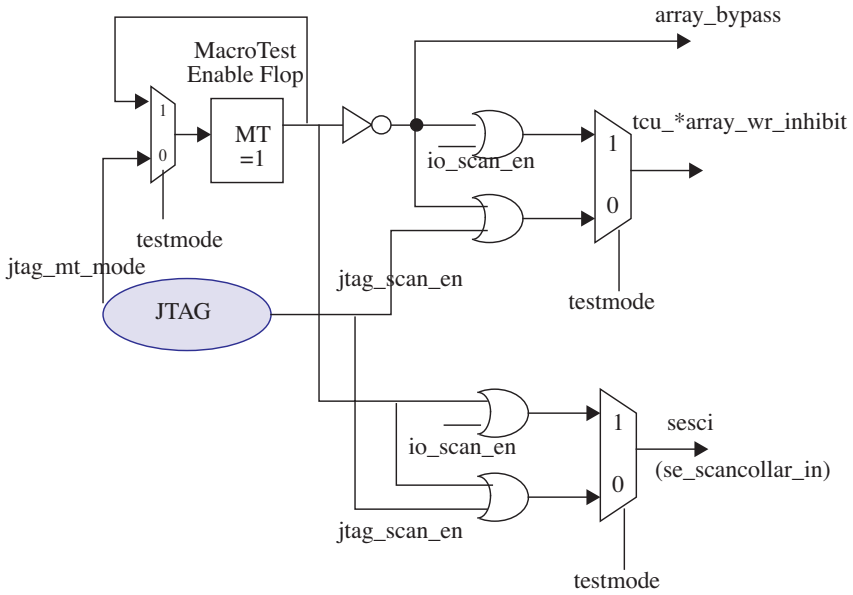
Clusters are ordered as shown with scan-in feeding the left-most block. Boundary scan cells inside the MIO and MCU clusters are included in chain 30. The BScan (boundary scan) chain is ordered from scan-in (TDI) to scan-out (TDO) as MCU0-MCU1-MCU3-MCU2-MIO.

4.5.2 MacroTest Scan

MacroTest on OpenSPARC T2 is primarily a subset of manufacturing scan. During MacroTest mode, control of the array write inhibit signal, scan enables for scan collar input (sesci) flops for arrays, and the array_bypass Signals is different than non-macrotest scan. This additional control allows arrays to be accessed and written to or read from via scan. To enable MacroTest mode, a control flop for array write inhibit, sesci and array_bypass must be set. For manufacturing scan this may be controlled with cell constraints.

MacroTest on OpenSPARC T2 is also be a subset of serial (JTAG) scan. To set JTAG MacroTest mode, the instruction TAP_MT_ACCESS should be programmed; this will set the MacroTest enable flop (default is off). The instruction TAP_MT_SCAN can then be used to perform MacroTest scan accesses. To clear the MacroTest enable flop, use TAP_MT_CLEAR. This mode exists solely to satisfy debug requirements for scan access to arrays.

FIGURE 4-5 Signals Controlled for Macrotest (in TCU)



JTAG MacroTest is used extensively in debug to access the arrays, and to allow control using JTAG the PLL is bypassed to allow TCK to be placed onto the clock tree during MacroTest mode. Before entering JTAG MacroTest mode the clocks to the chip should be stopped via a hard stop since TCK will need to be routed onto the gclk distribution. JTAG MacroTest will access all clock domains, there is no user control over individual domains.

Procedure for Entering JTAG MacroTest

Because the JTAG MacroTest must be run with the PLL locked, a special sequence is used to enter this mode. This sequence puts the chip in JTAG MacroTest mode while not disrupting the CCU (PLL), TCU or RST blocks.

1. Lock PLL : POR sequence (optional; can run in pll bypass mode with slow clock)

2. Run Diag (optional)
3. Stop Clocks : TAP_CLOCK_HSTOP or via Debug Event
4. Set Test Protect : TAP_TP_ACCESS
5. Set Macro Test Mode : TAP_MT_ACCESS
6. Set Chain Select if desired : TAP_CHAINSEL
7. Perform JTAG SCAN : TAP_MT_SCAN

4.5.3 Serial Scan

Serial scan refers to concatenating the 32 internal scan chains into a single chain primarily for observation during debug. Serial scan is initiated via JTAG instructions, where the scan chains are configured into a single long chain and placed between TDI and TDO.

Two JTAG instructions are available to control serial scan; the TAP_SERSCAN instruction places all 32 scan chains between TDI and TDO excluding the TCU, CCU and RST blocks. The chains will be concatenated in the order specified in Table 4-7 on page 26. One of the 32 scan chains may also be selected via TAP_CHAINSEL; See “Chain Select Register” on page 31

During serial scan the scan clocks (aclk, bclk) are generated from the leading and trailing edges of TCK during ShiftDR. The scan enable signal drives the l1clk to ‘1’; prior to unloading a scan chain with JTAG the clock should be stopped to that chain’s clock domain(s) using the JTAG clock stop instructions. The PLL is locked and running typically during serial scan but serial scan does not rely on the PLL. There is no ability to perform capture clocks during serial scan. A sample waveform is shown in FIGURE 4-6.

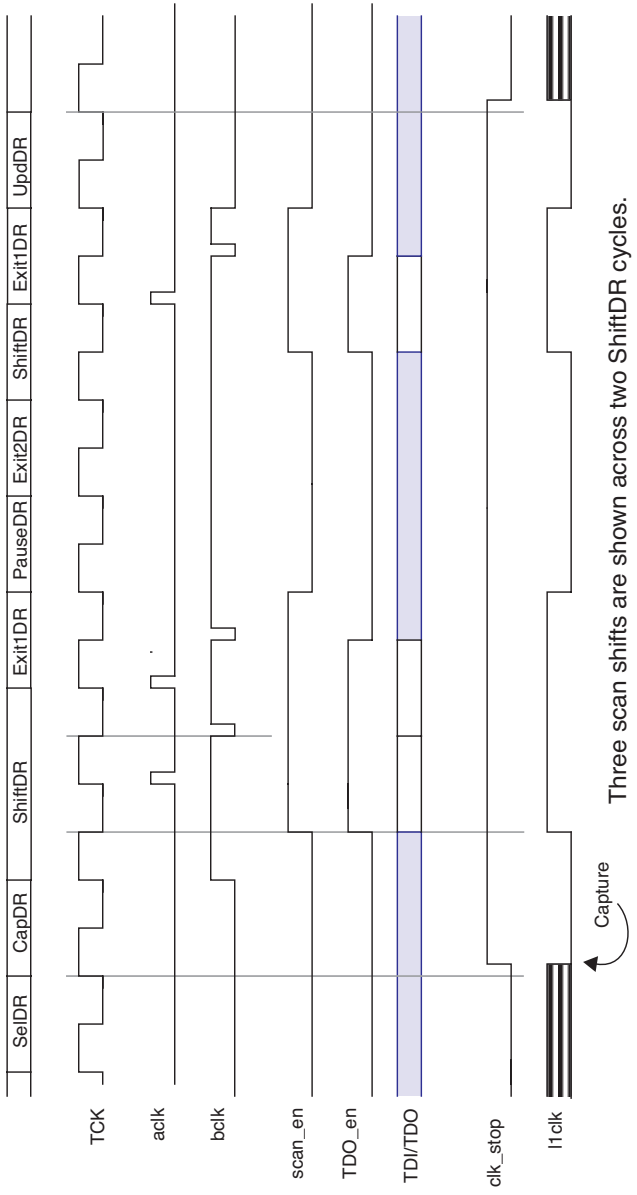

SelDR
CapDR
ShiftDR
Exit1DR
PauseDR
Exit2DR
ShiftDR
Exit1DR
UpdDR
TCK
aclk
bclk
scan_en
TDO_en
TDI/TDO
clk_stop
l1clk
Capture
Three scan shifts are shown across two ShiftDR cycles.

FIGURE 4-6 JTAG Serial Scan Sample Waveform

4.5.3.1 Chain Select Register

In order to observe and update the state of OpenSPARC T2, parallel scan chains 0 to 31 can be put between TDI and TDO by programming the TAP_SERSCAN instruction. The behavior is qualified by the Chain Select register. The internal scan register consists of the chains specified in Table 6-1 on page 22. This is a 6-bit register selected via the TAP_CHAINSEL instruction. It is only recognized when the TAP_SERSCAN (or TAP_MT_SCAN) instruction is programmed and allows one of 32 scan chains in OpenSPARC T2 to be selected with all others bypassed if the msb is set to '1'. If the msb (bit 5) is 0 then the chain selection field is ignored and chains

0 to 30 are concatenated during the serial scan operation. This register has no effect

TABLE 4-8 Chain Select Register

Enable Bit	Chain Selection Field
bit [5]	bits [4:0]
Enables the chain selection field when set to '1'	If enabled, specifies one of 31 chains to be placed between TDI and TDO
	x_xxxx -> Selects chains 0-30 concatenated
0	
	0_0000 -> Selects chain 0 & 1 of SPC0
1	0_0001 -> Selects chain 0 & 1 of SPC0
1	0_0010 -> Selects chain 0 & 1 of SPC1
1	0_0011 -> Selects chain 0 & 1 of SPC1
1	0_0100 -> Selects chain 0 & 1 of SPC2
1	0_0101 -> Selects chain 0 & 1 of SPC2
1	0_0110 -> Selects chain 0 & 1 of SPC3
1	0_0111 -> Selects chain 0 & 1 of SPC3
1	0_1000 -> Selects chain 0 & 1 of SPC4
1	0_1001 -> Selects chain 0 & 1 of SPC4
1	0_1010 -> Selects chain 0 & 1 of SPC5
1	0_1011 -> Selects chain 0 & 1 of SPC5
1	0_1100 -> Selects chain 0 & 1 of SPC6
1	0_1101 -> Selects chain 0 & 1 of SPC6
1	0_1110 -> Selects chain 0 & 1 of SPC7
	0_1111 -> Selects chain 0 & 1 of SPC7
1	
1	1_0000 -> Selects chain 16
1	1_0001 -> Selects chain 17
1	1_0010 -> Selects chain 18
1	1_0011 -> Selects chain 19
1	1_0100 -> Selects chain 20
1	1_0101 -> Selects chain 21
1	1_0110 -> Selects chain 22
1	1_0111 -> Selects chain 23
1	1_1000 -> Selects chain 24
1	1_1001 -> Selects chain 25
1	1_1010 -> Selects chain 26
1	1_1011 -> Selects chain 27
1	1_1100 -> Selects chain 28
1	1_1101 -> Selects chain 29
1	1_1110 -> Selects chain 30
	1_1111 -> Ignored

on chain 31 which is reserved for SERDES scan flops, which means that chain 31 is not accessible via jtag serial scan. Scan chains for Boundary Scan, TCU, CCU and RST units are not accessible via jtag serial scan; when chain_sel[4:0]=30 only the peu, mio, efu, db0 and db1 scan flops will be returned.

The Chain Select register is reset with TRST_L or entering TLR. The chain selection field is directly decoded to specify a chain from 0 to 30. Chain 31 is not selectable. Selecting either chain in a SPC core will result in both SPC scan chains being concatenated. The MBIST and shadow scan chains in each SPC will be concatenated to that SPC's chain[0] and chain[1], respectively.

Note – The length of the scan chain during JTAG serial scan will change between POR1/POR2 and WMR1/2. The chain will be longer during POR1/POR2 due to inclusion of MCU logic as shown in the next subsection "Logic Included in JTAG Serial Scan". Use of JTAG POR (See "JTAG Access During POR" on page 90) pauses the POR sequence during POR2 and the MCU logic will be included during JTAG serial scan. A subsequent warm reset will move the chip out of POR2 and cause the MCU FBD logic to be excluded from JTAG serial scan.

Note – Unpredictable behavior will result if you do a JTAG (tap) reset after scandumping any of the soc chains (this includes the "all chain" scan dump mode) since the TAP_TP_ACCESS command will be reset (See "Protecting TCU During Serial Scan: Test Protect Mode" on page 34). This will expose the logic protected by TAP_TP_ACCESS to any random data scanned during the SOC scan process, with indeterminate results.

4.5.3.2 Logic Included in JTAG Serial Scan

During jtag serial scan this logic is included:

- MBIST engines
- MCU FBD logic (during JT POR access in POR2)
- shadow scan - both spc & l2t
- Unavailable SPC cores and Banks

This logic is NOT included during jtag serial scan:

- cluster headers
- MCU FBD logic (when not in POR2 - i.e., during diag scan dumps)
- boundary scan flops
- any flops non-scannable for manufacturing scan
- CCU, RST and TCU

- SerDes (chain 31)

4.5.3.3 Protecting TCU During Serial Scan: Test Protect Mode

When JTAG serial scan is performed, random Signals can be generated to TCU inputs. If the TCU responds to these they can disrupt the JTAG serial scan; for example random debug requests from a SPC while it is being scanned can disrupt the TCU jtag scan process. To protect against this it is up to the user to tell TCU to protect itself. Two jtag instructions are available for this: TAP_TP_ACCESS to set the Test Protect mode, and TAP_TP_CLEAR to clear it. Setting the Test Protect mode will cause TCU to assert a signal tcu_test_protect which will block incoming SPC debug requests and incoming UCB requests. This signal also goes to RST and CCU and other blocks which need to block random UCB requests which may occur when scanning the SOC blocks (specifically NCU). This mode is also needed during MBIST scan operations, and possibly LBIST scan operations. The expected usage is to set the Test Protect mode before performing the test operations, and then clear it when done. For Transition Test and Macro Test the mode should be set via scan operations if needed.

Setting Test Protect mode should not interfere with PLL lock, but it may interfere with diags trying to change clock frequency or generate resets.

Note – When accessing any scan chains via JTAG mode, the TAP_TP_ACCESS protocol should be followed. This includes TAP_MBIST_DIAG and TAP_LBIST_ACCESS.instructions as well as variations of TAP_SERSCAN or TAP_MT_SCAN.

4.5.4 SerDes Scan

TCU supports scan for Serdes by connecting the SerDes macros onto chain 31. Package pin SCAN_IN31 becomes tcu_srd_atpgd and connects to ATPGD of fsr0, and then ATPGQ and ATPGD are connected to daisy-chain the SerDes macros, with ATPGQ of fsr4 connecting through TCU tsrd_tcu_atpgq to SCAN_OUT31. The scan-enable signal for SerDes is tcu_srd_atpgse and is driven by TCU from package pin io_scan_en. The following outlines the mode under which SerDes operates; bits [1:0] are accessible only via scan and bit [2] is driven directly from io_test_mode. The atpg mode bus tcu_srd_atpgmode[2:0] ultimately connects to the TESTCFG[18:16] of the SerDes macros.

- tcu_srd_atpgmode[2:0]
  - 000: for normal operation
  - 001, 010, 011: reserved

- 100: for stuck-at atpg
- 101: to select 2-clock transition test
- 110: to select 3-clock transition test
- 111: to select 4-clock transition test

4.6 Clock Stop

On OpenSPARC T2 the ability to stop clocks to various sections of the chip is provided via the TCU. Clocks can be stopped via JTAG directly or as a result of a debug or other event. Clocks are also stopped before any flush reset and then restarted after the flush reset is finished.

There are 2 modes of clock stops: a hard clock stop and a soft clock stop. The purpose of the hard clock stop is to stop as fast as possible, without waiting for the chip to become quiet. The 2nd method, soft clock stop, only applies to the cores and upon receiving a request the TCU will wait for the requesting core to settle into a quiescent state (via the core_running register) before stopping the clock to that core. Multiple cores may also be stopped this way. This allows the core(s) the possibility to restart after clocks are restarted. Clocks for the chip can be stopped either in parallel or serially across clock domains. After a clock stop, data can then be shifted out for debug via JTAG which allows the user to determine the state of the chip.

4.6.1 Serial and Parallel Clock Stop Modes

Stopping all clock domains in parallel may not be advisable due to excessive current fluctuations across the chip. Because of these di/dt concerns there is a serial clock stop mode where the clocks are stopped over several predefined clock domains with 128 cpu clock cycles between each clock stop activation. Stopping the clocks in such a staggered fashion with intervening delays is expected to lessen the di/dt concern. In the serial mode, via JTAG or software the user can update a clock domain register to specify which clock domain should be stopped first. Subsequent domains will then be stopped in a predetermined order, but the order is fixed.

During a parallel clock stop, the clocks will all be stopped relative to the same cpu clock cycle from the TCU. For both the serial and parallel clock stop methods, due to division ratios between the cpu and other clock domains, the actual cpu clock cycle at which a non cpu clock domain stops may vary between those domains, although it should be repeatable. To specify a parallel stop, all bits in the clock domain register should be set to 1, signifying they should all stop first.

Specification of serial or parallel is controlled by setting the 32-bit Clock Domain register with JTAG TAP_CLOCK_DOMAIN instruction, ordered as specified in Table 4-9 (bit == stop number). Setting only one bit indicates the starting point for serial stopping. If serial and parallel clock stop modes are mixed, that is multiple bits are set in the clock_domain register, the clocks will stop in both serial and parallel across the specified bit fields. Originally tcu only supported either sequential or parallel without mixed modes but flexibility was given to allow the modes to work together. This results in a mixed behavior, with all bits set to '1' stopping in parallel, but with sequential stop behavior across the remaining fields of '0' bits. The user should consider this when programming mixed serial and parallel clock stopping.

Because the ability to stop selected domains in parallel would mainly be used for scan dump purposes, it doesn't matter if the remaining bits stop sequentially as described since the object of the scan dump should be in the domains that are set to stop in parallel.

4.6.2 Hard Clock Stop

A hard clock stop request will result in the clocks being stopped without waiting for the chip to quiesce. The clocks may be stopped either in serial or parallel mode. In all cases the clocks will be stopped over all the chip as specified by the Clock Domain register, except the RST, CCU and TCU clocks will not be stopped.

A hard clock stop may be initiated in response to a flush request, a specific JTAG request via TAP_CLOCK_HSTOP, or in response to a debug event. A status register can be polled via TAP_CLOCK_STATUS to determine the state of the clock sequencer. When the status indicates clocks are stopped, a scan dump via TAP_SERSCAN can be done. To restart clocks, the JTAG TAP_CLOCK_START is used.

4.6.3 Soft Clock Stop

A soft clock stop request will not be serviced until the core requesting the soft clock stop is quiesced. The cores are the only clusters that can request a soft clock stop, and only the clocks to the target cores will be stopped by any soft stop request. A soft clock stop may be initiated in response to a JTAG request via TAP_CLOCK_SSTOP or in response to a debug event. Via JTAG, multiple cores can be soft-stopped. A debug event can only stop a single core as a default, however, setting bit 3 of the TCU DCR causes the TCU to soft stop all enabled SPCs if any requests a soft stop; See "TCU Debug Control Register" on page 61.

To request a soft clock stop via JTAG the target core(s) should first be parked with UCB access to the 64-bit Core Run register; the user is responsible for setting all 8 bits per each core to be parked to '0'. When status is indicated via TAP_CORE_RUN_STATUS that the target threads are parked (64-bits), the clocks may be stopped via TAP_CLOCK_SSTOP.

For example, to quiesce and stop SPC cores 2 and 5 only, first check with TAP_CORE_RUN_STATUS and then set the Core Run register to 64'hkkkk 00kk kk00 kkkk. This sets bits 23:16 for core 2, and 47:40 for core 5 to all 0's; 'k' means keep previous value read with TAP_CORE_RUN_STATUS. Check that the target threads are parked with TAP_CORE_RUN_STATUS. Set the target cores that should respond to a clock stop using TAP_CORE_SEL (in this case, cores 2 and 5) to set the core select register; any subsequent TAP_CLOCK_SSTOP will only stop clocks to cores 2 and 5. Note that TAP_CLOCK_START will clear the core select register.

In response to a debug event the requesting core's clock will be stopped similarly.

When the clocks are stopped a status register is set indicating the clocks are stopped. Polling of this status register can be done with TAP_CLOCK_STATUS to determine when it is safe to do a subsequent scan dump of the stopped cores. To restart the clock to the target core, the JTAG TAP_CLOCK_START is used.

The TAP_CORE_SEL instruction allows the user to enable cores to respond to a soft clock stop JTAG request using TAP_CLOCK_SSTOP and assumes all cores were already quiesced. If it is used without quiescing the cores you will get in effect a hard stop across only the cores. See the section "Cycle Step Mode" on page 57 for usage of TAP_CORE_SEL.

When using the soft stop mode, the Clock Domain register should be all 0s if TCU DCR bit [2] is '1'. This applies to JTAG TAP_CLOCK_SSTOP and a request for Soft Stop by spc debug event when tcu_dcr[3] is set to 1 to stop all cores. In general, TCU DCR bit [2] should be '0' if any Soft Stop is used, otherwise the interaction between the Clock Domain and Core Select registers is complex.

The TCU clock sequencer - described in the next section - is controlled directly by the Clock Domain register. The Core Select register has no effect on the clock sequencer inside TCU, it will sequence independently of Core Select. The sequencer always runs through all 24 clock domains.

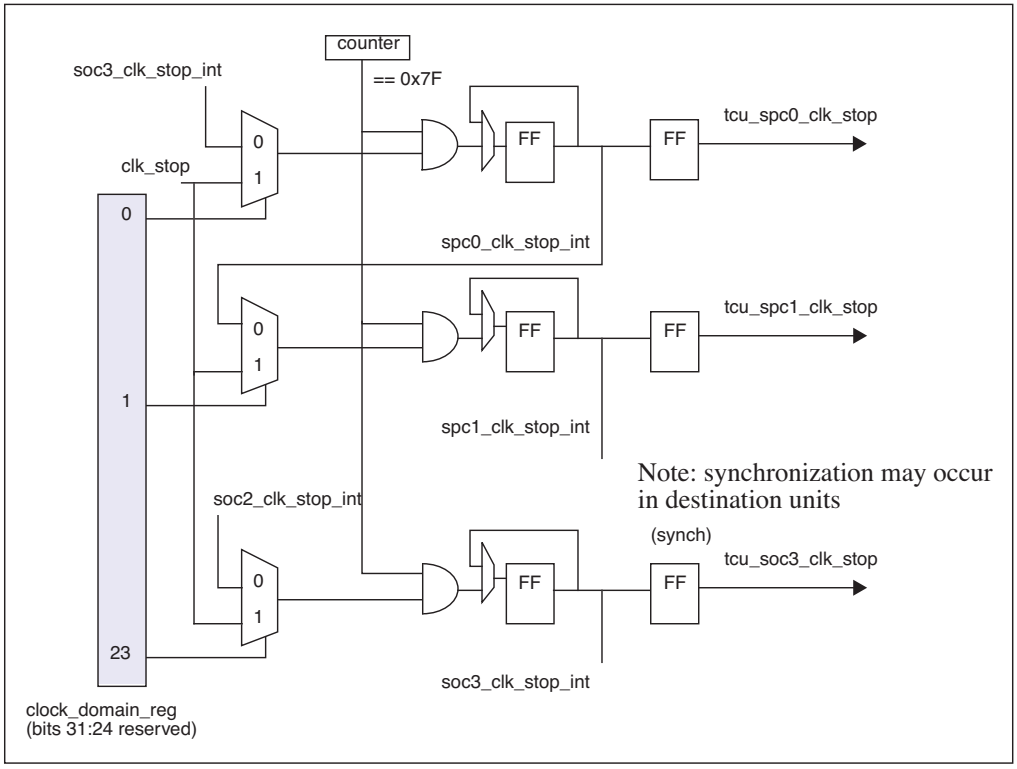
In soft stop mode, only the 8 outputs associated with the spc cores are allowed to propagate. When Core Select is set and TAP_CLOCK_SSTOP issues, this begins the clock sequencer and it should be in a default mode starting with spc0 - this is either with Clock Domain all zeros or 24'b1. If the Clock Domain is something else, then it will tell the clock sequencer to begin at a different starting point, or if multiple bits are set the sequencer will stop clocks in parallel for those bits. The cores will still be stopped, but in an unexpected order.

For example, if clock_domain = 24'h000003 then spc 1 and spc 2 will be stopped in parallel and spc2 will be the starting point of the sequence - so spc0 will be stopped last. If clock_domain = 24'h000083, then spc7, spc1 and spc2 will stop in parallel.

4.6.4 Stop Domains

Clock domains are partitioned so that control is achieved for disabling sections of the chip with respect to the L2 and Core Enable/Available registers, and to minimize di/dt. The sequence of stopping the clocks serially will always be the same given a specific start point and defaults to the order given in Table 4-9 on page 40. The user can program the starting point, but then the domains will stop in the predetermined order and wrap around until reaching the first domain stopped. For instance, stopping with spc1 first will result in spc0 being stopped last.

FIGURE 4-7 TCU Clock Sequencer



A 7-bit Clock Stop Delay counter (programmable via TAP_CLKSTP_DELAY) provides a delay of up to 128 (default) cmp clock cycles between generation of successive clock stop Signals from the TCU. Setting this value to zero results in a 1-cmp clock cycle delay between clock stop Signals. This may be bypassed by setting bits [23:0] in the Clock Domain register via JTAG, so that all clocks stop in parallel.

The general structure of the clock sequencer control logic in the TCU is shown in FIGURE 4-7. To stop clocks starting with spc0, bit 0 of Clock Domain register is set to ‘1’ with the remaining bits all to ‘0’. The clock sequencer state machine is in an initial state because clk_stop is low ‘0’, then the clk_stop signal is activated and held to ‘1’ to begin the sequence. When the Clock Stop Delay counter reaches 0xFF the tcu_spc0_clk_stop is set to ‘1’ and held; when the counter next reaches 0xFF the tcu_spc1_clk_stop is set to ‘1’ and the machine continues this sequence until all clock domains are disabled. When the clk_stop signal is driven to ‘0’ the clk_stop Signals are sequenced off starting with the domain specified in Clock Domain register.

The Clock Domain register is shown in the following table.

TABLE 4-9 Clock Domain Register

Stop Number	Clock Domain Controlled	Stop Number	Clock Domain Controlled
0	SPC 0: cmp clock domain	12	Bank 4: L2 T, D, B : cmp clock domain
1	SPC 1: cmp clock domain	13	Bank 5: L2 T, D, B : cmp clock domain
2	SPC 2: cmp clock domain	14	Bank 6: L2 T, D, B : cmp clock domain
3	SPC 3: cmp clock domain	15	Bank 7: L2 T, D, B : cmp clock domain
4	SPC 4: cmp clock domain	16	MCU 0: cmp and io clock domains
5	SPC 5: cmp clock domain	17	MCU 1: cmp and io clock domains
6	SPC 6: cmp clock domain	18	MCU 2: cmp and io clock domains
7	SPC 7: cmp clock domain	19	MCU 3: cmp and io clock domains
8	Bank 0: L2 T, D, B : cmp clock domain	20	SOC0: sii, sio, ncu, efu: cmp and io clock domains. ccx; cmp clock domain. db0, db1, mio: io clock domain
9	Bank 1: L2 T, D, B : cmp clock domain	21	SOC1: rdp, mac, rtx, tds io and io2x clock domains
10	Bank 2: L2 T, D, B : cmp clock domain	22	SOC2: dmu: io clock domain
11	Bank 3: L2 T, D, B : cmp clock domain	23	SOC3: peu: io and pc clock domains

All clock stop control logic in the TCU is in the cmp clock domain. The outgoing Signals, tcu_*_clk_stop, are sent from the cmp clock domain and are staged at the cpu level before reaching the cluster headers. The cluster header synchronizes the clk_stop into the corresponding clock domain. For clusters with io or io2x clock domains, the tcu_*_clk_stop is synchronized to the io clock domain before leaving TCU. This is done to provide transition test the capability of controlling the clock stop relative to the target domain. The dr clock domain clock stops are synchronized into the dr clock domain before leaving TCU, to mesh with the top-level dr staging flops.

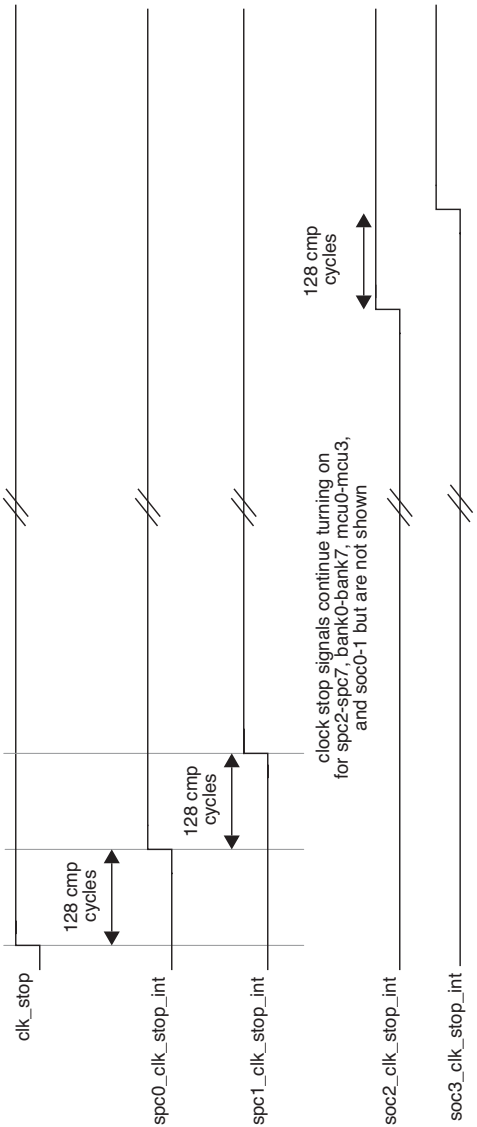
Most clusters with both cmp and io clock domains have separate clock stop Signals from TCU, one for each domain. In some cases clusters with multiple clock domains share a single clock stop. The DB0 and DB1 clusters have both cmp and io domains, but share a clock stop synchronized to the io clock domain. The same holds for MIO. The RDP, RTX and TDP clusters have both io and io2x but share a clock stop synchronized to the io clock domain. The effect of this is that for clock stopping, the logic in these clusters will stop at different clock cycles. For example, in MIO during a clock stop the cmp logic will stop 3 (cmp) cycles later than other cmp logic.

- tcu_db0_clk_stop and tcu_db1_clk_stop => each is connected to cmp and io headers

- tcu_rdp/rtx/tdp_io_clk_stop => each is connected to io and io2x headers
- tcu_mio_clk_stop => connected to 4 cmp headers and one IO header

Clocks are restarted by turning off clk_stop Signals. When started serially, the 128 cmp cycle delay is used again to reduce di/dt concerns.

FIGURE 4-8 Clock Stop Sequencing through Clock Domains



4.6.5 FBD Logic in MCU

The FBD logic in the MCUs is handled differently from other SOC logic. A separate clock stop signal is provided to each MCU, tcu_mcu[0123]_fbd_clk_stop, which is activated only during POR1 and POR2 (to facilitate flush reset of the FBD logic) or if

the MCU is disabled via bank available or bank enable. During jtag serial scan, the FBD logic is bypassed and left running. This is achieved with a second shared signal to all 4 MCUs, tcu_mcu_testmode, which is '1' only during POR1, POR2, or manufacturing atpg testing.

If jtag serial scan is performed while in POR2 then the MCU FBD logic will be included, so during the JTPOR access window (See "JTAG Access During POR" on page 90) the jtag serial scan length will be longer than after POR2 completes.

4.6.6 Clock Stopping and Core/L2 Available and Disable Controls

4.6.6.1 Core and L2 Available Control

SPC cores are made unavailable if these Signals are not asserted after transfer from EFU after POR1 or POR2:

- ncu_spc0_core_available
- ncu_spc1_core_available
- ncu_spc2_core_available
- ncu_spc3_core_available
- ncu_spc4_core_available
- ncu_spc5_core_available
- ncu_spc6_core_available
- ncu_spc7_core_available

L2 Logic (L2 Tag, L2 Data, L2 Buffer) can be made unavailable if the corresponding bits in this bus are not asserted after transfer from EFU after POR1 or POR2. Note that the L2 Tags will not have their clocks stopped even if listed as unavailable or disabled.

- ncu_tcu_bank_avail[7:0]

4.6.6.2 Core and L2 Disabling Control

SPC Cores can be disabled via Software after a warm reset with these Signals:

- ncu_spc0_core_enable_status
- ncu_spc1_core_enable_status
- ncu_spc2_core_enable_status
- ncu_spc3_core_enable_status

- ncu_spc4_core_enable_status
- ncu_spc5_core_enable_status
- ncu_spc6_core_enable_status
- ncu_spc7_core_enable_status

L2 Banks (Two L2 Data and L2 Buffers along with associated MCU) can be disabled via Software after a warm reset with these Signals:

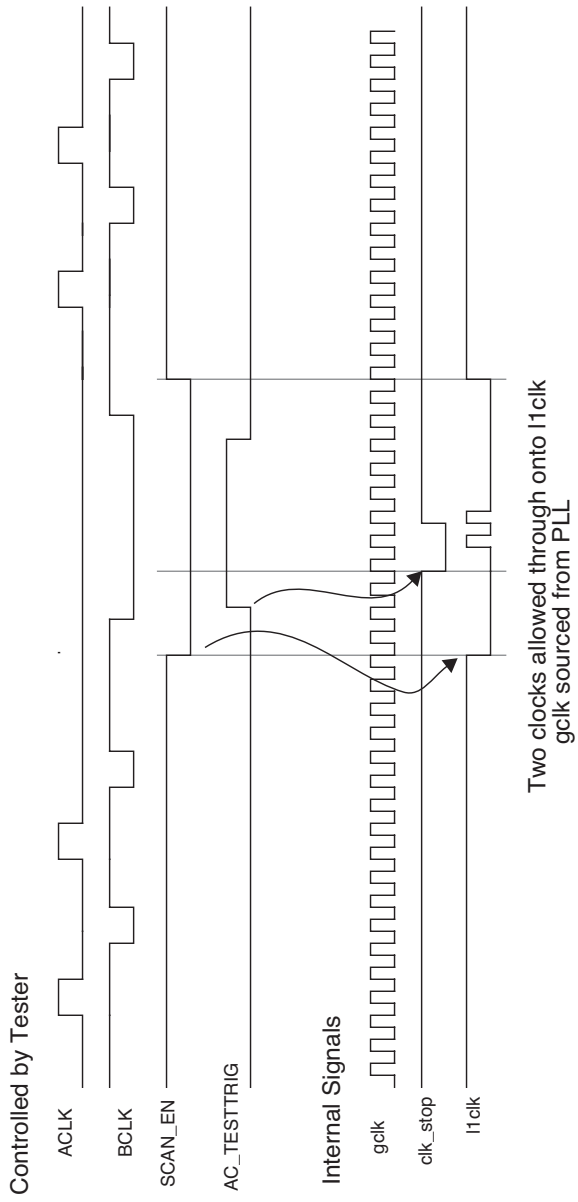
- ncu_spc_pm
- ncu_spc_ba01
- ncu_spc_ba23
- ncu_spc_ba45
- ncu_spc_ba67

There are certain legal combinations for the Signals controlling disabling of SPCs and L2 Banks, for details refer to the *OpenSPARC T2 Programmer"s Reference Manual*. Also, JTAG can be used to overwrite these values in certain cases such as using the JTAG POR access window or via JTAG UCB access. TCU looks at the Available Signals after POR1 or POR2 and deasserts clocks to any unavailable SPC or L2 (except L2 Tag clocks are not stopped since they contain top-level staging flops). This is shown in the reset waveform Figure 4-22 on page 88. The Disabling Signals are observed by TCU after any WMR2, and clocks will be deasserted by TCU for any disabled SPC core or L2 bank and associated MCU (except for L2 Tags).

4.7 Transition Testing

Transition test on OpenSPARC T2 is designed to be run with the PLL locked to allow testing of the clock domains for transition faults. The patterns will be generated via the ATPG tools and applied on the wafer and chip testers. The testmode pin and ac_testmode pins must both be driven to '1' by the tester to enable transition test mode. The AC_TESTTRIG pin is used to tell the internal logic to allow a programmed number of l1clk cycles to reach the target flops.

FIGURE 4-9 Transition Test Sample Vector



4.7.1 Operation and Constraints During Transition Test

The transition test logic inside TCU is programmed via scan with a setup routine, or via cell constraints. A counter specifies how many system clock pulses are issued. This counter is based on the clock for the domain that is under test and is loaded with the number of clock pulses that will be issued to the functional logic. Only one clock domain may be active at one time since transition test across clock domains introduces non-determinism, and only the cmp and io clock domains are supported. During transition test the array_wr_inhibit signal from TCU is driven high into the clusters and overrides the write inhibit generated by the cluster headers from clock stop transitions.

When the mio_tcu_io_ac_testtrig input to TCU (package pin AC_TESTTRIG) is driven to '1' by the tester it is synchronized and used to enable the clock stop counter. The clk_stop signal to the target clock domain and cluster(s) is generated by TCU and pipelined out to the target cluster headers, and held to '0' for the programmed number of clock cycles. The counter is 8-bits and allows up to 255 clock pulses to be issued. The transition test control bit used to select the clock domain (cmp or IO) is 1 bit.

- tcusig_ttclksel_reg
  - 0 => selects cmp clock domain (default)
  - 1 => selects io clock domain

The control register, the 8-bit counter, and the flops driving the clock_stop Signals all must be set to the appropriate values before each transition test capture cycle. The 8-bit counter values are true binary representations for cmp clock domain, and should be set to a multiple of 4 for the io clock domain. So to get 2 io clock cycles during the capture phase, the counter should be set to binary 0000_1000. The counter will start counting aligned to the io sync enable pulses during both cmp and io clock domain testing to achieve accurate clock counting during the io clock domain tests. A counter value of zero is not supported for transition test.

To set the clock stop flops in the tcu, the user should scan in values of zero to all clock stop flops, except set a one to flop _0 of all targeted domains. The values are inverted onto the clock stop Signals. There are 24 clock domains as shown in Table 4-9 on page 40. Bit 0 is closest to scan-in. Each clock domain has 2 flops associated with it and MCU and SOC have extra flops for io and dr clock domains. Flops that can be set to activate clock pulses to a domain are indicated. To select an io clock domain, set the corresponding _0 flop and also set the transition test control bit to '1'.

- sync_ff_clk_stop_spc0_0Set to '1' for SPC0 cmp clock domain
- sync_ff_clk_stop_spc0_1
- ...
- sync_ff_clk_stop_spc7_0Set to '1' for SPC7 cmp clock domain

- sync_ff_clk_stop_spc7_1
- sync_ff_clk_stop_bnk0_0Set to ‘1’ for BNK0 cmp clock domain
- sync_ff_clk_stop_bnk0_1
- sync_ff_clk_stop_l2t0_0Set to ‘1’ for L2T0 cmp clock domain
- sync_ff_clk_stop_l2t0_1
- ...
- sync_ff_clk_stop_bnk7_0Set to ‘1’ for BNK7 cmp clock domain
- sync_ff_clk_stop_bnk7_1
- sync_ff_clk_stop_l2t7_0Set to ‘1’ for L2T7 cmp clock domain
- sync_ff_clk_stop_l2t7_1
- sync_ff_clk_stop_mcu0_0Set to ‘1’ for MCU0 cmp or io clock domain
- sync_ff_clk_stop_mcu0_1
- sync_ff_ioclk_stop_mcu0_1
- sync_ff_drclk_stop_mcu0_1
- ...
- sync_ff_clk_stop_mcu3_0Set to ‘1’ for MCU3 cmp or io clock domain
- sync_ff_clk_stop_mcu3_1
- sync_ff_ioclk_stop_mcu3_1
- sync_ff_drclk_stop_mcu3_1
- sync_ff_clk_stop_soc0_0Set to ‘1’ for SOC0 cmp or io clock domain
- sync_ff_clk_stop_soc0_1
- sync_ff_ioclk_stop_soc0_1
- sync_ff_clk_stop_soc1_0Set to ‘1’ for SOC1 io clock domain
- sync_ff_ioclk_stop_soc1_1
- sync_ff_clk_stop_soc2_0Set to ‘1’ for SOC2 io clock domain
- sync_ff_ioclk_stop_soc2_1
- sync_ff_clk_stop_soc3_0Set to ‘1’ for SOC3 io clock domain
- sync_ff_clk_stop_soc3_1(cmp goes to pc clock domain, not supported)
- sync_ff_ioclk_stop_soc3_1

The transition test counter flops are:

- tcuregs_ttcounter_reg[7:0]Set to binary count as described above

In addition, these synchronizer flops should be scanned to ‘00’ so that they do not interfere with the clock stop logic when in transition test mode.

- cpu.tcu.sigmux_ctl.tap_spc7_mb_cs_sync_reg

- ...
- cpu.tcu.sigmux_ctl.jtag_l2t0_ss_cs_sync_reg

These flops should be scanned to 0 also. They are the first stage of pipeline flops on the clock stop Signals as they leave TCU.

- cpu.tcu.clkstp_ctl.clkstp_spc0stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc1stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc2stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc3stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc4stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc5stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc6stop_reg
- cpu.tcu.clkstp_ctl.clkstp_spc7stop_reg
- cpu.tcu.clkstp_ctl.clkstp_bnkstop_reg
- cpu.tcu.clkstp_ctl.clkstp_l2tstop_reg
- cpu.tcu.clkstp_ctl.clkstp_mcustop_reg
- cpu.tcu.clkstp_ctl.clkstp_mcuiostop_reg
- cpu.tcu.clkstp_ctl.clkstp_mcudrstop_reg
- cpu.tcu.clkstp_ctl.clkstp_mcufbdstop_reg
- cpu.tcu.clkstp_ctl.clkstp_soc0stop_reg
- cpu.tcu.clkstp_ctl.clkstp_soc0iostop_reg
- cpu.tcu.clkstp_ctl.clkstp_soc1iostop_reg
- cpu.tcu.clkstp_ctl.clkstp_soc2iostop_reg
- cpu.tcu.clkstp_ctl.clkstp_soc3stop_reg
- cpu.tcu.clkstp_ctl.clkstp_soc3iostop_reg

Note that this listing of flops is intended to be used as a guide only and may not include all flops necessary to implement all variations of transition test.

For IO clock domain, the counter should be set to reflect the desired io clock pulses. For example, to get 2 io clock pulses the counter should be set to 8, for 3 io clock pulses the counter should be set to 12, etc. One or more CMP clock domains can be tested with transition test at the same time; the same is true for IO clock domains. However, CMP and IO clock domains cannot be tested together during transition test, one or the other must be specified. Finally, the dr clock domain cannot be tested with transition test since the dr clock is asynchronous to the logic in TCU.

In mio, db0 and db1 there are both cmp and IO clock domains but a single clock stop is sent as soc0_io_clk_stop, so in transition test these blocks cannot be tested since they use the same clock stop for both cmp and IO cluster headers.

4.7.2 Procedure for Entering Transition Test

Because the transition test must be run with the PLL locked, a special sequence is used to enter the transition test mode. This sequence makes use of the dedicated TDI package pin to put the chip in transition test mode while not disrupting the CCU (PLL), TCU or RST blocks.

1. Lock PLL : POR sequence
2. Stop Clocks : TAP_CLOCK_HSTOP
3. Set Test Protect : TAP_TP_ACCESS
4. Drive TDI to ’1’
5. Drive TEST_MODE to ’1’
6. Drive AC_TEST_MODE to ’1’ (active)
7. SCAN_EN to ’0’ (inactive)
8. ACLK to ’0’ (inactive)
9. BCLK to ’0’ (active)
10. AC_TESTTRIG to ’0’ (inactive)
11. Drive TDI to ’0’ and hold it
12. - this allows (6) Signals to propagate but with values unchanged
13. Enter ATPG sequence - scan_en, aclk, bclk, shifting, etc.
14. - hold TCK low and load JTAG_CTL with safe (all 0) state

Before entering TT mode, test_protect is asserted with TAP_TP_ACCESS so that it doesn’t change when AC_TEST_MODE goes high. Test_protect is OR’d with AC_TEST_MODE so that it is always high during TT, even during shifting, to prevent external Signals from affecting TCU, RST and CCU during TT.

4.7.3 SerDes Transition Test

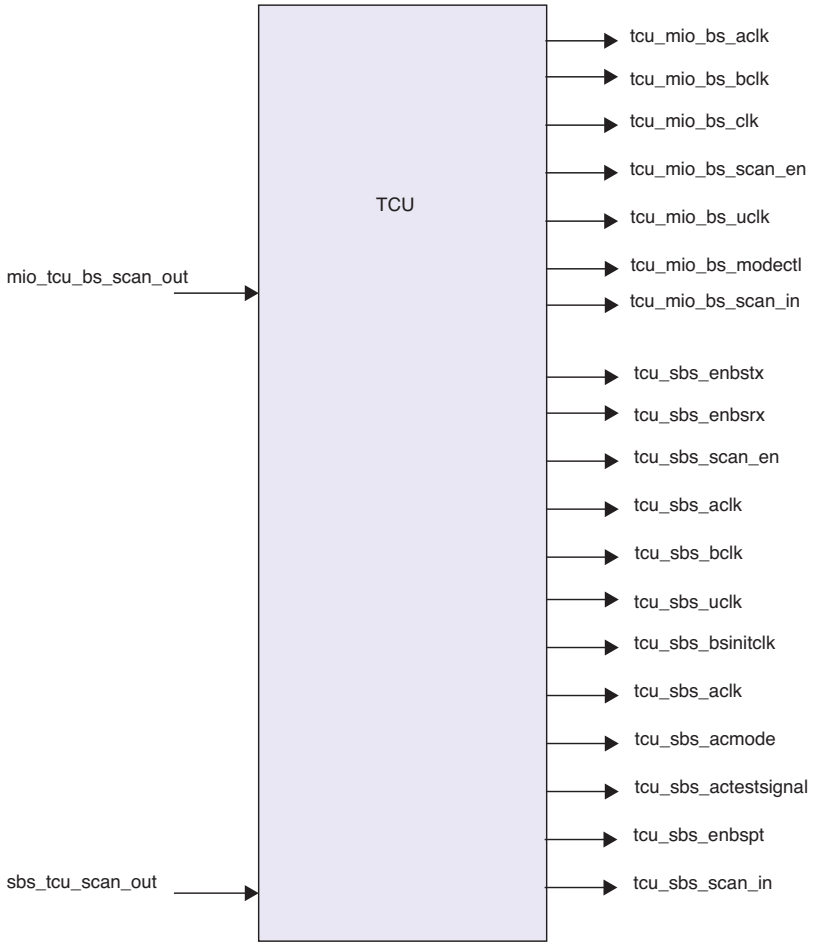
See “SerDes Scan” on page 34.

4.8 Boundary Scan

The TCU has logic to support boundary scan testing, through the use of JTAG instructions. The interface will provide the following JTAG instructions: Sample/Preload, Extest, HighZ, and Clamp for 1149.1 support in the MIO and some SerDes, and Extest_Pulse and Extest_Train for 1149.6 support for SerDes. The boundary scan cells have also been designed such that they will be included as part of the scan chain.

Timing for boundary scan will be similar to JTAG serial scan as shown in FIGURE 4-6 with an additional update clock occuring in the Update-DR state. During manufacturing scan the boundary scan control Signals will be driven from the package pins through multiplexors.

FIGURE 4-10 TCU to Boundary Scan Interface

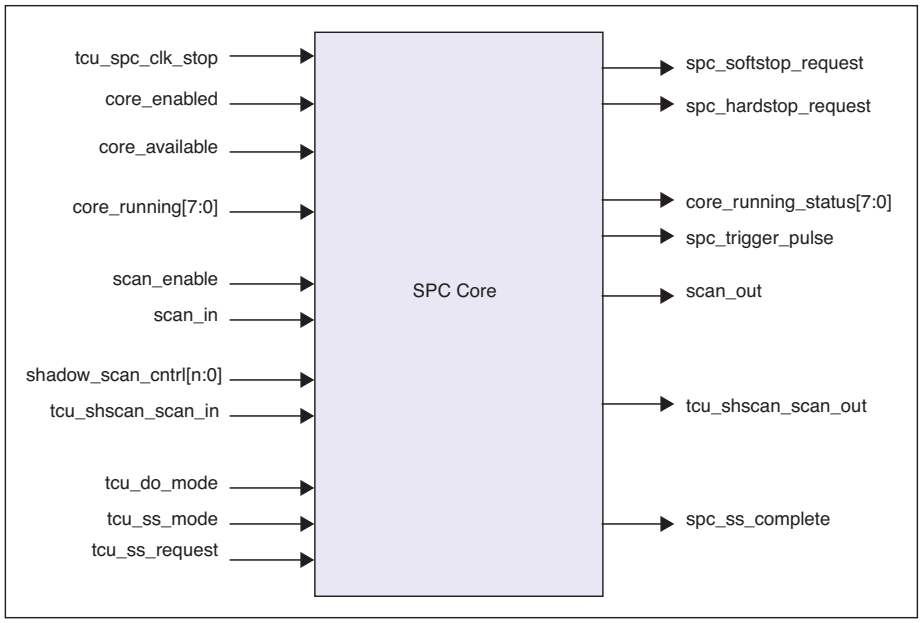


SerDes boundary scan Signals are indicated with _sbs_ in the signal name and connect to the clusters which interface to the SerDes macros since the SerDes boundary scan cells are located in the MCU clusters. These Signals are generated as described in the SerDes DFT document for OpenSPARC T2.

4.9 TCU Debug Interface to SPC Cores

The TCU interfaces with the SPC cores to support debug as shown in Figure 4-11 on page 52.

FIGURE 4-11 TCU to SPC Core Debug Interface



4.9.1 Clock Interface

The TCU provides a clock stop signal to the flop headers in the core, and drives this signal active when the core is unavailable.

4.9.1.1 Tcu_spc_clk_stop

This signal is deasserted to allow the OpenSPARC T2 core's clocks to run. This is the main signal the TCU uses to control the OpenSPARC T2 core's clocks. This signal can be set to '1' at any time to stop the OpenSPARC T2 core's clocks. Sourced from TCU as tcu_spc7_clk_stop ... tcu_spc0_clk_stop.

4.9.1.2 Core_available & Core_enabled

Core_available is set via efuse at manufacturing time and determines whether the physical core can be used in normal operation. It serves as a clock gate and if '0' will result in the clk_stop being asserted to the core (this happens in the TCU). Core_enabled is driven from the ASI_CMP_CORE_ENABLED register.

4.9.1.3 Core_running & Core_running_status

The core_running bus is an input from the NCU by which the TCU requests the core to perform a soft-stop. When the core sees the core_running Signals transition from a '1' to a '0', it will stop issuing instructions, and wait for all pending core operations to complete. Once all core-initiated memory operations have been globally performed, the core will raise the Soft_stop_req signal to allow the TCU to stop the clocks to the core and also raise the core_running_status bus to indicate to the CMP logic that the core is parked. When the soft_stop_req is a '1', the OpenSPARC T2 core will not issue instructions or initiate any activity, until the TCU drives core_running to a '1'.

4.9.1.4 Scan_enable

Besides configuring the scan chains for scanning, this signal also gates off OpenSPARC T2 core's interface Signals so that other SOC units do not respond to spurious OpenSPARC T2 core interface activity during scanning. At least the crossbar PCX interface is protected in this way.

4.9.1.5 Hardstop_request & Softstop_request

These Signals are outputs to the TCU which indicates that the core has reached either a hard-stop or a soft-stop condition and wants to request service from TCU. When a hardstop_request is received, the TCU should disable the clocks to the core using the TCU_spc_clk_stop signal. When the softstop is received, the TCU should request a soft-stop of the entire core via the core_running bus and upon receiving the softstop_request acknowledgement from all 8 threads in the core, the TCU will then stop the clocks via TCU_spc_clk_stop. In both cases, the TCU should begin

decrementing the Cycle Counter when the stop_request is received; when the Cycle Counter reaches 0 the clock_stopping sequence is initiated by the TCU. Note that the type of stop (hard vs. soft) is determined either by the Soft_stop_req signal, or which configured event in the DECR has occurred.

Setting bit 3 of the TCU DCR causes the TCU to soft stop all enabled SPCs if any requests a soft stop; See "TCU Debug Control Register" on page 61.

These debug event requests will be honored when the Cycle Counter and Debug Event counters reach zero, and the Reset Counter is not enabled.

4.9.2 Debug Event Interface

This group of core outputs Signals that either an error or a debug trigger event has occurred. These debug event requests will be honored when the Cycle Counter and Debug Event counters reach zero, and the Reset Counter is not enabled.

4.9.2.1 Trigger_event

This is a signal from the core to TCU. If the OpenSPARC T2 core is configured to trigger on an event in the DECR, and the associated event occurs, this signal transitions from a '0' to a '1'. It then transitions back to '0', unless another enabled DECR trigger event occurred that cycle. The TCU will pass this signal to a package pin as the OR of the (64) bits from all cores. The trigger_event signal will be synchronized to the I/O clock frequency.

4.9.3 Scan Interface

Not all Signals relevant to the scan interface are detailed here (e.g., not all the scan clocks and controls are listed).

4.9.3.1 Scan_in & Scan_out

There are three scan chains in each core. All flops on this scan string are reset both at POR and during warm reset unless protected via use of the "warm_reset_flop_header".

There are three external scan-out Signals per core; each corresponds to a scan-in signal. During JTAG access via scan an entire physical core may be scanned; in this mode the TCU will concatenate the three scan chains in the core, in addition to any JTAG private scan chains such as for shadow scan or memory bist.

4.9.3.2 Shadow_scan_in

This is the scan-in for the shadow-scan string.

4.9.3.3 Shadow_scan_cntrl[n:0]

This bus controls shadow scan operation and identifies which thread's state will be sampled to the shadow scan string.

When the TCU wants to do a shadow scan on a particular core, it first sends the command to the OpenSPARC T2 core on this bus. At some time later, OpenSPARC T2 core will capture the state requested by the TCU on the internal shadow scan flops. At that point the TCU can scan out the state by accessing the shadow scan scan string. For details of operation See "Shadow Scan" on page 78

The Signals included in this bus are:

- tcu_shscanid[2:0] : selects one of 8 threads
- tcu_shscan_pce_ov : provides a capture signal to the shadow scan reg.
- tcu_shscan_clk_stop : stops the clock to the shadow scan register to allow it to be scanned via JTAG
- tcu_shscan_aclk & tcu_shscan_bclk : shift clocks to perform the scan operation
- tcu_shscan_scan_en : a separate scan_enable for the shadow scan register

4.9.3.4 Shadow_scan_out

This is the scan-out of the core's shadow-scan string.

4.9.4 Single Step Mode

Individual threads can be placed in single step mode via JTAG. To place threads in single step mode the following sequence is used. The user must keep one physical core (SPC) in functional mode.

1. Specify which threads to be in single step mode via TAP_DOSS_ENABLE by setting the corresponding bits in the 64-bit disable overlap/single step enable register.
2. Park all threads by deasserting core_running[7:0] to the target SPCs via the TAP_CREG_ or TAP_NCU_ instructions and accessing the corresponding 8-bit fields in the 64-bit core run register. For any SPC to be operated in single-step mode, all of its threads should first be parked (turned off in core run register).

3. Wait until all threads from the targeted SPCs indicate they are parked via core_running_status[7:0]. This is done by reading the 64-bit core run status register via TAP_CORE_RUN_STATUS. Each bit corresponds to a thread.

4. When all targeted cores are parked, set the DOSS_MODE register to '11' using TAP_DOSS_MODE. Bit [0] indicates single step mode and bit [1] enables the mode. At this stage, the tcu_ss_mode signal is asserted to the targeted physical cores.

5. Assert core_running to the threads that will be single-stepped, via TAP_CREG or TAP_NCU; these threads should correspond to those set in DOSS_ENABLE to maintain compability with future enhancements.

6. Pulse the tcu_ss_request signal by executing a TAP_SS_REQUEST (the pulse is generated by going through the update-DR tap state); each running thread in a physical core enabled with tcu_ss_mode will fetch/execute a single instruction.

7. When a SPC's threads have all finished the single-step operation, then that SPC will pulse spc_ss_complete. The TAP_DOSS_STATUS is used to check the spc_ss_complete bit and returns 8 bits, one for each SPC. The status is held until the next TAP_SS_REQUEST. When all SPC's indicate they have completed, another single-step can be requested via TAP_SS_REQUEST.

8. Steps 6 and 7 can be repeated to execute a string of 'n' instructions.

9. To exit single step mode, park all threads in the SPCs being single-stepped using TAP_CREG_ or TAP_NCU_. After TAP_CORE_RUN_STATUS indicates all threads are parked, disable the mode using TAP_DOSS_MODE to set the mode to '00'. The DOSS_ENABLE register should also be cleared. Then unpark the desired threads by asserting the respective bits in the core run register using TAP_CREG_ or TAP_NCU_.

Note – Single stepping for multiple threads can be executed independently by control of the targeted threads' respective bits in core_running[7:0] in the above actions.

4.9.5 Disable Overlap Mode

Placing a SPC in disable overlap mode is similar to that for single step mode:

1. Specify which threads to be in disable overlap mode via TAP_DOSS_ENABLE by setting the corresponding bits in the 64-bit disable overlap/single step enable register.

2. Park all threads by deasserting core_running[7:0] to the target SPCs via the TAP_CREG_ or TAP_NCU_ instructions and accessing the corresponding 8-bit fields in the 64-bit core run register. For any SPC to be operated in disable-overlap mode, all of its threads should first be parked (turned off in core run register).
3. Wait until all threads from the targeted SPCs indicate they are parked via core_running_status[7:0]. This is done by reading the 64-bit core run status register via TAP_CORE_RUN_STATUS. Each bit corresponds to a thread.
4. Set the number of cycles to run during disable overlap mode using TAP_CYCLE_COUNT to set the cycle counter.
5. When all targeted cores are parked, set the DOSS_MODE register to '10' using TAP_DOSS_MODE. Bit [0] indicates single step mode and bit [1] enables the mode. At this stage, the TCU will automatically:
6. Assert tcu_do_mode to the target SPCs
7. Unpark the targeted threads
8. Start counting down the cycle counter, waiting until it reaches zero
9. Park the targeted SPCs
10. Set the DOSS_STATUS register
11. Status can be checked with TAP_DOSS_STATUS; bits will be set corresponding to the SPCs which have completed running in disable overlap and are parked.
12. To exit disable overlap mode, park all threads in the target SPCs using TAP_CREG_ or TAP_NCU_. After TAP_CORE_RUN_STATUS indicates all threads are parked, disable the mode using TAP_DOSS_MODE to set the mode to '00'. The DOSS_ENABLE register should also be cleared. Then unpark the desired threads by asserting the respective bits in the core run register using TAP_CREG_ or TAP_NCU_.

Note – The latency of parking and unparking the threads via UCB should be considered when setting the cycle counter.

4.9.6 Cycle Step Mode

Cycle step refers to stopping the clocks to a SPC and then stepping a number of clock cycles to that SPC. Individual SPC cores can be placed in cycle step mode via JTAG. In this mode, a SPC core is stopped via a hard clock stop, then a predefined number of clock pulses is allowed through. After this, the SPC flop contents can be

examined. In this mode the clock domains must be controlled since the clocks are stopped to the target SPCs; this is done by performing a hard stop across only the target SPCs, not the entire chip as in the default hard clock stop.

1. Use TAP_CORE_SEL to set the corresponding bits of the SPCs targeted for cycle stepping.
2. Use TAP_CLOCK_SSTOP to stop the clocks to the SPCs - note this will perform a hard stop on the target SPCs since the TAP_CORE_SEL is active. No SPCs will be parked.
3. Program the Cycle Counter using TAP_CYCLE_COUNT; this can be done before steps 1 and 2 also.
4. Verify that the clocks are stopped via TAP_CLOCK_STATUS, the value should be '10' indicating the clock stop operation is finished.
5. Issue a cycle step command via TAP_CS_MODE and loading a '1'. This begins the Cycle Counter operation and allows the number of clocks specified in the Cycle Counter to be sent to the target cores. When the Cycle Counter reaches zero, the clocks will again be stopped to the target SPCs.
6. Check status using TAP_CS_STATUS. This returns a 1-bit value that will be set when the Cycle Counter has finished. It does not indicate if the clocks have yet been stopped, the TAP_CLOCK_STATUS must be used for this.
7. When the status indicates the cycle step has completed, further actions may be taken such as dumping the core contents.
8. To turn the clocks to the SPCs back on, use TAP_CLOCK_START to turn clocks on to the target cores. Note it is impractical to expect the cores to resume operation as a hard stop was in effect.

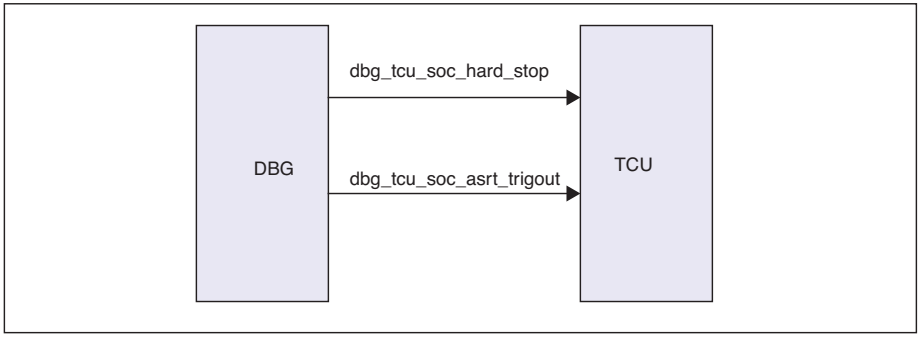
4.9.7 JTAG Priority for Debug

In general, any JTAG instructions related to core debug will take priority over other debug functionality in the TCU. This means that TCU responses to debug events could be blocked if another JTAG instruction is active. Results for JTAG debug operations concurrent with SPC debug service requests are unpredictable.

4.10 TCU Debug Interface to SOC Logic

The TCU interfaces with the SOC logic via the DBG unit as shown in Figure 4-12 on page 59. These debug event requests will be honored when the Cycle Counter and

FIGURE 4-12 TCU to SPC Core Debug Interface



4.10.1 Clock Interface

Hardstop_request

The DBG unit drives a signal dbg_tcu_soc_hard_stop to request a hard stop of the clocks to the chip. This signal is pulsed in io clock domain and when received, TCU will count down the cycle_counter and then begin stopping the chip's clocks. The manner in which the clocks are stopped - serial or parallel - is determined by the contents of the clock_domain register. There is no soft-clock-stop for the SOC logic. The TRIGIN pin is treated as an SOC hard stop request.

4.10.2 Debug Event Interface

Trigger_event

To send a watchpoint to the external trig_out pin, the DBG unit pulses dbg_tcu_soc_asrt_trigout high ('1') for one io clock cycle. TCU will pass this out to the I/O pins.

4.11 TCU Debug Registers

The TCU handles debug events requests from the SPC cores directly, or from the SOC via the DBG unit, as described in "Clock Stop" on page 35, "TCU Debug Interface to SPC Cores" on page 52 and "TCU Debug Interface to SOC Logic" on page 58. The response to these requests is to stop the clocks (hard or soft) or pass the watchpoint signal to the I/O pins. A set of registers is provided in the TCU to assist in control of these responses to debug event requests.

4.11.1 Cycle Counter

This is a 64-bit counter that can delay the response to a debug event. For example, if the TCU receives a hard-stop request the Cycle Counter will begin counting down with each cmp clock cycle and when it reaches zero then the hard-stop will be performed. All debug event requests from the SPC cores or a hard-stop request from SOC logic will be delayed by the Cycle Counter. The Clock Domain register is ignored in this mode: an SOC hard stop will start with clock domain [8]; a SPC hard stop will start with the requesting SPC, and a SPC soft stop will start with SPC0 or all SPCs in parallel if TCU_DCR[3] = 1 (also see Section 4.6.3 on page 36).

These actions are only valid when TCU_DCR[2] = 0. For behavior when TCU_DCR[2] = 1, see "TCU Debug Control Register" on page 61. The Cycle Counter is loaded with JTAG instruction TAP_CYCLE_COUNT; default is zero.

4.11.2 Debug Event Counter

This ia a 32-bit counter that must be zero before the Cycle Counter is enabled. If it is non-zero, then each debug event request received at the TCU will decrement it and when zero is reached, the Cycle Counter will begin decrementing with the next debug event request. No differentiation is made regarding debug event requests, so it is up to the user to insure only one type of debug event is enabled when using the Debug Event Counter. Debug event requests consist of these requests from SPC: spc_softstop_request, spc_hardstop_request, spc_trigger_pulse, or these requests from SOC: dbg_tcu_soc_hard_stop, dbg_tcu_soc_arst_trigout, and the trigin package pin. The events are counted per cmp clock cycle.

The Debug Event Counter is only recognized when TCU_DCR[2] = 0. When TCU_DCR[2] = 1, the Debug Event Counter is disabled. The Debug Event Counter is accessed with JTAG instruction TAP_DE_COUNT; default is zero.

4.11.3 TCU Debug Control Register

The TCU has a 4-bit register to control responses to debug events, the TCU DCR (Debug Control Register). When bit 2 of TCU DCR is '0' the Cycle Counter and Debug Event Counters perform as described above.

When bit 2 of the TCU DCR is set to '1' the lower 32 bits of the Cycle Counter are treated as a Reset Counter. In this mode, the Reset Counter begins decrementing with each cmp clock cycle after the Power-on Reset (POR) sequence ends (when tcu_rst_flush_stop_ack goes high at the end of flush reset sequencing in WMR2, see See "WMR2" on page 90). Once zero is reached either a watchpoint, a hard clock stop or a clock stretch can be performed, or the upper 32-bits of the Cycle Counter can then be used. In this mode (bit 2 of TCU DCR = 1) the Debug Event counter will be ignored.

The behavior of the Debug Event and Cycle Counters is determined by the values in the TCU DCR as specified in the following table. The TCU DCR is loaded with JTAG instruction TAP_TCU_DCR; default is zero ('0000').

TABLE 4-10 TCU Debug Control Register Field Definitions

Soft Stop [3]	Enable [2]	[1:0]	Description
0/1	0	xx	Debug Event and Cycle Counter recognize SPC debug events
x	1	00	Watchpoint pulsed
x	1	01	Hard Stop and Watchpoint Pulsed
x	1	10	Clock Stretch and Watchpoint Pulsed
x	1	11	Clock Stretch and Watchpoint, followed by Hard Stop and second Watchpoint

The following actions are valid when bit 2 of the TCU DCR is set to '1':

4.11.3.1 Watchpoint

If the TCU DCR is set to '100', then a single pulse of an external chip pin (TRIGOUT) will occur when the Reset Counter reaches zero. The pulse will be synchronized to the io clock domain. The upper 32-bits of the Cycle Counter are ignored.

4.11.3.2 Hard Stop

A hard clock stop will be performed if the TCU DCR is set to '101', as specified in "Hard Clock Stop" on page 36, and a watchpoint pulse generated, when the Reset Counter reaches zero. The upper 32-bits of the Cycle Counter are ignored.

4.11.3.3 Clock Stretch

If the TCU DCR is set to '110', then a clock-stretch signal will be pulsed out of the TCU when the Reset Counter reaches zero, and a watchpoint pulse will also be generated. The upper 32-bits of the Cycle Counter are ignored.

4.11.3.4 Clock Stretch then Hard Stop

If the TCU DCR is set to '111', when the Reset Counter reaches zero a clock stretch will be triggered and a watchpoint pulse will also be generated, and then the upper 32-bits of the Cycle Counter will be allowed to count down to trigger a clock hard stop and a second watchpoint will also be generated.

Note – The Soft Stop bit 3 when set will cause TCU to Soft Stop across all enabled SPCs when any SPC requests a soft stop. It should only be active when Enable bit 2 is '0'; if Soft Stop bit 3 is set when Enable bit 2 is set, the Clock Domain and Core Select registers will interact with each other - see Section 4.6.3 on page 36 for details.

For JTAG access to the clock stop logic, the TCU DCR should be in a reset condition so that bits 3 and 2 are both inactive (0). If either of these bits is set, the interaction between JTAG and the TCU DCR can become unpredictable.

4.11.4 Erratum #34 TRIGOUT (Watchpoint) Events

A watchpoint is also referred to as a Trigout event and pulses the TRIGOUT package pin. Watchpoints can be generated by the SPC cores, the SOC logic, SW, or as defined when TCU DCR bit [2] is set (see "TCU Debug Control Register Field Definitions" on page 61).

TCU will forward a trigout request from SOC or SPC when the cycle counter and debug events counters are zero, but it will only forward that one request. For clock stop/scan dump usage where a trigout is used, TCU generates the trigout based upon the debug event.

For clock stop/scan dump, one trigout may be sufficient. For soc/spc generated trigout requests, it may be desirable to have the ability for TCU to send out every one it receives. However, TCU was designed to only forward/create out one trigout. Two questions came up in regard to this:

1. After the first trigout request is recognized by TCU and forwarded, how do we reset TCU to forward the next trigout request?
2. How do we get multiple trigout events forwarded by TCU to the output pin?

For (1), this was not intended usage. A work-around is to wait until the first debug event has initiated a TRIGOUT request, and then to re-program the debug_event counter (the count does not matter, should be non-zero) and upon the next trigout request the TCU will forward it, and then block subsequent trigout requests. Re-programming the debug_event_counter can be done via SW, no JTAG is needed.

Alternatively, after the initial debug event, programming a TAP_CLOCK_START and then putting jtag in test-logic-reset will allow TCU to recognize the next TRIGOUT request.

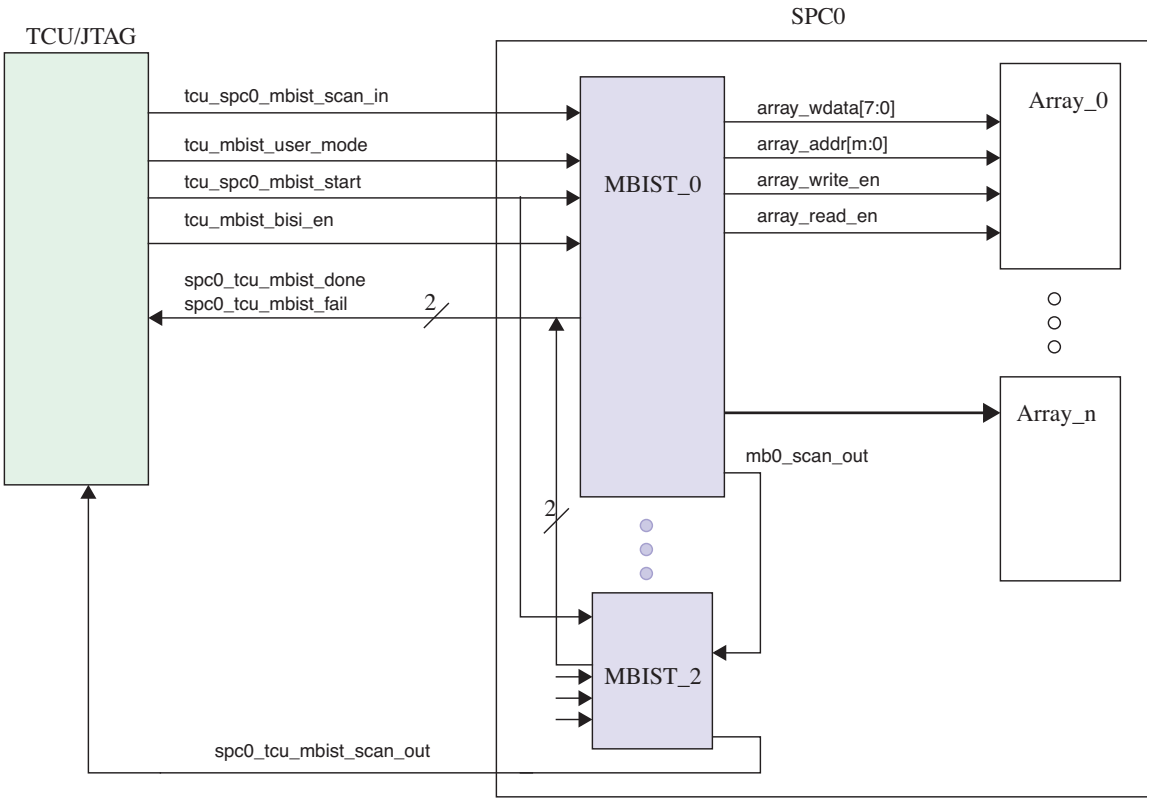
For (2), again this was not the intended usage. A work-around is to wait until the initial debug event occurs and the first TRIGOUT is forwarded by TCU. Then, program the TAP_CLOCK_START and keep it active - do not go to test-logic-reset, instead go to run-test-idle state. Then, all trigout requests received by TCU will be forwarded to the package TRIGOUT pin as long as the TAP_CLOCK_START is active.

4.12 Memory BIST Control

4.12.1 Overview

The memory BIST or MBIST engines for OpenSPARC T2 are based on the engine used in OpenSPARC T1. The general organization between TCU/JTAG, a single MBIST engine in SPC0, and its associated arrays is shown in the following figure. In OpenSPARC T2 there are 80 MBIST engines: 3 per core (24 total) and 56 distributed throughout the SOC logic. Each MBIST engine will therefore test several arrays. Note that even though there are 80 engines, only 48 are visible from the TCU MBIST controller as explained in Section 4.12.7 on page 68.

FIGURE 4-13 Overview of MBIST Control via TCU/JTAG



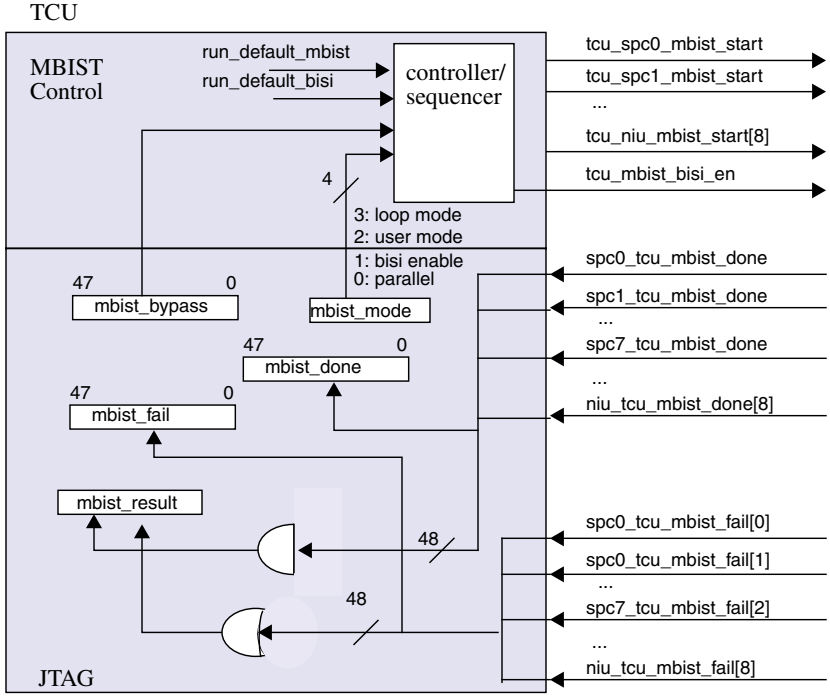
The MBIST operation may be controlled by the TCU during reset sequencing via the JTAG interface or as invoked via SW. The MBIST engines can either be operated in a serial mode, a parallel mode or a diagnostic mode for memory bit-fail mapping. Both the serial and parallel modes run MBIST in a pass/fail mode, where the only information available is whether MBIST passed all of its arrays, or failed at least one of them.

4.12.2 Memory BIST Operation

TCU controls operation of MBIST through an MBIST controller which can be programmed either via JTAG or SW. Operation of the MBIST controller is described in the following sections.

The signal mbist_parallel tells a sequencer in the TCU to begin either serial activation of the start Signals if low, or parallel activation if high, based on the contents of the mbist_bypass register. The signal run_default_mbist is activated if enabled during the reset sequence to perform MBIST over all the arrays. The signal run_default_bisi activates BISI on all arrays, in which case the sequencer would set tcu_mbist_bisi_en which goes to all arrays, along with sequencing start signals either serially or in parallel. To run BISI on a given array, the TAP_MBIST_DIAG instruction must be used to program a given MBIST engine's config bits. Selection of BISI or MBIST is done by setting the corresponding bit in the MBIST config register of the MBIST engines, or by setting the BISI bit in the mbist_mode register.

FIGURE 4-14 Conceptual Look at TCU/JTAG MBIST Control



When the serial or parallel MBIST is determined to be finished (via polling/examination of the done/fail register, or a timeout), all that is known is that either all arrays passed, or at least one of them failed. To get information on which MBIST engine finished, the TAP_MBIST_GETDONE instruction must be used. This allows capture of 48 done bits which may then be observed at TDO; the TAP_MBIST_GETFAIL similarly captures 48 fail bits. If detailed information as to which array failed within a given MBIST engine is needed, then the TAP_MBIST_DIAG instruction must be used to retrieve the contents of the specific MBIST engine that indicated a fail.

4.12.3 Serial Mode

JTAG will typically be used to run MBIST in the serial mode. When activated in serial mode, the MBIST engines will be started sequentially in the following order.

Serial MBIST ordering:

- SPC0: MBIST 0, 1, 2
- SPC1: MBIST 0, 1, 2
- . . .
- SPC7: MBIST 0, 1, 2
- SII: MBIST 0
- . . .

To enable the serial MBIST mode via JTAG the instruction TAP_MBIST_BYPASS must be used to specify which of the 48 MBIST engines to bypass, if any, via the mbist_bypass register. Next, the TAP_MBIST_MODE is used to clear the parallel mode bit in the mbist_mode register. The TAP_MBIST_START instruction is then programmed into JTAG; when JTAG enters the run-test-idle state, the MBIST operation will be started; it is not necessary to remain in the rti state. It is up to the user to wait a predetermined number of cycles for the MBIST operation for all arrays to finish. Status can be checked using the TAP_MBIST_RESULT instruction and capturing the mbist_result register (2 bits) in the CaptureDR state and examing them; this can be done repeatedly for polling (via captureDR, without staying in run-test-idle). This allows early truncation of the test (via the TAP_MBIST_ABORT instruction) if the fail bit becomes active before the MBIST operation is done. The done bit must be set to validate a fail bit of 0 indicating a passing condition. A done bit set to 1 and a fail bit set to 0 indicates all arrays for the selected mbist engines passed MBIST.

The default operation is to run BISI instead of MBIST. To run BISI the instruction TAP_MBIST_DIAG may be used to access the MBIST engine specified via the mbist_bypass register, and setting the bisi/bist bit to 1 in the config register if provided in the BIST engine. Another option for running BISI is to set the BISI bit in the TAP_MBIST_MODE register. Optionally, the memory BIST can be run in the serial mode without programming JTAG; this is done via the run_default_mbist signal.

4.12.4 Parallel Mode

JTAG must be used to run MBIST in the parallel mode. When activated in parallel mode, the MBIST engines will be started in parallel, while the arrays controlled by each individual MBIST engine will test their arrays sequentially. Operation of MBIST parallel mode via JTAG is similar to the serial mode, except that the parallel mode

bit of the mbist_mode register must instead be set, using the TAP_MBIST_MODE instruction. There is no non-JTAG default method of running MBIST in parallel mode.

4.12.5 Diagnostic Mode

In one method to perform bit-fail mapping, the TAP_MBIST_DIAG instruction is used to access the MBIST engine as the target JTAG data register. In this diagnostic mode only one MBIST engine should be selected, by setting the appropriate bits in the mbist_bypass register via the TAP_MBIST_BYPASS instruction; it is up to the user to bypass all but one MBIST engine. Only one array controlled by the selected MBIST engine may be active; this is specified by scanning in (loading) the target MBIST engine registers. After both the MBIST engine and array are specified, the TAP_MBIST_START is programmed, and entering run-test-idle will start the MBIST operation on the selected array. After an appropriate wait time the test should finish. Polling via TAP_MBIST_RESULT can be used to inspect the done/fail JTAG data register, or the TAP_MBIST_GETDONE and TAP_MBIST_GETFAIL can be used to determine the MBIST test results.

To get the detailed information on the target array, the TAP_MBIST_DIAG instruction must be used. This allows the contents of the targeted MBIST engine to be scanned as the mbist_diag register via TDO. This architecture is depicted in the following figure for spc0 where all three MBIST engines in the core are on the same chain. Similarly, an individual scan chain is provided for each spc and soc cluster as listed in Table 4-11.

4.12.6 Abort Mode

To abort any MBIST activity the TAP_MBIST_ABORT instruction should be used. This will cause all MBIST start Signals to be deasserted, and any internal JTAG states to be reset. A separate instruction is useful since the JTAG MBIST instructions have memory. Use of TAP_MBIST_ABORT does not clear any of the JTAG data registers used for or during MBIST, only the control states and Signals, and does not clear the MBIST engine flops; this allows the TAP_MBIST_DIAG to be used to get data on the failing arrays. Note: Entering test-logic-reset state does not stop MBIST.

4.12.7 MBIST Engine Ordering

The MBIST engines are ordered as follows for the 48-bit JTAG done, fail and bypass registers:. There are 80 MBIST engines in OpenSPARC T2 but from a JTAG

TABLE 4-11 MBIST Engine Ordering

Cluster	# of Engines	JTAG Reg.	Cluster	# of Engines	JTAG Reg.
SPC0	3	bits[0]	L2B_2	1	bit[20]
SPC1	3	bit[1]	L2B_3	1	bit[21]
SPC2	3	bit[2]	L2B_4	1	bit[22]
SPC3	3	bit[3]	L2B_5	1	bit[23]
SPC4	3	bit[4]	L2B_6	1	bit[24]
SPC5	3	bit[5]	L2B_7	1	bit[25]
SPC6	3	bit[6]	L2T_0	3	bit[26]
SPC7	3	bit[7]	L2T_1	3	bit[27]
SII	2	bits[9:8]	L2T_2	3	bit[28]
SIO	2	bits[11:10]	L2T_3	3	bit[29]
NCU	2	bits[13:12]	L2T_4	3	bit[30]
MCU0	1	bit[14]	L2T_5	3	bit[31]
MCU1	1	bit[15]	L2T_6	3	bit[32]
MCU2	1	bit[16]	L2T_7	3	bit[33]
MCU3	1	bit[17]	DMU	2	bits[35:34]
L2B_0	1	bit[18]			
L2B_1	1	bit[19]			

perspective only 48 are visible. Three engines in each SPC are visible as one engine by JTAG, and similarly for the L2 Tags. Since there are 8 SPCs and 8 L2Ts, this is a reduction of 32 (24 to 8 for SPCs, 24 to 8 for L2Ts) for a total reduction of 80-32=48 engines visible by JTAG.

4.12.8 Notes

The TCU sources the MBIST engine register scan controls over the scan controls for the holding cluster/core. When an MBIST engine is accessed via a JTAG MBIST instruction the other scan chains in the cluster will also scan, but the data will be lost in those chains..

To use TAP_MBIST_DIAG, the user must bypass all engines (using TAP_MBIST_BYPASS) except the one desired, else the result is indeterminate.

All three core MBIST engines and associated array information (if any) selected via the JTAG instructions are placed in that core's first scan chain for ATPG test mode.

JTAG instructions to support MBIST:

TAP_MBIST_BYPASS 48-bit mbist_bypass register

TAP_MBIST_MODE4-bit mbist_mode register

TAP_MBIST_STARTno user data register

TAP_MBIST_RESULT2-bit mbist_result register

TAP_MGIST_DIAGx-bit mbist_diag Reg: Engine + array flops

TAP_MBIST_GETDONE48-bit mbist_done register

TAP_MBIST_GETFAIL48-bit mbist_fail register

TAP_MBIST_ABORTno user data register

TAP_MBIST_CLKSTPENno user dr; enables clock stop via cycle counter

4.12.9 JTAG MBIST Data Registers

JTAG accessible registers for MBIST are:

TABLE 4-12 JTAG MBIST Registers

Register	JTAG Instr.	Fields
Result[1:0]	TAP_MBIST_RESULT	bit[1] : 1 when all 80 mbist engines are done bit[0] : 1 if any of 80 mbist engines reports a fail
Bypass[47:0]	TAP_MBIST_BYPASS	One bit per mbist engine; to bypass an engine during MBIST testing set its bit to 1
Done[47:0]	TAP_MBIST_GETDONE	One bit per mbist engine; a 1 indicates the corresponding engine is done; same order as mbist_bypass register

TABLE 4-12 JTAG MBIST Registers

Register	JTAG Instr.	Fields
Fail[47:0]	TAP_MBIST_GETFAIL	One bit per mbist engine; a 1 indicates the corresponding engine failed MBIST for one of its arrays
Diag[k:0]	TAP_MBIST_DIAG	Includes targeted MBIST engines in a cluster; variable length
Mode[3:0]	TAP_MBIST_MODE	bit[3] : user loop mode bit[2] : user mode bit[1] : bisi mode if 1, bist mode if 0 bit[0] : parallel mode if 1, serial mode if 0
none	TAP_MBIST_CLKSTPEN	Enables mbist controller to begin Cycle Counter; reset with TLR or TAP_CLOCK_START

4.12.10 MBIST Clock Stop and Scan Dump

The Cycle Counter may be used in conjunction with MBIST to stop clocks and perform a scan dump. The instruction TAP_MBIST_CLKSTPEN must be programmed to enable the Cycle Counter for MBIST. If enabled, the Cycle Counter will begin decrementing when the MBIST controller begins operation. When the Cycle Counter reaches zero, a hard clock stop will be issued to the clock sequencer.

All relevant registers - such as clock domain and clock stop delay - will be recognized in this mode to allow control of the clock stop sequence. The clock stop status may be checked with TAP_CLOCK_STATUS, and when stopped the scan chains can be dumped via TAP_SERSCAN.

Using this feature and repeatedly running MBIST with successively greater cycle count values allows another method of bit-fail mapping arrays. This is sometimes referred to as MBIST Plus. Since the start of MBIST and when the Cycle Counter begins decrementing is coordinated and synchronized to the same cmp clock cycle, the entire process should be repeatable and cycle accurate.

4.12.11 MBIST DMO - Direct Memory Observe

The basic operation as implemented in TCU is described here. There are three JTAG instructions, TAP_DMO_ACCESS, TAP_DMO_CLEAR and TAP_DMO_CONFIG, as described in Table 4-3 on page 5. The TAP_DMO_ACCESS puts the chip in DMO mode, so that read data from L2 Tags and some SPC or NIU arrays will be observable at package pins during MBIST operation, in addition to done and fail

information. TAP_DMO_CLEAR clears this mode. To access and program the dmo control logic inside TCU the TAP_DMO_CONFIG should be used to set the 48-bits as desired. TAP_MBIST_ABORT does not clear DMO mode.

TABLE 4-13 JTAG DMO Configuration Register accessed via TAP_DMO_CONFIG

Register	Field	Description
DMO_Config[47:0]	[47:16]	32-bit shift register; bit 47 is used to sample dmo data
	[15]	1 selects CMP clock domain (SPC/L2T) 0 selects IO clock domain
	[14:13]	00 selects dmo path to cores 4, 5, 1 or 0 01 selects dmo path to cores 6, 7, 3 or 2 10 selects dmo path to L2 tags 4, 5, 1 or 0 11 selects dmo path to L2 tags 6, 7, 3 or 2
	[12:11]	Not defined
	[10:8]	Not defined
	[7]	selects SPC data cache upper/lower word
	[6]	1 selects SPC instr. cache and L2 Tag output 0 selects SPC data cache and L2 Data output
	[14:13] & [5:0]	[14:13] [5:3][2:0] ==> Cluster Selected 00 xx0 xxx ==> CORE4 10 xx0 xxx ==> L2T4 00 x01 xxx ==> CORE5 10 x01 xxx ==> L2T5 00 011 xxx ==> CORE1 10 011 xxx ==> L2T1 00 111 xxx ==> CORE0 10 111 xxx ==> L2T0 01 xxx xx0 ==> CORE6 11 xxx xx0 ==> L2T6 01 xxx x01 ==> CORE7 11 xxx x01 ==> L2T7 01 xxx 011 ==> CORE3 11 xxx 011 ==> L2T3 01 xxx 111 ==> CORE2 11 xxx 111 ==> L2T2

Note – As in most single-access jtag instructions in OpenSPARC T2, reading the dmo config register using TAP_DMO_CONFIG is done via the capture-DR state and is always followed by writing the register when update-DR is passed through. Thus, the dmo config register should not be accessed while dmo is actively running as the shift register contents will be disturbed during update-DR.

In DMO mode, TCU will pass the data, done and fail information for the mbist array under test to the package pins, along with a synchronization pulse.

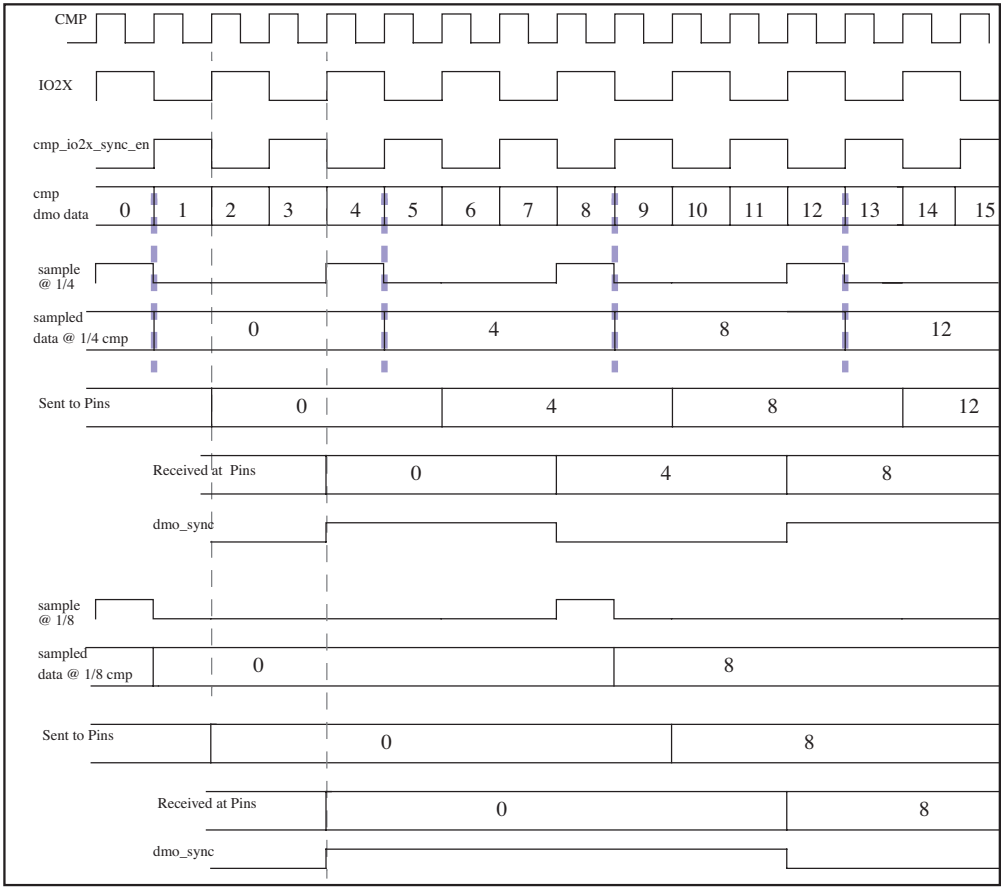
4.12.11.1 MBIST Done and Fail Observability at Pins

The capability exists to have any MBIST engine pass its done and fail Signals to the package pins and is enabled with TAP_DMO_ACCESS. Due to synchronization it is not guaranteed that all fail pulses will be seen at the package pin. The done and fail information is intended to be observed only in user mode with one array selected. In non-user mode the fail pin is indeterminate if failures occur (if no failures occur, the fail pin will pulse only at MBIST sequencer initiation).

When the MBIST sequencer is initiated, TCU will pulse the mbist done and fail pins for one io2x clock cycle. While MBIST is running TCU will pass the fail information received from the MBIST engine which is running to the pins in user mode only (non-user mode is indeterminate). When the MBIST sequencer finishes, TCU will assert done to the pins and assert fail to the pins if there was any failure recorded from MBIST. This means that there will be at least one pulse generated by TCU on the fail pin even if no fail occurred.

Upon initiation of the MBIST sequencer the pulses on the done and fail pins will be coincident. The reason for the initial pulses on the done and fail pins is so the user can determine that the MBIST controller in TCU has started and the pins are capable of toggling. The done and fail will only be pulsed upon MBIST start when the MBIST controller begins operation. This means that the done and fail will not pulse each time a successive MBIST engine is started if more than one engine is left in non-bypass mode.

FIGURE 4-15 Sample: MBIST DMO data coming from CMP clock domain



The figure above is representative only, and shows 1/4 sampling (32-bit shift register set to 32'b00010001000100010001000100010001) and sampling at 1/8 (32-bit shift register set to 32'b00000001000000010000000100000001). Note that the msb (bit 47 of the dmo config register, bit 31 of the shift register inside the dmo config register) is used to sample the dmo data.

4.12.12 Scanning of MBIST Engines via JTAG

When scanning MBIST engines using the TAP_MBIST_DIAG instruction or TAP_SERSCAN it may be that flops outside the targeted scan chain may be disturbed. The effects for SPC and SOC engines are different.

When scanning a SPC MBIST engine the controls from TCU will not be seen by any logic outside the target SPC. However, if TAP_MBIST_DIAG is used to obtain a short chain between TDI and TDO the flops outside this chain in the target SPC will be affected by aclk, bclk and scan enable. It may be advantageous to use TAP_SERSCAN with only the targeted SPC selected so that all values in the SPC scan chain can be controlled.

When scanning an SOC MBIST engine there is no individual scan control. So all SOC logic except for RST, TCU and CCU will be affected by the scanning of any SOC MBIST engine.

Also please see "Protecting TCU During Serial Scan: Test Protect Mode" on page 34 for proper use of TAP_TP_ACCESS during MBIST serial scanning.

4.12.13 Effect of Unavailable or Disabled Cores and Banks

The MBIST sequencer in TCU observes the Available and Disabling Signals as described in Section 4.6.6 on page 43. When either BIST or BISI is run the MBIST sequencer in TCU will automatically bypass any MBIST engines in an unavailable or disabled SPC or L2 array including MCU and the associated L2 Tag. If only one MBIST engine is selected, with all others bypassed, and that MBIST engine is in an unavailable or disabled SPC or L2/MCU, the MBIST sequencer will bypass it and effectively do no MBIST testing. This is an illegal state and MBIST sequencer operation is not determinstic.

4.12.14 BIST During Reset

During the POR sequence as described in the chapter on reset (See "Reset Sequencing" on page 87), TCU will run a BISI sequence after POR1 and optionally either run BISI or BIST between WMR1 and WMR2. The BISI run after POR1 is in parallel mode by default and has a timeout counter of 32 bits. The signal tcu_rst_bisx_done will be asserted when all non-bypassed engines return their done Signals to TCU or the timeout counter expires. BISI will use the bypass register to select which engines to run and will not expect done Signals from bypassed engines. The BIST mode register is not applicable except for changing from parallel to serial mode since the BISI run is "hard-coded" to run after POR1.

An optional BIST or BISI run is available if programmed by Software and will be recognized by TCU after the next WMR1 and will be serviced between WMR1 and WMR2. This optional run will recognize the bypass and mode registers.

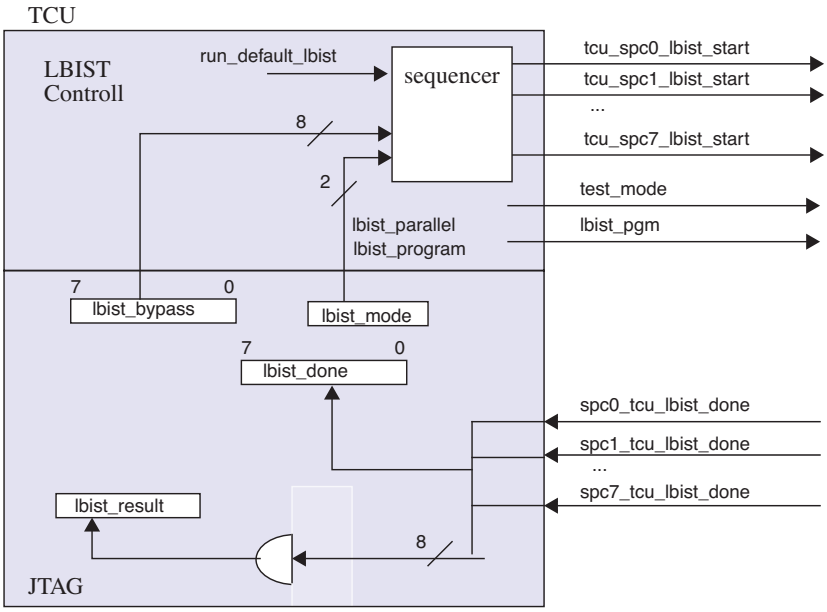
Both the POR1 BISI and the optional Software-requested BIST/BISI will obey the SPC and L2 available and disable criteria as specified in section 13.13 Effect of Unavailable or Disabled Cores and Banks. Note, the BISI enable is written by logic during the power-on reset sequence,and once written it will remain high until it is programmed otherwise.

4.13 Logic BIST Control

The Logic BIST test function is only applied to the SPC cores in OpenSPARC T2, and one engine is instantiated per core. The control of the Logic BIST engines comes from the TCU either via SW or JTAG.

The control logic allows the Logic BIST engines to be run in parallel or in series, and gathers the done Signals for JTAG to query. There is no pass/fail indication that comes from the Logic BIST engines, so the engine must be scanned to determine the result.

FIGURE 4-16 Conceptual look at TCU/JTAG Logic BIST control



To start a single Logic BIST engine, the TCU will drive the lbist_start signal high and hold it until the lbist_done signal is received. TCU will also source a test_mode signal to all Logic BIST engines to control them during manufacturing scan. In parallel mode, the non-bypassed engines will be sent lbist_start Signals in the same cycle. In serial mode, the first non-bypassed engine (counting from 0) will be sent an lbist_start indication, then when its lbist_done is received the next non-bypassed engine will be sent an lbist_start indication. All lbist_start Signals will be held until the sequencer has received done indications from all engines.

4.13.1 JTAG Logic BIST Instructions

The instruction TAP_LBIST_START can be used to begin Logic BIST sequencing, and TAP_LBIST_ABORT can be used to stop sequencing. The JTAG accessible registers for Logic BIST are:

TABLE 4-14 JTAG Logic BIST Registers

Register	JTAG Instr.	Fields
Bypass[7:0]	TAP_LBIST_BYPASS	One bit per Logic BIST engine; to bypass an engine during testing set its bit to 1
Mode[1:0]	TAP_LBIST_MODE	bit[1] : program access mode selected bit[0] : parallel mode if 1, serial mode if 0
Lbist[k:0]	TAP_LBIST_ACCESS	Includes targeted Logic BIST engines across cores
Done[7:0]	TAP_LBIST_GETDONE	One bit per mbist engine; a 1 indicates the corresponding engine is done; same order as Logic BIST bypass register

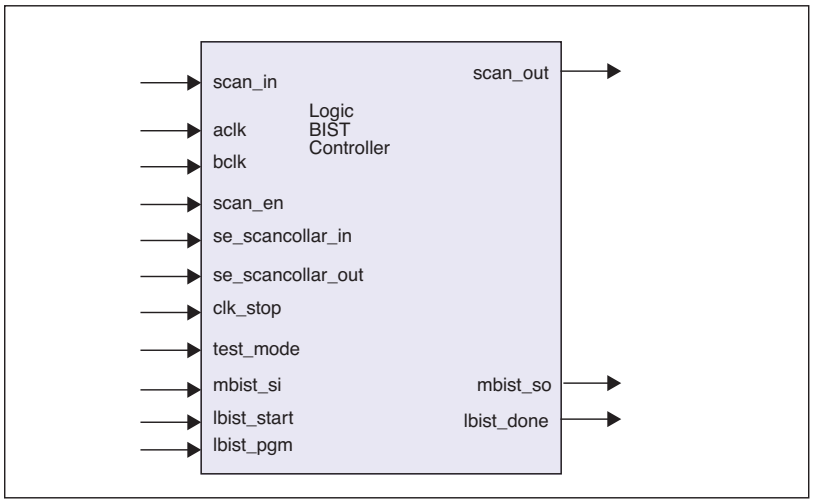
4.13.2 Accessing Pass/Fail Signature

To determine if the Logic BIST engine passed or failed, the signature must be scanned out via JTAG using TAP_LBIST_ACCESS. The signature must be compared against a known-good value. Alternatively, Software may access the signature. This mechanism will be implemented in the SPC core.

Also please see "Protecting TCU During Serial Scan: Test Protect Mode" on page 34 for proper use of TAP_TP_ACCESS during LBIST serial scanning with TAP_LBIST_ACCESS.

4.13.3 Logic BIST Interface

FIGURE 4-17 Logic BIST Controller Interface with TCU

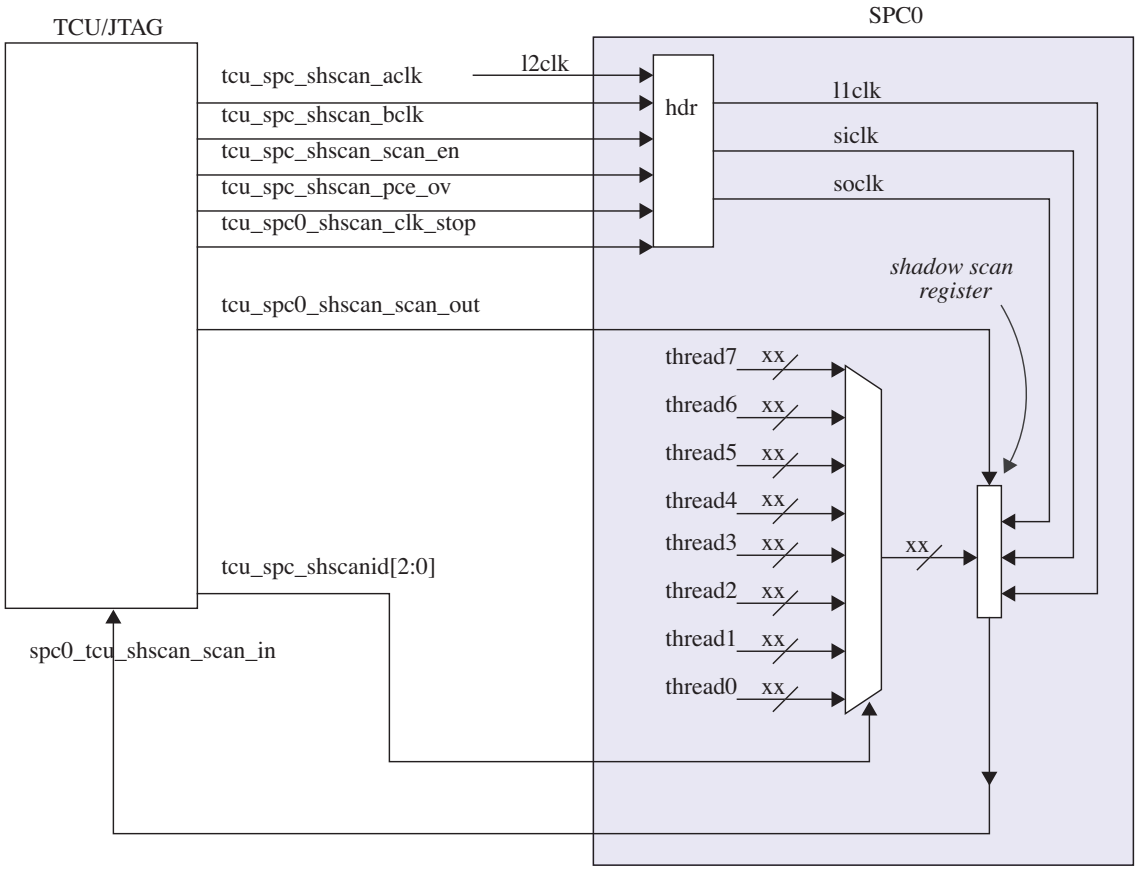


4.14 Shadow Scan

4.14.1 Core Shadow Scan

Shadow scan for the cores is controlled via JTAG. The architecture is shown FIGURE 4-18; the header is a conceptual view of both the cluster and flop headers combined. Each core shadow scan will be contained in a separate scan chain, with its own clock headers and controls coming from the TCU. The contents to be captured in the shadow scan are in the *OpenSPARC T2 Programmer's Reference Manual*. If a core is disabled then its shadow scan contents

FIGURE 4-18 Core Shadow Scan Architecture



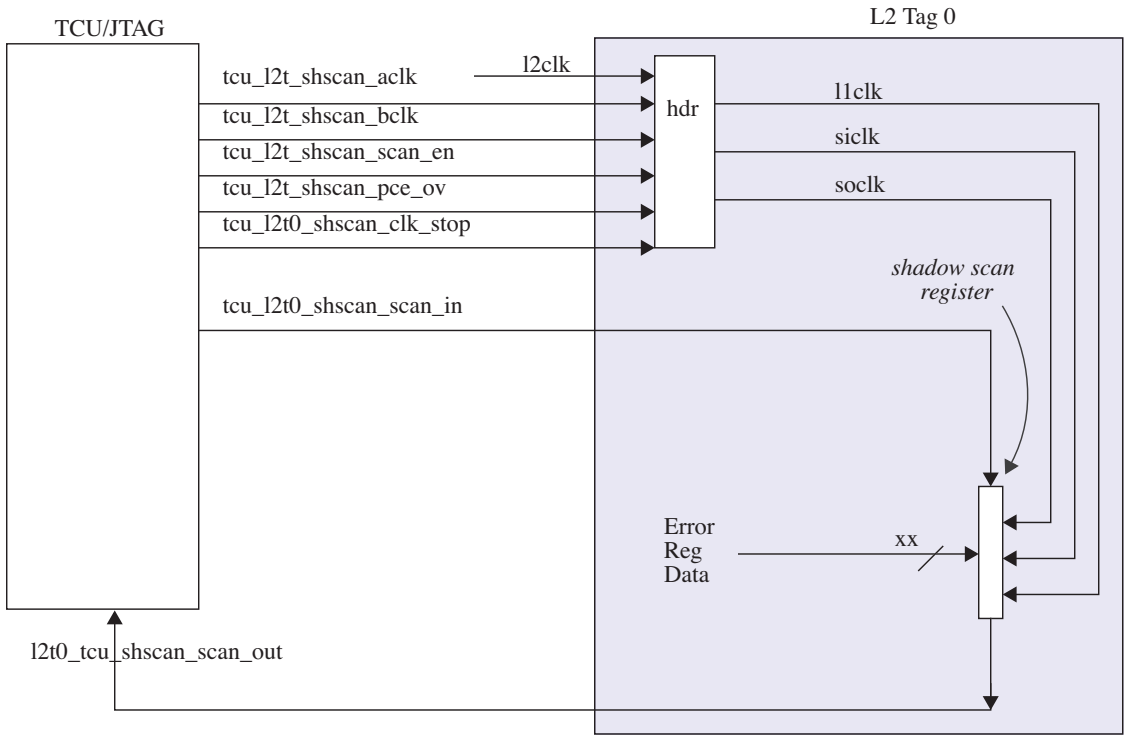
will be excluded and the number of TCK clocks should be reduced to reflect the unavailable core(s).

4.14.2 SOC Shadow Scan

Shadow scan for the SOC consists solely of L2Tag error registers, and is controlled via JTAG. The architecture is shown in FIGURE 4-19; the header is a conceptual view of both the cluster and flop headers combined. Each L2 Tag shadow scan will be contained in a separate scan chain, with its own clock headers and controls coming from the TCU. The contents to be captured in the shadow scan are in the *OpenSPARC T2 Programmer's Reference Manual*.

siisyn_data[61]	**"000001"**	**"000001"**	**"000001"**	**"000001"**	**"000001"**	**"000001"**
Etag[2:0]	"001"	"111"	"101"	"000"	"100"	"110"

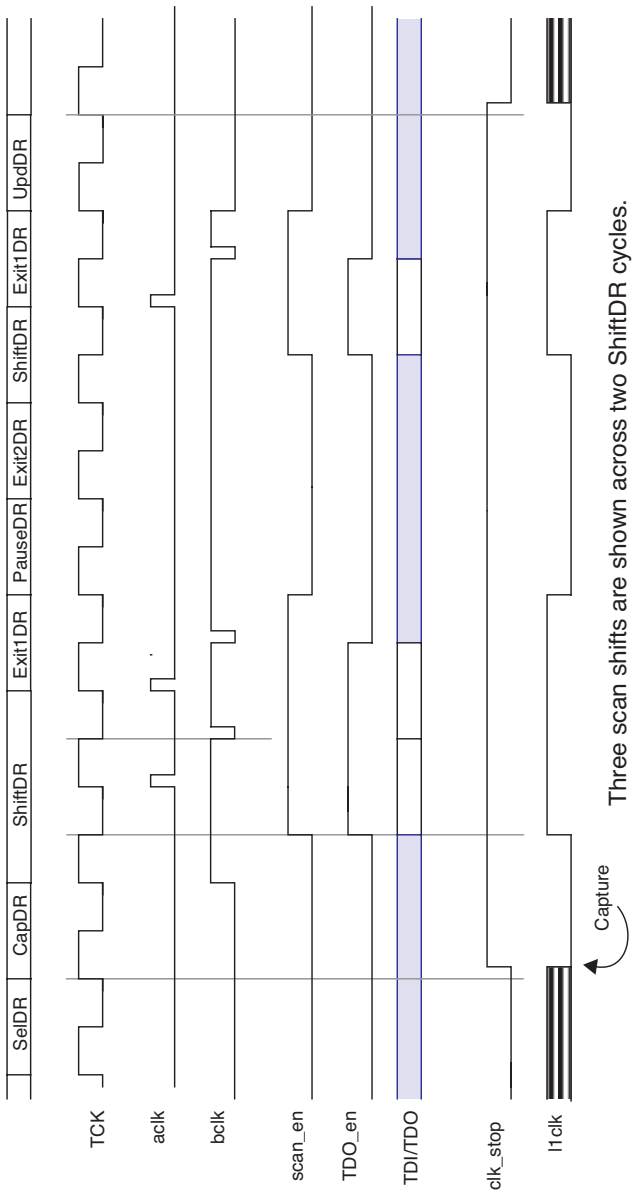
FIGURE 4-19 L2 Tag Shadow Scan Architecture



4.14.3 Shadow Scan Operation

During a shadow-scan operation, the PLL is running and JTAG is used to capture the desired values into the shadow scan register. The contents are then scanned-out via TDO. Both the core and L2 tag shadow scan registers can only be read; any value scanned into them will be overwritten. Because TCK is specified to be at a much slower frequency than any cpu clock, any cpu clock cycles required for synchronization from TCK to cpu clock domains will not cause overlapping.

FIGURE 4-20 JTAG Shadow Scan Sample Waveform



NOTES:

1. All 8 core shadow scans are scanned serially as one chain, with core 0 closest to TDI and core 7 closest to TDO.
2. Assignment of shadow scan contents is in the *OpenSPARC T2 Programmer's Reference Manual*.
3. Any core marked unavailable in the CMP core_available register will not be included when scanned via TDI to TDO.
4. The shadow scan chain for a given core is placed in that core's second scan chain during ATPG test mode; they are accessible via JTAG shadow scan instructions and during jtag serial scan.
5. All 8 L2 Tag shadow scan contents are captured at the same time, and are available at TDO with L2T0 first and L2T7 last (closest to TDO).
6. JTAG instructions to support Core Shadow Scan:
   - TAP_SPCTHR0_SHSCAN Thread 0 contents for all available cores
   - TAP_SPCTHR1_SHSCAN Thread 1 contents for all available cores
   - TAP_SPCTHR2_SHSCAN Thread 2 contents for all available cores
   - TAP_SPCTHR3_SHSCAN Thread 3 contents for all available cores
   - TAP_SPCTHR4_SHSCAN Thread 4 contents for all available cores
   - TAP_SPCTHR5_SHSCAN Thread 5 contents for all available cores
   - TAP_SPCTHR6_SHSCAN Thread 6 contents for all available cores
   - TAP_SPCTHR7_SHSCAN Thread 7 contents for all available cores
7. JTAG instructions to support L2 Tag Shadow Scan
   - TAP_L2T_SHSCAN

4.15 Array Guidelines to Support Scan Test

To facilitate scan test the arrays should be configured so that they can be inhibited during scan load and unload, and surrounded with scan collars. There are several different scan modes used on OpenSPARC T2 and this section outlines the use and requirements for the scan mode control Signals. The different scan modes consist of Manufacturing or pin-based scan, also known as ATPG scan, MacroTest, Logic BIST (LBIST), Transition Test, JTAG scan and Flush scan.

The TCU sources four Signals for scan control specifically related to arrays:

- *tcu_se_scancollar_in* - connect to “se” port of flop headers for memory “input” flops
- *tcu_se_scancollar_out* - connect to “se” port of flop headers for memory “output” flops
- *tcu_array_wr_inhibit*
- tcu_array_bypass

4.15.1 Flop (Clock) Headers

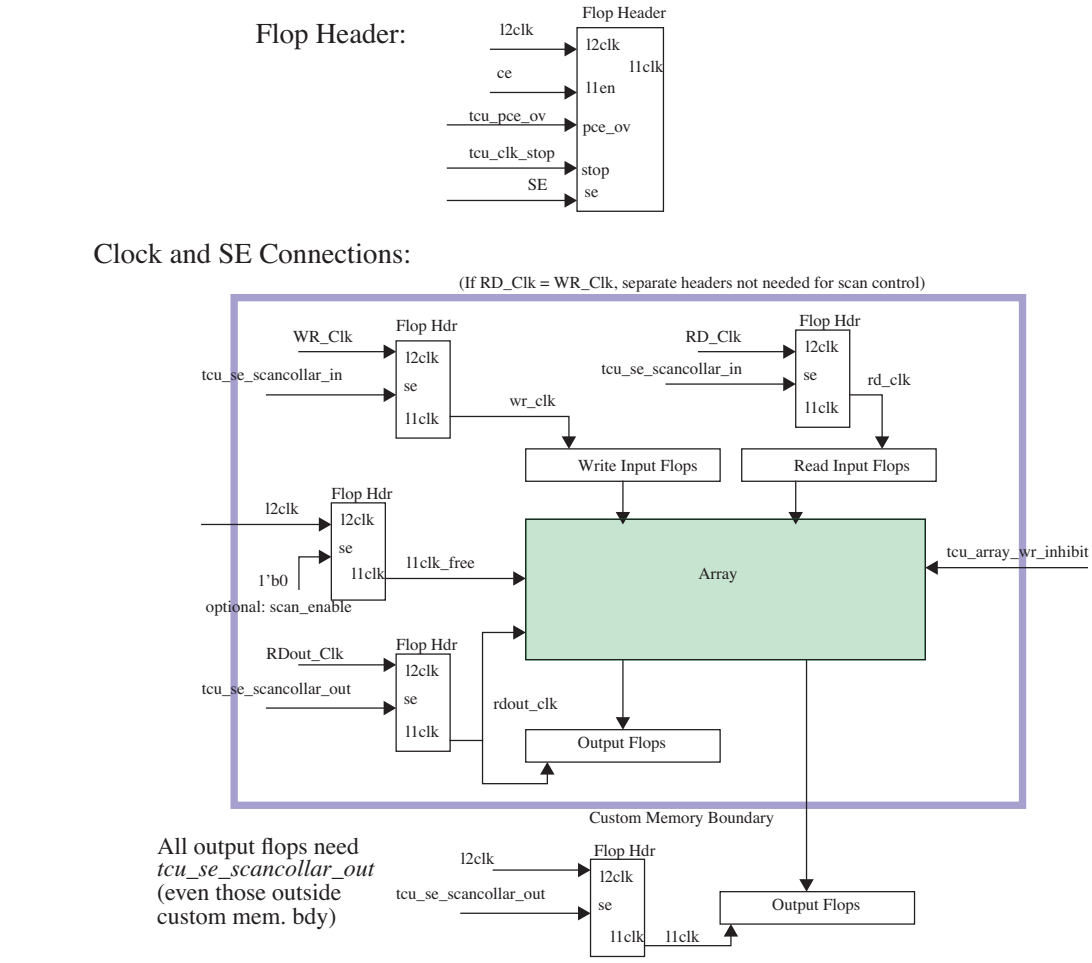
To control the clocks to arrays during the various scan modes, clock headers are needed with specific se (scan enable) Signals. The se Signals from TCU to arrays are *tcu_se_scancollar_in* and *tcu_se_scancollar_out*.

For any input flops including write address/data, read address/data, write/read enable, and inputs related to lookup for CAMS, the flop headers for these flops should use *tcu_se_scancollar_in*. For any output flops, such as read data, a second flop header is required and should use *tcu_se_scancollar_out*. The various flop headers in an array should share the *tcu_pce_ov* and *tcu_clk_stop* Signals and all flops can share *tcu_aclk* and *tcu_bclk*. If the “se” port of a flop header is tied low, then during scan operations the l1clk will track the l2clk - this is sometimes referred to as making the l1clk free-running. If the power-savings function of a flop header is not needed then the “l1en” (ce) can be tied high. Refer to Bob Molyneaux’s “Test Clocking” document on the OpenSPARC T2 MAS web page for further details on the flop header.

4.15.2 Write Inhibit and Bypass

To inhibit writing to the arrays the TCU generates a signal *tcu_array_wr_inhibit*. When active, this signal should protect the array from updates and can also be used to turn off read logic and CAM compare logic if desired. If it is determined by test coverage analysis that there is logic not tested, such as shadow logic, a bypass mux may be needed. In general it is expected that most arrays will not need a bypass mux; this decision will be made for each array individually. The bypass mux will be controlled via *tcu_array_bypass*. Plan-of-record for OpenSPARC T2 arrays is to place the bypass mux outside the custom memory boundary (in RTL logic), except for CAMS that have already placed bypass muxes in the custom memory.

FIGURE 4-21 Array Flop Header Guidelines



Note – If WR_Clk and RD_Clk are different clock domains then separate stop Signals should be used as provided by the TCU.

4.15.3 Scan Modes

The values for the various test control Signals and the clocks are given in the following table for the different scan modes (*tcu_clk_stop*=0 and *tcu_pce_ov*=1). The value of *l1clk_free* tracks *l2clk*.

TABLE 4-15 Array Control Signals During Scan Modes

Scan Mode	Phase	l2clk	se_scancollar_in	se_scancollar_out	array_wr_inhibit	l1clk
ATPG	Scan Shift	Tester drives to 1	1	1	1	1
	Capture	Tester toggles	0	0	1	l2clk
MacroTest	Scan Shift	Tester drives to 1	1	1	1	1
	Capture	Tester toggles	1	0	0	in: 1 out: l2clk
Logic BIST	Scan Shift	PLL Locked	1	1	1	1
	Capture	PLL Locked	0	1	1	in: l2clk out: 1
Trans. Test	Scan Shift	PLL Locked	1	1	1	1
	Capture	PLL Locked	0	0	1	l2clk
JTAG	Scan Shift	PLL Locked	1	1	1	1
Flush	Scan Shift	PLL Locked	1	1	1	1

4.15.4 Scan Cell Ordering Guidelines

Scan Cell Ordering: There is no specific requirement for ordering of scan cells in the scan chain, although it is desirable for all flops of the same function to be grouped in the chain to facilitate macrotest pattern development. Lockup latches are not needed since the scan clocks are always non-overlapping.

It is required that the ordering of scan cells in the circuit match that in the RTL.

4.15.5 Reset

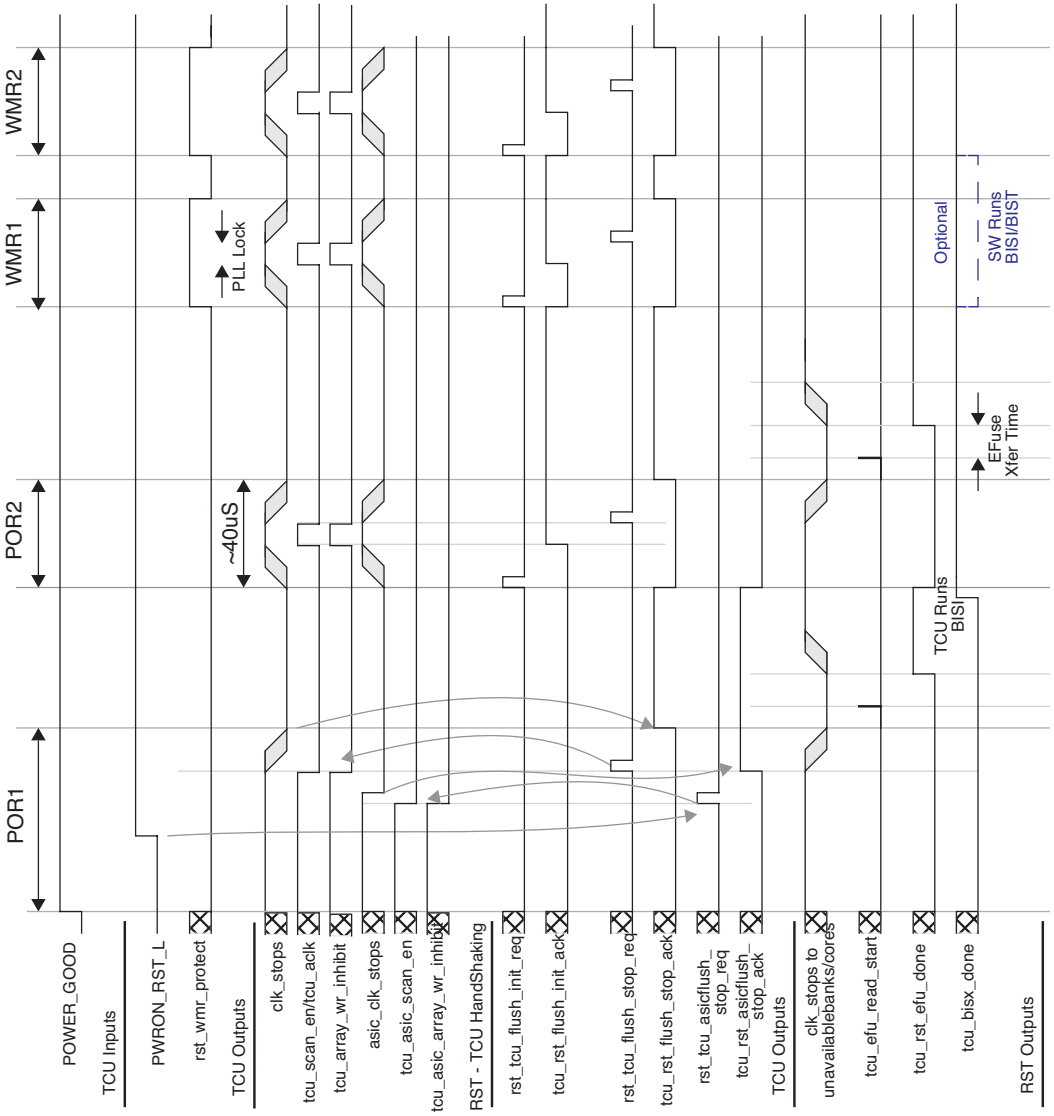
During portions of the power-on-reset sequence, such as before the PLLs lock or during flush scan, tcu_array_wr_inhibit will be driven active to protect the arrays.

ASIC arrays do not participate in flush scan, so aclk/bclk would be inhibited during this time to those arrays.

4.16 Reset Sequencing

The TCU participates in Power-On Reset by interfacing with the RST unit and providing flush reset. The waveforms with respect to TCU during power-on reset are shown FIGURE 4-22.

FIGURE 4-22 Power-On Reset Sequence



There are four phases to the Power-On Reset sequence: Two Power-On Resets - POR1 and POR2 - and two Warm Resets - WMR1 and WMR2.

POR1

At time zero, PWR_ON_RST_L is driven to '0'; the TCU sees this and then asserts all clk_stop Signals and resets itself and all chip logic (except for JTAG, RST, CCU, and DMU which get their own reset Signals) via flush reset.

The TCU is released from reset and becomes active when the PWR_ON_RST_L has deasserted (goes to '1') and the RST block drives rst_tcu_asicflush_stop_req high. Once the TCU is active it will wait for rst_tcu_flush_stop_req before ending the flush state and starting the clocks by turning off clk_stops to the 24 clock domains. Note that this does not apply to the ASICs as described below (See "ASIC Reset" on page 91). When it has turned on all clock domains, it will drive tcu_rst_flush_stop_ack to '1' and then assert tcu_efu_rvclr after a delay of 32 cmp clock cycles, hold tcu_rvclr for 8 cmp clock cycles, deassert and wait 8 cmp clock cycles, then pulse tcu_efu_read_start for 8 cmp clock cycles. The hold times and delays allow the efuse Signals to be synchronized into the io clock domain.

After the EFU is done, the TCU will start the default BISI sequence. This will complete before POR2 is entered. From a TCU perspective, POR1 ends when POR2 begins.

Notes

Since the PLL is locking during POR1 reset, the flush will be held until the PLL is stable before exiting flush_POR1. This is controlled by the RST unit.

The clocks to unavailable SPC cores and L2 Banks will be stopped; for details See "Clock Stopping and Core/L2 Available and Disable Controls" on page 43.

POR2

The POR2 state is recognized by TCU when the rst_tcu_flush_init_req goes to '1' and the rst_wmr_protect is low. The TCU responds by stopping all clocks, and drives tcu_rst_flush_init_ack high when it begins to flush the scan chains after clocks have been stopped.

When the rst_tcu_flush_stop_req is received the TCU will cease flushing the scan chains, and turn all clock domains back on, followed by driving tcu_rst_flush_stop_ack high to tell RST the clocks are running. TCU also pulses tcu_efu_read_start to again signal the EFU to begin operation. The timing of the efuse handshaking is the same as in POR1.

The DMU is excluded from flush resets and clock stopping (the RST, CCU and TCU are excluded from POR2). The start of WMR1 indicates the end of POR2 from a TCU perspective.

WMR1

The WMR1 state is recognized by rst_wmr_protect being active during the receipt of rst_tcu_flush_init_req to begin this phase. The actions are similar to POR2 between TCU and RST as shown, except no efuse start is sent by TCU. When TCU drives the tcu_rst_flush_stop_ack high this indicates the end of WMR1 from a TCU perspective.

Before leaving WMR1 TCU will start any pending BIST/BISI sequence which may have been requested via SW; this will be allowed to complete before WMR2 is entered.

WMR2

The WMR2 state is recognized only after WMR1 has occurred by rst_wmr_protect being active during the receipt of rst_tcu_flush_init_req; TCU responds as in WMR1. At the end of WMR2, though, the Reset Counter will be allowed to decrement as specified in the section on debug; See "TCU Debug Registers" on page 60. The Reset Counter should be programmed before WMR2 ends.

When TCU drives the tcu_rst_flush_stop_ack high this indicates the end of WMR2 from a TCU perspective.

4.16.1 JTAG Access During POR

JTAG is operational after TRST_L goes inactive during POR1; however, to access registers that are outside of JTAG the clocks must be running and the targeted areas should be capable of receiving the JTAG actions. During the POR sequence there are specific times when such access via JTAG is possible. To insure the user performs JTAG accesses during a safe period in the POR sequence, three JTAG instructions have been provided that create a window in the POR sequence so that JTAG instructions can be safely performed. By executing TAP_JTPOR_ACCESS a signal inside TCU will be set that will cause TCU to pause after EFUSE 2 transfer completes by delaying activation of the tcu_rst_efu_done signal. The user should execute TAP_JTPOR_ACCESS after releasing TRST_L in the POR sequence. The status of TCU can then be checked with TAP_JTPOR_STATUS; when the status is '1' this indicates that the TCU is paused and the JTAG programming window is active. Clocks will be running and JTAG instructions can be executed during this window. To continue with POR the user should execute TAP_JTPOR_CLEAR, which will cause TCU to continue with the POR sequence.

Note that when setting the JT POR access via TAP_JTPOR_ACCESS, it is possible to hold the chip input pin PWRON_RST_L low to allow enough time for the jtag programming to be completed. The sequence then would be to begin the POR sequence, release TRST_L but hold PWRON_RST_L low, complete the TAP_JTPOR_ACCESS programming, and then release PWRON_RST_L to allow the reset sequence to continue.

To shorten the bisi sequence in POR1 the JTAG TAP_MBIST_ABORT can be used. The execution of this instruction would need to be controlled by counting sys clock pulses and choosing the appropriate cycle to abort. If the BISI sequencer in TCU receives an MBIST-Abort request it will simulate a BISI timeout and the sequence will continue as if an actual timeout had occurred. This is useful during testing on ATE to shorten the POR sequence. It is up to the user to determine when to execute the TAP_MBIST_ABORT, and this does not require using TAP_JTPOR_ACCESS as the abort needs to occur before the JTPOR pause occurs.

One of the primary uses of JTAG access during POR is to bypass the efuse. One important point to keep in mind is that anything that has already happened in the POR sequence before the pause won't see the bypassed efuse values. In particular, BISI will have already completed during POR1 using data from the first EFUSE transfer. Thus, if a bank available row in the EFUSE array is bypassed during JTPOR to make an unavailable bank available, the bank that was marked as unavailable during POR1 will not have been initialized by BISI and will contain garbage data. It is up to software to tolerate this.

4.16.2 ASIC Reset

The ASIC blocks in OpenSPARC T2 DMU are treated differently from other clusters during the reset sequence and warm or debug resets.

During POR1 the DMU has its clocks stopped until the RST unit tells TCU to release them with rst_tcu_asicflush_stop_req; this signal comes earlier than rst_tcu_flush_stop_req. When the asicflush_stop_req is received, TCU releases itself from its own flush reset and turns off the clock stops to the ASICs and deasserts tcu_asic_scan_en. The tcu_asic_aclk is not asserted at all during POR1, preventing a flush state to the ASICs. During subsequent resets (WMR1, WMR2) the ASIC clock stops are allowed to activate in the normal clock stop sequence but the ASICs are not flushed. During debug resets (DBR) the signal rst_tcu_dbr_gen is active and TCU does not activate the clock stops to the ASICs to allow them to continue running. During JTAG clock stop operations, these blocks behave as other SOC blocks. During POR2 the ASIC clock stops will be held deasserted.

4.17 EFuse

The interface between the TCU and the E-Fuse Unit (EFU) is similar to that from OpenSPARC T1. This section only describes the TCU to EFU interface including the JTAG instructions used. For information about the EFU refer to its micro-architecture specification.

There are five modes of operation which the TCU recognizes for interfacing with the TCU. All except the POR mode require JTAG instructions and user intervention. The POR mode is handled directly by the TCU during the power-on reset sequence.

Note: Bypass data register is in EFU, and has reversed bit ordering from other JTAG registers (msb is closest to TDO). Bit ordering may also apply to other EFU register values as interpreted by the EFU. Please see the EFU MAS for details.

4.17.1 POR Mode

During the power-on reset sequence the EFU needs to send data to the chip. It does this when activated by a signal from TCU called tcu_efu_read_start (ioclk domain). This signal is pulsed and released to start the EFU, and may be activated multiple times during the POR sequence.

4.17.2 JTAG Read Access

This allows the user to read each row of the E-fuse Array (EFA) inside the EFU. The EFA is 64 rows by 32 columns. The JTAG instructions in the suggested order of application are:

- TAP_FUSE_READ_MODE Set the mode bits
- TAP_FUSE_ROW_ADDRSpecify the row address
- TAP_FUSE_READRead the specified row

4.17.3 Program Mode

This allows the usere to program the EFA one bit at a time in conjunction with proper application of the package pins required for EFA programming (fct_efa_prog_en). The JTAG instructions in the suggested order of application are:

- TAP_FUSE_ROW_ADDRSpecify the row address

- TAP_FUSE_COL_ADDRSpecify the column address
- Assert appropriate package pins per TI specifications

These steps may be repeated; the addresses will remain active until changed so once a row address is set it need not be changed until all columns have been traversed.

4.17.4 Bypass Mode

This allows the user to bypass the EFA, so that the EFU will treat user-supplied data as if it came from the EFA. This is useful for sending user data to SRAM redundancy registers to verify repairability. The JTAG instructions in the suggested order of application are:

- TAP_FUSE_BYPASS_DATA Send the data to EFU that will be used in place of EFA
- TAP_FUSE_BYPASSTell the EFU to use the bypass_data and send it out

4.17.5 Sample Mode

This allows the usere to sample a redundancy value from an array. The JTAG instructions in the suggested order of application are:

- TAP_FUSE_BYPASS_DATA Specify register to be sampled
- TAP_FUSE_DEST_SAMPLERequest EFU to get data and return it to JTAG

4.17.6 Redundancy Value Clear

To provide a means of clearing redundancy values, the TAP_ FUSE_RVCLR instruction is provided. This allows the user to clear all or specific redundancy values via the EFUSE unit. The same mechanism is used by TCU to tell EFUSE to clear all redundancy values before initiating an EFUSE start sequence during POR.

TABLE 4-16 EFUSE Redundancy Value Clear Register

Register	JTAG Instr.	Fields
efu_rvclr[6:0]	TAP_FUSE_RVCLR	efu_rvclr[6] = 1 enables a clear efu_rvclr[5:0] = block_id per efuse spec.; selects Redundancy Value to clear efu_rvclr[5:0] = 11_1111 will tell efuse to clear all RV's

4.18 TCU Local CSR Assignments

The Base Address for TCU is 0x85_0000_0000.

Devices can access the following registers in TCU via the UCB protocol with the offset addresses listed. Reference the corresponding section in this document for details on each register. **Note (1)**: In the case of the MBIST Mode and Bypass registers the default value is over-written by logic during the power on reset sequence: the BISI enable bit 1 of the MBIST mode register is written by logic during the power-on reset sequence,and once written it will remain high until it is programmed otherwise. The value of the MBIST Bypass register will depend on the core and bank available fuse values after POR1; if there is no partial mode, then the MBIST Bypass register will be all 1's in bits 47:0.

4.18.1 Memory BIST Registers

These registers are protected during warm resets unless modified via JTAG.

TABLE 4-17 MBIST Mode Register (0x00)

Bits	Name	Initial Value	R/W	Description
[63:4]	Reserved	X	RW	Reserved
[3]	Loop	0	RW	Loop mode if '1'
[2]	User	0	RW	Diagnostic (user) mode if '1'
[1]	BISI	0 (1 - note 1)	RW	BISI if '1', BIST if '0'
[0]	Parallel	0	RW	Parallel mode if '1'

TABLE 4-18 MBIST Bypass Register (0x08)

Bits	Name	Initial Value	R/W	Description
[63:48]	Reserved	X	RW	Reserved
[47:0]	Bypass	0 (48'hFFFFFFFFFFFF - note 1)	RW	MBIST Bypass

TABLE 4-19 MBIST Start Register (0x10)

Bits	Name	Initial Value	R/W	Description
[63:1]	Reserved	X	W	Reserved
[0]	Start	0	W	Starts MBIST Sequence when written to ‘1’

TABLE 4-20 MBIST Abort Register (0x18)

Bits	Name	Initial Value	R/W	Description
[63:1]	Reserved	X	W	Reserved
[0]	Abort	0	W	Aborts MBIST Sequence when written to ‘1’

TABLE 4-21 MBIST Result Register (0x20)

Bits	Name	Initial Value	R/W	Description
[63:2]	Reserved	X	R	Reserved
[1]	Result	0	R	MBIST Done (bit 1)
[0]	Result	0	R	MBIST Fail (bit 0)

TABLE 4-22 MBIST Done Register (0x28)

Bits	Name	Initial Value	R/W	Description
[63:48]	Reserved	X	R	Reserved
[47:0]	Done	0	R	MBIST Done

TABLE 4-23 MBIST Fail Register (0x30)

Bits	Name	Initial Value	R/W	Description
[63:48]	Reserved	X	R	Reserved
[47:0]	Fail	0	R	MBIST Fail

TABLE 4-24 MBIST Start WMR Register (0x38)

Bits	Name	Initial Value	R/W	Description
[63:1]	Reserved	X	W	Reserved
[0]	Start	0	W	Starts MBIST Sequence when written to '1', but delayed until after the next warm reset occurs

4.18.2 Logic BIST Registers

These registers are protected during warm resets unless modified via JTAG.

TABLE 4-25 LBIST Mode Register (0x40)

Bits	Name	Initial Value	R/W	Description
[63:2]	Reserved	X	RW	Reserved
[1]	Program	0	RW	Program mode if '1'
[0]	Parallel	0	RW	Parallel mode if '1'

TABLE 4-26 LBIST Bypass Register (0x48)

Bits	Name	Initial Value	R/W	Description
[63:8]	Reserved	X	R/W	Reserved
[7:0]	Bypass	0	R/W	LBIST Bypass

TABLE 4-27 LBIST Start Register (0x50)

Bits	Name	Initial Value	R/W	Description
[63:1]	Reserved	X	W	Reserved
[0]	Start	0	W	Starts LBIST Sequence when written to '1'

TABLE 4-28 LBIST Done Register (0x60)

Bits	Name	Initial Value	R/W	Description
[63:8]	Reserved	X	R	Reserved
[7:0]	Done	0	R	LBIST Done Status

4.18.3 Debug Control Registers

This register can operate during warm resets if enabled by TCU DCR.

TABLE 4-29 Cycle Counter Register (0x100)

Bits	Name	Initial Value	R/W	Description
[63:0]	Cycle Counter	0	RW	See "Cycle Counter" on page 60

CHAPTER 5

Clock Control Unit (CCU)

This chapter contains the following sections:

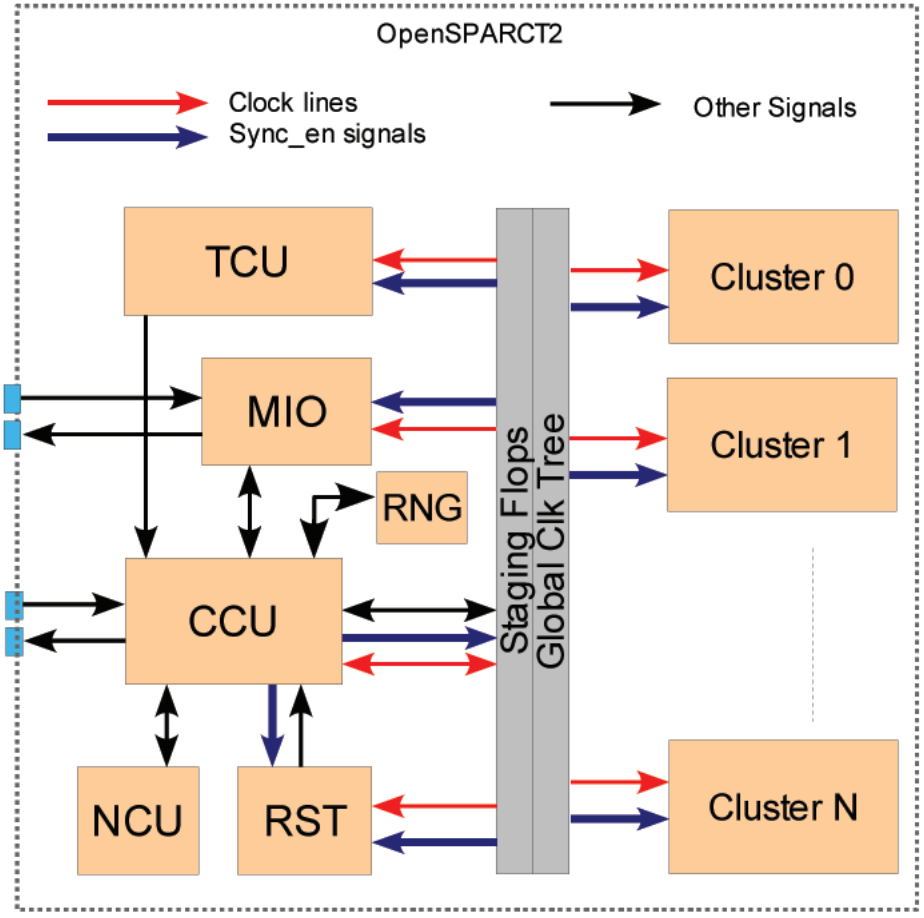
5.1 Overview

This is the microarchitecture specification for the CCU block. It encompasses the following functionality

- PLL to drive the core and memory clocks
- Interfacing with random number generator
- UCB interface for programming the pll's/rng and reading rng data
- Provide sync pulses for deterministic clock domain crossing
- Clock stretch and other test clocking mechanisms such as SerDes testing (via DTM) for OpenSPARC T2

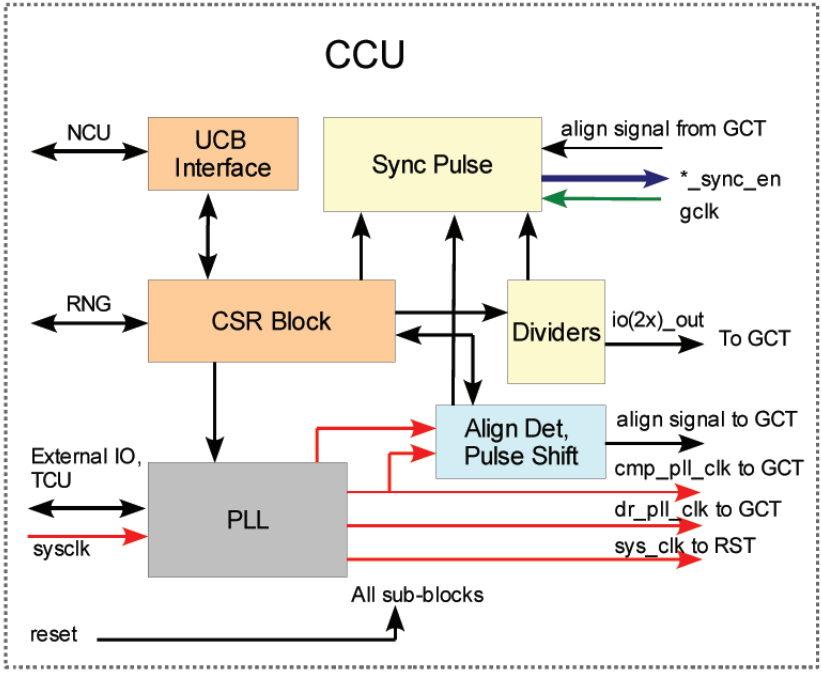
5.1.1 System Block Diagram

FIGURE 5-1 System Block Diagram



5.1.2 CCU Block Diagram and Description

FIGURE 5-2 CCU Block Diagram



Currently, the PLL is the only hard-macro in the CCU. The PLL generates clocks and also performs clock stretching. This is described in Section 5.3, “Clock and Reset Inside CCU” on page 5-15 and Section 5.4, “SYNC Pulses” on page 5-20. Section 5.4, “SYNC Pulses” on page 5-20 also includes information on the clock dividers, global clock tree and staging flops.

Interfacing with the RNG block involves an LFSR that can be accessed as a CSR register. Details are given in Section 5.5, “RNG Description” on page 5-27.

UCB interface for programming CSR's is reused from the RST cluster. Its protocol is described in Section 7.4, “ Interface Signals, Protocols, and Timing Diagrams” on page 7-10.

The actual registers resides in the CSR block. It also includes logic for interfacing to the other side of UCB, as described in the UCB interface document. Register addresses and fields are defined in Section 5.6, “CSR Block” on page 5-29.

Sync pulse generation is described in Section 5.4, “SYNC Pulses” on page 5-20, with the detailed analysis in the appendices.

5.2 CCU Port List

TABLE 5-1 CCU Port Listing

Name	Dir	Width	Domain	Description
CLK Input/Outputs				
gclk	I	1	N/A	Input to CCU cluster headers from global clk tree
dr_pll_clk	O	1	N/A	Connect to global clock tree input
cmp_pll_clk	O	1	N/A	Connect to global clock tree input
CCU-NCU Interface				
ccu_ncu_stall	O	1	io	UCB interface between CCU <-> NCU
ncu_ccu_vld	I	1	io	UCB interface between CCU <-> NCU
ncu_ccu_data	I	[3:0]	io	UCB interface between CCU <-> NCU
ncu_ccu_stall	I	1	io	UCB interface between CCU <-> NCU
ccu_ncu_vld	O	1	io	UCB interface between CCU <-> NCU
ccu_ncu_data	O	[3:0]	io	UCB interface between CCU <-> NCU
PLL-Bump Interface				
pll_sys_clk_p	I	1	N/A	Differential input reference to PLL
pll_sys_clk_n	I	1	N/A	Differential input reference to PLL
pll_vdd	I	1	static	PLL VDD – static tie high
mio_ccu_pll_char_in	I	1	async	Direct bump input to PLL – selects internal PLL signal during characterization active when mio_pll_testmode== 1. Also CSR programmable
CCU-RNG Interface				
rng_arst_l	O	1	async	Asynchronous reset of rng, also used to precharge voltage of large caps of the RC filters
rng_data	I	1	async	Input bit stream of random data (combined from up to 3 noise cells). Loaded into LFSR which is accessible via CSR address RNG_DAT
rng_bypass	O	1	async	Relates to generation of entropy in noise cells-- CSR programmable
rng_vcoctrl_sel	O	[1:0]	async	PMOS diode D/A setting bus -- CSR programmable
rng_ch_sel	O	[1:0]	async	Channel select for using entropy from 1,2 or 3 noise cells -- CSR programmable

TABLE 5-1 CCU Port Listing *(Continued)*

Name	Dir	Width	Domain	Description
rng_anlg_sel	O	[1:0]	async	Selects internal analog signal for characterization -- CSR programmable
SCAN/Test Related				
scan_in	I	1	aclk	Scan chain input – (currently hooked up to DMU scan_out output)
tcu_scan_en	I	1	async	Scan enable from TCU
tcu_aclk	I	1	N/A	aclk input to clkgen module. Connect to TCU
tcu_bclk	I	1	N/A	bclk input to clkgen module. Connect to TCU
scan_out	O	1	aclk	Scan chain output – (currently drives RST scan_in port)
tcu_atpg_mode	I	1	async	Puts the CCU in test mode for ATPG testing. Unless this signal is asserted, aclk, bclk and scan inputs into the CCU are all held low, and the scan chain is shorted.
ccu_dbg1_serdes_dtm	O	1	io	Sets DBG1 mux controls for DTM
ccu_mio_serdes_dtm	O	1	io	Sets MIO mux controls for DTM
Global Clock Tree Interface				
ccu_cmp_io_sync_en	O	1	cmp	Sync pulse for cmp -> io clk domain
ccu_io_cmp_sync_en	O	1	cmp	Sync pulse for io -> cmp clk domain
ccu_dr_sync_en	O	1	cmp	Sync pulse for cmp -> dr clk domain
ccu_io2x_sync_en	O	1	cmp	Sync pulse for cmp -> io2x clk domain
ccu_io2x_out	O	1	cmp	Divider phase signal output – rate of CMP clk. Connect to ccu_div_ph of clkgen module in other clusters as needed
ccu_io_out	O	1	cmp	Divider phase signal output – rate of CMP clk. Connect to ccu_div_ph of clkgen module in other clusters as needed
gl_ccu_io_out	I	1	cmp	Divider phase input; similar to ccu_io_out inputs for other clusters.
ccu_vco_aligned	O	1	vco	Align signal tightly coupled to PLL clock domain
gclk_aligned	I	1	cmp	Align signal tightly coupled to (cmp) gclk domain
ccu_serdes_dtm	O	1	async	Places chip in DTM mode where dr_clk == io_clk, and cmp, dr, io clock phases are deterministic. Currently unused in cluster headers.
CCU-MIO Interface				
mio_pll_testmode	I	1	async	Dedicated. Input from external IO through MIO – Used to place PLL in test mode (active high)

TABLE 5-1 CCU Port Listing *(Continued)*

Name	Dir	Width	Domain	Description
ccu_mio_pll_char_out	O	[1:0]	async	Dedicated. Digital characterization output of PLL. Connect to external IO through MIO. Valid when mio_pll_testmode==1
mio_ccu_vreg_selbg_l	I	1	static	Dedicated. Input from external IO through MIO – controls VREG input of PLL and RNG
mio_ccu_pll_clamp_fltr	I	1	static	Shared. Input from external IO through MIO – Used to control clamp filter input of PLL in test mode (mio_pll_testmode==1)
mio_ccu_pll_div2	I	[5:0]	async	Shared. Input from external IO through MIO – Used to program D2 of PLL in test mode (mio_pll_testmode==1)
mio_ccu_pll_div4	I	[7:0]	async	Shared. Input from external IO through MIO – Used to program D4 of PLL in test mode (mio_pll_testmode==1)
mio_ccu_pll_trst_l	I	1	async	Shared. Input from external IO through MIO – Used to reset PLL in test mode (mio_pll_testmode==1)
CCU-TCU Interface				
gl_ccu_clk_stop	I	1	cmp	Clock stop for cmp domain (providisional signal. as of now, no application for it)
gl_ccu_io_clk_stop	I	1	cmp	Clock stop for io domain (providisional signal. as of now, no application for it)
tcu_pce_ov	I	1	async	Overrides clock stop assertion (providisional signal. as of now, no application for it)
tcu_ccu_mux_sel	I	[1:0]	cmp	Controls PLL muxes from TCU – one of 4 signals to gclk tree inputs: PLL VCO, sysclk, bypass clock, or stretched clock.
tcu_ccu_ext_cmp_clk	I	1	N/A	Bypass clock input for CMP clk (muxed with TCK) from TCU
tcu_ccu_ext_dr_clk	I	1	N/A	Bypass clock input for DR clk (muxed with TCK) from TCU
tcu_ccu_clk_stretch	I	1	cmp	Controls clock stretch in PLL
CCU-RST Interface				
rst_ccu_pll_	I	1	sys	Active low PLL reset – once de-asserted, PLL will start to lock
rst_ccu_	I	1	sys	Active low reset for CCU logic – staggered with respect to rst_ccu_pll_
ccu_sys_cmp_sync_en	O	1	cmp	Special sync pulse for sys -> cmp clk domain – ONLY for RST cluster

TABLE 5-1 CCU Port Listing *(Continued)*

Name	Dir	Width	Domain	Description
ccu_cmp_sys_sync_en	O	1	cmp	Special sync pulse for cmp -> sys clk domain – ONLY for RST cluster
ccu_rst_sys_clk	O	1	N/A	Provides buffered version of sysclk that the PLL is running off
ccu_rst_sync_stable	O	1	cmp	When asserted after PLL has finished locking, indicates to RST block that all clocks and sync pulses are stable
ccu_rst_change	O	1	io	When asserted, indicates to the RST block that the pll divider values WILL change. **NOTE. CCU does NOT PERFORM an actual check of old and new divider values. It relies solely on the value of CHANGE field in PLL_REG for the RST to determine if PLL lock required**
rst_wmr_protect	I	1	async	Prepares CCU for a warm reset
cluster_arst_l	I	1	async	Holds cluster header output clock low

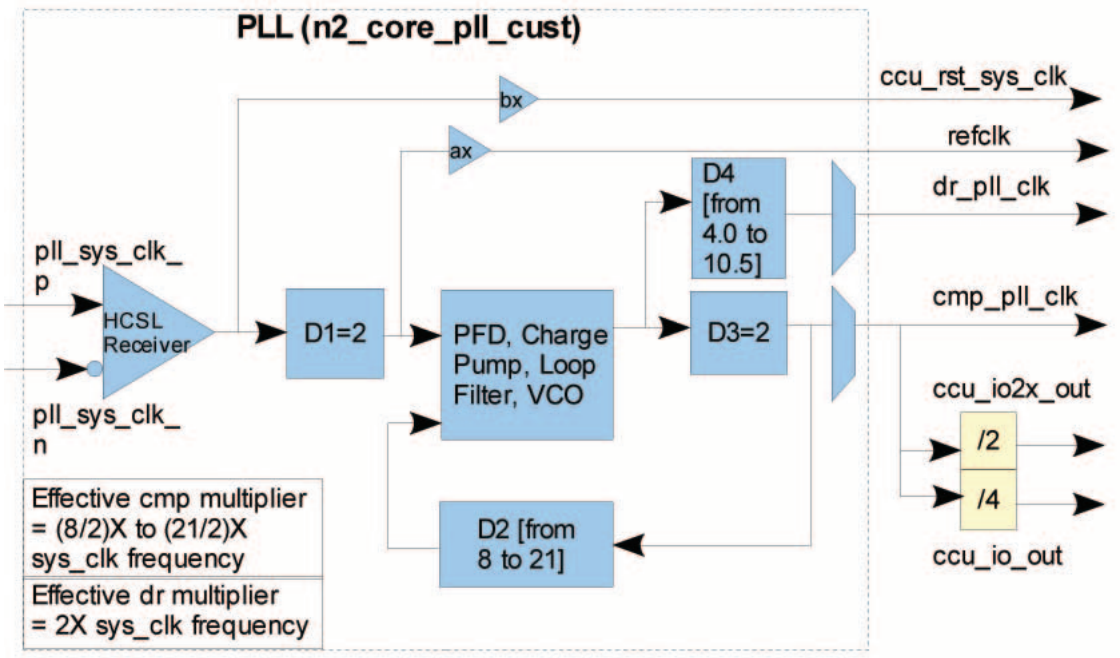
5.2.1 Clock Generation and Distribution

5.2.1.1 Generation

There is one PLL in the CCU for generating both core and memory clocks. The SPARC cores, CCX and L2 cache operate at the **cmp** frequency. Parts of the chip are on the **io** and **io2x** domains, both of them derived directly from the **cmp** clock. The other clock that goes into the MCU for interfacing with FBDIMM's is the **dr clock.**

The same system reference clock is **pll_sys_clk_p/n**. This is a differential input that is fed from the bumps directly into the core pll. There are 2 independent dividers that generate cmp and dr clocks, which are rational multiples of each other. Hence, they are *ratioed synchronous*. The following diagram gives a simplistic representation of how the PLL generates these clocks.

FIGURE 5-3 PLL Clock Generation during Mission Mode



The other clocks, **io_clk** and **io2x_clk,** are derived from the cmp clock, by generating divided down (by 4 and 2 respectively), phase signals. They are distributed as clocks from *cluster header* outputs. The distribution is discussed in more detail in Section 5.2.4, "Distribution" on page 5-13.

Note that the PLL feedback loop is entirely self-contained within the PLL. Thus there is arbitrary phase difference between the rising edge of a sys_clk cycle and a rising edge of a clock into the CLK input of a flop in any cluster.

5.2.2 PLL Programming

The PLL is programmed through a combination of registers (CSR fields), direct chip-level pin control and combinational logic. The CSR based controls are covered in Section 5.6, "CSR Block" on page 5-29. External pin-level control is applicable typically in test mode, and is covered in Section 5.7, "CCU TESTABILITY" on page 5-32. This sub-section focuses on divider configuration (CSR programmable) and combinational mux controls from the TCU.

Dividers D1, D2 and D3 perform integer division. D4 has fractional divide capability in discrete increments of 0.5 by using both phases of the VCO clock. The divider configurations allow cmp_pll_clk to run at different multiples of pll_sys_clk, but dr_pll_clk is always twice as fast as pll_sys_clk. The DR clock output may not have 50/50 duty cycle, but should be within +/-10%. This is not an issue within OpenSPARC T2 since there is no operation on the low phase.

The divider values are summarized in TABLE 5-2 with information on both effective and actual bits.

TABLE 5-2 PLL Divider Programmation for Mission Mode

Div	Bits	(Effective) Valid Range	Binary Encoded Values	Comments
D1	6	2	00_0001	Binary value = Effective value – 1
D2	6	8 — 21	00_0111 – 01_0100	Binary value = Effective value – 1
D3	6	2	00_0001	Binary value = Effective value – 1
D4	7	4.0 — 10.5	00_0100_0 – 00_1010_1	Binary value [6:1] = Effective value; bit [0] = 0 for integer effective, and 1 for effective x.5

Even though all 4 dividers can be programmed via CSR writes, there is a subset of values that are valid. D3, for example, needs to be set to divide by 2. Putting a divide by 3 or higher will result in a non 50/50 duty cycle cmp clock. dr_pll_clk may not be produced correctly since it uses both phases of the VCO clock. Acceptable values for normal operating or mission mode with corresponding clock frequencies are given in Table 4.

The clock frequency multiplication equations with respect to the external oscillator output (sys_clk) are shown.

fvco = (D2 X D3 /D1) fsys
fcmp = (1 /D3) fvco = (D2 /D1) fsys
fdr = (1 /D4) fvco = (D2 X D3)/(D1 X D4) fsys
fio = fcmp = (D2 /4D1) fsys
fio2x = fcmp = (D2 /2D1) fsys

The first row in any of the 3 sets in TABLE 5-3 holds the default divider ratio during power-on-reset. The rows in blue (14, 10 and 7) of the 3 sets refer to the targeted operating frequencies. Grayed out sections are beyond the scope of expected operation, even though within the CCU there is no check for these configurations.

TABLE 5-3 Clock Frequency Table in Mission Mode

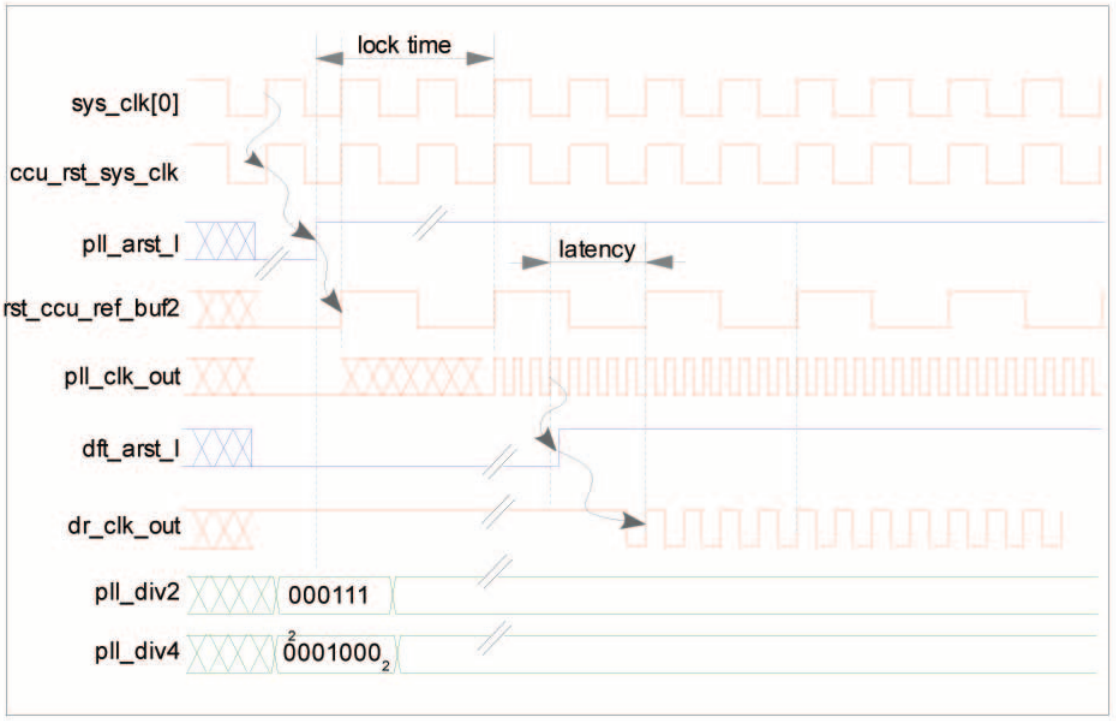
No.	sys_clk (MHz)	D1	D2	D3	D4	D2*D3	VCO (MHz)	cmp_clk (MHz)	io_clk (MHz)	io2x_clk (MHz)	r_clk (MHz)	cmp:dr (ratio)
1	133.33	2	8	2	4.00	16.00	1066.67	533.33	133.33	266.67	266.67	2.00
2	133.33	2	9	2	4.50	18.00	1200.00	600.00	150.00	300.00	266.67	2.25
3	133.33	2	10	2	5.00	20.00	1333.33	666.67	166.67	333.33	266.67	2.50
4	133.33	2	11	2	5.50	22.00	1466.67	733.33	183.33	366.67	266.67	2.75
6	133.33	2	12	2	6.00	24.00	1600.00	800.00	200.00	400.00	266.67	3.00
6	133.33	2	13	2	6.50	26.00	1733.33	866.67	216.67	433.33	266.67	3.25
7	133.33	2	14	2	7.00	28.00	1866.67	933.33	233.33	466.67	266.67	3.50
8	133.33	2	15	2	7.50	30.00	2000.00	1000.00	250.00	500.00	266.67	3.75
8	133.33	2	16	2	8.00	32.00	2133.33	1066.67	266.67	533.33	266.67	4.00
10	133.33	2	17	2	8.50	34.00	2266.67	1133.33	283.33	566.67	266.67	4.25
11	133.33	2	18	2	9.00	36.00	2400.00	1200.00	300.00	600.00	266.67	4.50
12	133.33	2	19	2	9.50	38.00	2533.33	1266.67	316.67	633.33	266.67	4.75
13	133.33	2	20	2	10.00	40.00	2666.67	1333.33	333.33	666.67	266.67	5.00
14	133.33	2	21	2	10.50	42.00	2800.00	1400.00	350.00	700.00	266.67	5.25
1	166.67	2	8	2	4.00	16.00	1333.33	666.67	166.67	333.33	333.33	2.00
2	166.67	2	9	2	4.50	18.00	1500.00	750.00	187.50	375.00	333.33	2.25
3	166.67	2	10	2	5.00	20.00	1666.67	833.33	208.33	416.67	333.33	2.50
4	166.67	2	11	2	5.50	22.00	1833.33	916.67	229.17	458.33	333.33	2.75
6	166.67	2	12	2	6.00	24.00	2000.00	1000.00	250.00	500.00	333.33	3.00
6	166.67	2	13	2	6.50	26.00	2166.67	1083.33	270.83	541.67	333.33	3.25
7	166.67	2	14	2	7.00	28.00	2333.33	1166.67	291.67	583.33	333.33	3.50
8	166.67	2	15	2	7.50	30.00	2500.00	1250.00	312.50	625.00	333.33	3.75
8	166.67	2	16	2	8.00	32.00	2666.67	1333.33	333.33	666.67	333.33	4.00
10	166.67	2	17	2	8.50	34.00	2833.33	1416.67	354.17	708.33	333.33	4.25
11	166.67	2	18	2	9.00	36.00	3000.00	1500.00	375.00	750.00	333.33	4.50

TABLE 5-3 Clock Frequency Table in Mission Mode *(Continued)*

No.	sys_clk (MHz)	D1	D2	D3	D4	D2*D3	VCO (MHz)	cmp_clk (MHz)	io_clk (MHz)	io2x_clk (MHz)	r_clk (MHz)	cmp:dr (ratio)
12	166.67	2	19	2	9.50	38.00	3166.67	1583.33	395.83	791.67	333.33	4.75
13	166.67	2	20	2	10.00	40.00	3333.33	1666.67	416.67	833.33	333.33	5.00
14	166.67	2	21	2	10.50	42.00	3500.00	1750.00	437.50	875.00	333.33	5.25
1	200	2	8	2	4.00	16.00	1600.00	800.00	200.00	400.00	400	2.00
2	200	2	9	2	4.50	18.00	1800.00	900.00	225.00	450.00	400	2.25
3	200	2	10	2	5.00	20.00	2000.00	1000.00	250.00	500.00	400	2.50
4	200	2	11	2	5.50	22.00	2200.00	1100.00	275.00	550.00	400	2.75
6	200	2	12	2	6.00	24.00	2400.00	1200.00	300.00	600.00	400	3.00
6	200	2	13	2	6.50	26.00	2600.00	1300.00	325.00	650.00	400	3.25
7	200	2	14	2	7.00	28.00	2800.00	1400.00	350.00	700.00	400	3.50
8	200	2	15	2	7.50	30.00	3000.00	1500.00	375.00	750.00	400	3.75
8	200	2	16	2	8.00	32.00	3200.00	1600.00	400.00	800.00	400	4.00
10	200	2	17	2	8.50	34.00	3400.00	1700.00	425.00	850.00	400	4.25
11	200	2	18	2	9.00	36.00	3600.00	1800.00	450.00	900.00	400	4.50
12	200	2	19	2	9.50	38.00	3800.00	1900.00	475.00	950.00	400	4.75
13	200	2	20	2	10.00	40.00	4000.00	2000.00	500.00	1000.00	400	5.00

PLL output clock behavior with respect to its reset signals and sys_clk are shown. They are produced independently of the PLL specification. The example FIGURE 5-4 assumes D1 and D3 are left in their default states (effective value of 2 for both).

FIGURE 5-4 PLL Clocking Waveforms



5.2.3 PLL Mux Control

All functional clock muxing in OpenSPARC T2 is performed in custom blocks – the PLL, cluster headers and specialized L1 headers. However, the PLL clock mux control logic is divided between the CCU and the PLL hard macro, with the CCU

often exercising additional constraints. The combination of CCU inputs, and equivalent mappings to PLL inputs are tabulated inTABLE 5-4. For actual usage related to various modes of operation, refer to the Testability sections of this MAS.

TABLE 5-4 CCU and PLL Mapping

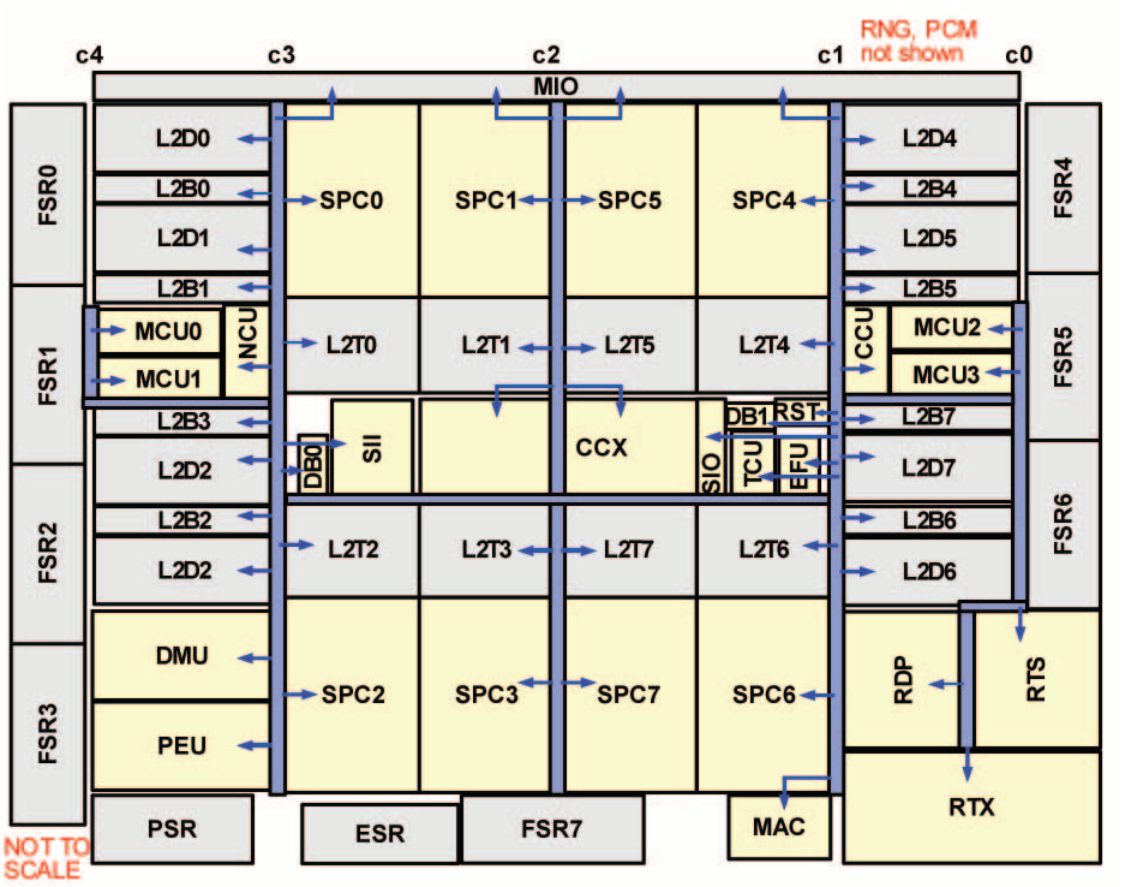
CCU or CSR inputs				PLL inputs				PLL outputs		
Mode	dtm[1,2]	atpg_mode	mux_sel	pll_arst_l	pll_bypass	pll_sel_a	pll_dtm	dr_sel_a	pll_clk_out	dr_clk_out
Func	0	0	00	rst_ccu_pll_	0	00	0	00	sys_clk x N	sys_clk x M
Str	0	0	01	rst_ccu_pll_	0	01	0	01	sys_clk x N (str)	sys_clk x M (str)
ATPG	0	1	10	0	1	10	1	10	ext_cmp_clk	ext_dr_clk
Byp	0	1	11	0	1	11	1	11	sys_clk_p	sys_clk_p
DTM	1	0	00	rst_ccu_pll_	0	00	1	11	sys_clk x N	sys_clk_p
MTest	0	0	11	0	1	10	1	10	ext_cmp_clk	ext_dr_clk

FIGURE 5-3 and FIGURE 5-19 complement the information in the table. Note that the PLL input and output signals above will be allowed to change based on any sequential logic within the CCU or PLL. For example, if DTM 1 or 2 mode is programmed into the CSRs, during PLL reset, *pll_bypass* will be held high to avoid internal PLL clock mux contention, and then set to 0 upon reset release. Under all other conditions, *pll_bypass* will simply be assigned to *tcu_ccu_mux_sel[1]*.

5.2.4 Distribution

As shown in FIGURE 5-2, **cmp_clk** and **dr_clk** are distributed via a global clock tree to **gclk** inputs of various clusters. Each cluster receives the same phase of gclk. The CCU also sends out a few control signals that are distributed closely with gclk's and pipelined on various tap points of the global clock tree. A high level diagram is shown in FIGURE 5-5.

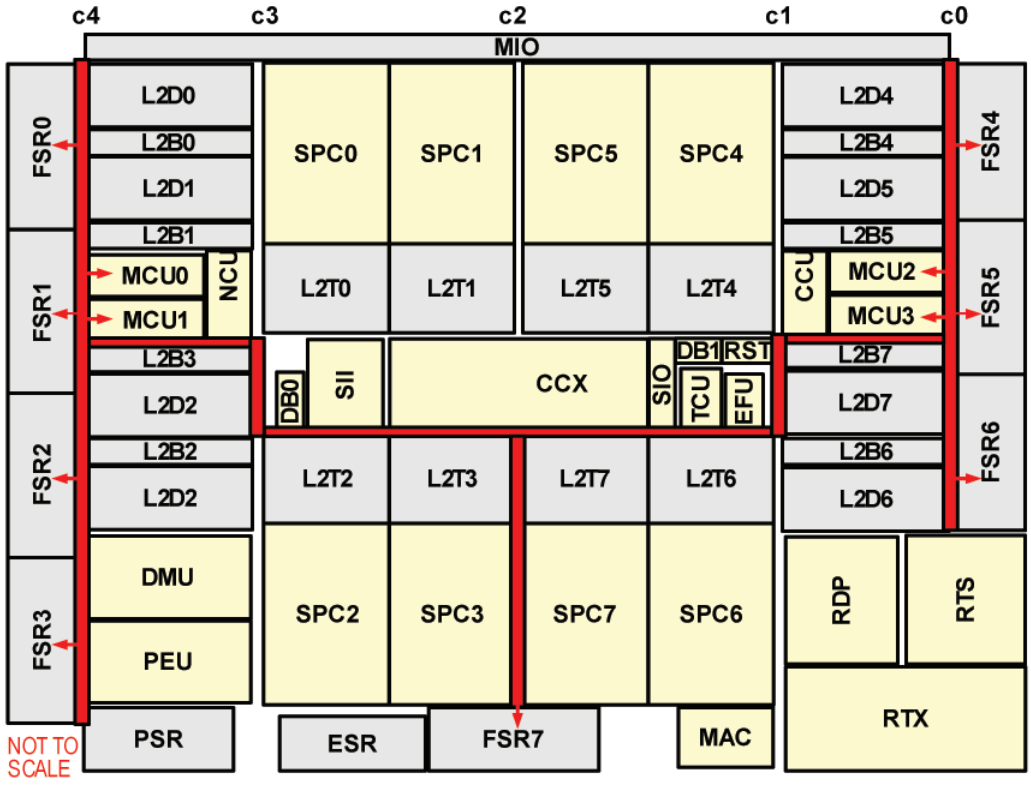
FIGURE 5-5 Simplified Global Distribution of CMP Clock



There are a set of other global signals from the CCU such as the divider phase signals, and sync pulses that are staged in the global clock tree, along with reset lines from the RST and clock stops from the TCU. All these staged signals are sent out on the CMP domain, or in some cases on the DR domain. Synchronization to the gridded clock outputs is performed in the cluster header, as described in the Usage document.

For completeness and references to it in other sections, DR distribution is shown in FIGURE 5-6.

FIGURE 5-6 Global Distribution of the DR Clock



5.3 Clock and Reset Inside CCU

5.3.1 Clock Domains

There are 3 clock domains within the CCU: L2, IOL2, and CMP_PLL. There is a distinction between CMP_PLL and L2 domains. The L2 domain is synchronous to all other clusters. CMP_PLL domain is the result of using the PLL output clock at CMP rate prior to distributing the clock through the GCLK macro. L2 clock is a phase shifted version of CMP_PLL clock ; the phase shift due to distribution can vary from process to process from 0.5 to ~1.5 CMP periods.

Note. There is temporarily a 4^{th} clock domain (SYSCLK) to work around missing latch models in the std cell library. Once the latch becomes available, SYSCLK domain will be removed.

A breakdown of the CCU by clock domain and approximate functionality appears in FIGURE 5-7. FIGURE 5-8 details the clock align detection logic since it uses non-conventional clocking and has a few special constraints:

The first pair of flops act as synchronizers should the negative edge of the reference clock be sampled by the flops (the outome of the align detection is immaterial since it is zero in this case).

There is a half-cycle, or 2X clockig path where data from negative clocked flops gets transferred to positive edge-triggered flops.

5.3.2 Reset Scheme

The CCU relies on the RST block for explicit reset signals, and does not operate via flush reset. Also, it needs to be released from reset before all other blocks on the chip. One reset is solely for the PLL, and the other for the remaining CCU logic, loosely speaking. The CCU itself needs to generate one or two staggered resets. These resets work in a domino like fashion to ultimately provide a signal to the RST unit that indicates the CCU is done with initialization, and that the RST block may release the rest of the chip from reset. This signal is ccu_rst_sync_stable. When the signal goes high, all clocks from the CCU are valid, at the correct frequency, and all sync pulses are operating in their proper positions.

Depending on whether clocks may be stable or not, the CCU needs to use either asynchronous or synchronous reset. However, all resets within the CCU are released synchronously. Emphasis has been placed on determinism and repeatability, so even where brute-force synchronization is used, additional signals ensure determinism.

There is only 1 CSR register in the CCU that is warm reset protected (see section 9.0). All clock generating and pll programmation bits are warm reset protected. The rest are not.

5.3.3 Initialization Sequence

The Power-On-Reset scheme in the CCU is highlighted by the waveforms in FIGURE 5-9.

For functional operation, the CCU is activated in a very simple manner. There are two resets to the CCU, ccu_rst_pll_ and ccu_rst_ that need to be applied in a sequence. Testmode, and divider_bypass pins need to be held low. When the CCU

sends asserts ccu_rst_sync_stable to the RST block, all clocks and sync pulses are being generated correctly, and the RST may .... An explanation of the variousnumbered parameters is given.

TABLE 5-5 Key Parameters in Initialization Sequence

Parm #	Description	Duration
1	Time taken for first rising edge of refclk to appear from release of rst_ccu_pll_	<1 sys_clk cycle
2	Deassertion of rst_ccu_pll_ to rising edge of stable CMP PLL clock ouput	LOCK TIME
3	Clock distribution delay of global clock tree from PLL output to gclk input of cluster header	~0.5 – ~1.3 CMP cycle
4	Deassertion of rst_ccu_ to gclk_rst_n (requires use of brute force synchronizer)	1 to 2 CMP cycles
5	Rising edge of refclk to assertion of aligned_shift pulse.	3 CMP cycles
6	Shift of aligned_shift pulse to create VCO aligned	4 to 17 CMP cycles depending on pll_div2[5:0]
7	Transfer of aligned signal from CMP PLL domain to CMP_GCLK domain.	Tracks parameter #3
8	From first aligned pulse to aligned_rst_n signal for internal CCU blocks for coherent reset release.	1 CMP cycle
9	Deassertion of aligned_rst_n to first rising edge of ccu_io2x_out	2 CMP cycles
10	Deassertion of aligned_rst_n to first rising edge of ccu_io_out	4 CMP cycles
11	Time when aligned == 1 to deassertion of divider for generating DR clock within PLL	2-3 CMP cycles depending on pll_div4[6:0]
12	Deassertion of dft_a_rst_l to first rising edge of dr_clk	5-6 CMP cycles depending on pll_div4[6:0]

FIGURE 5-7 CCU Clock Domains and Function

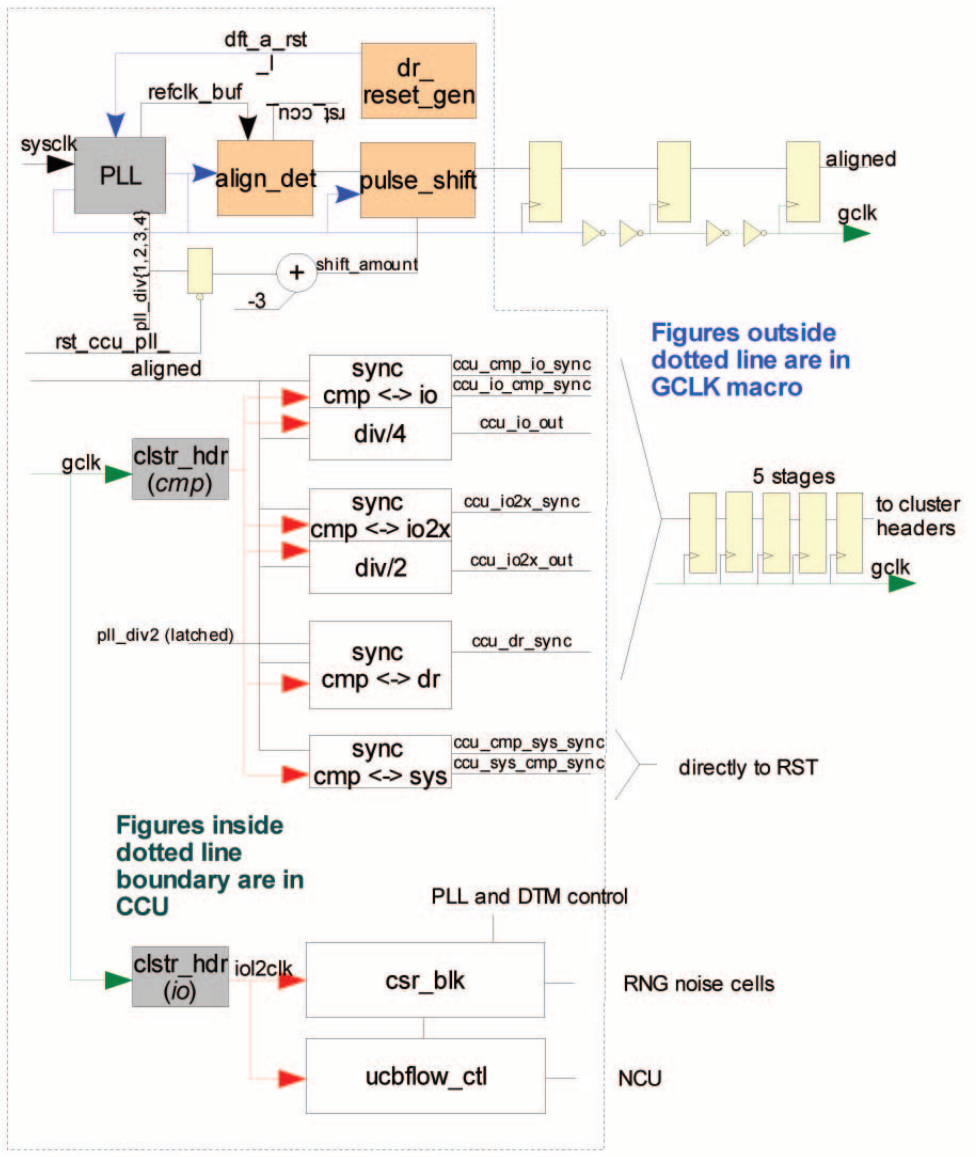


FIGURE 5-8 Align Detection Circuitry

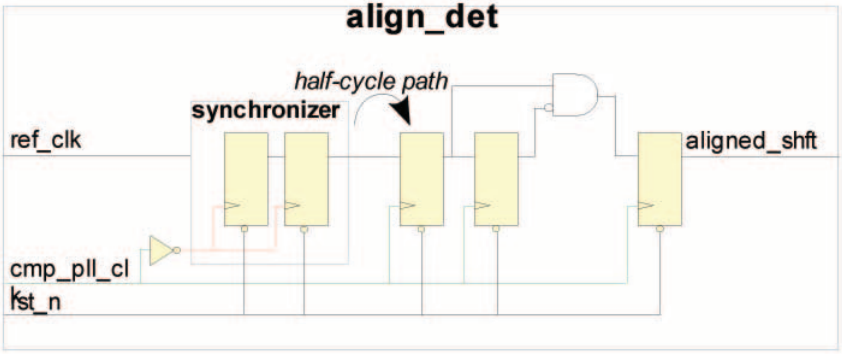
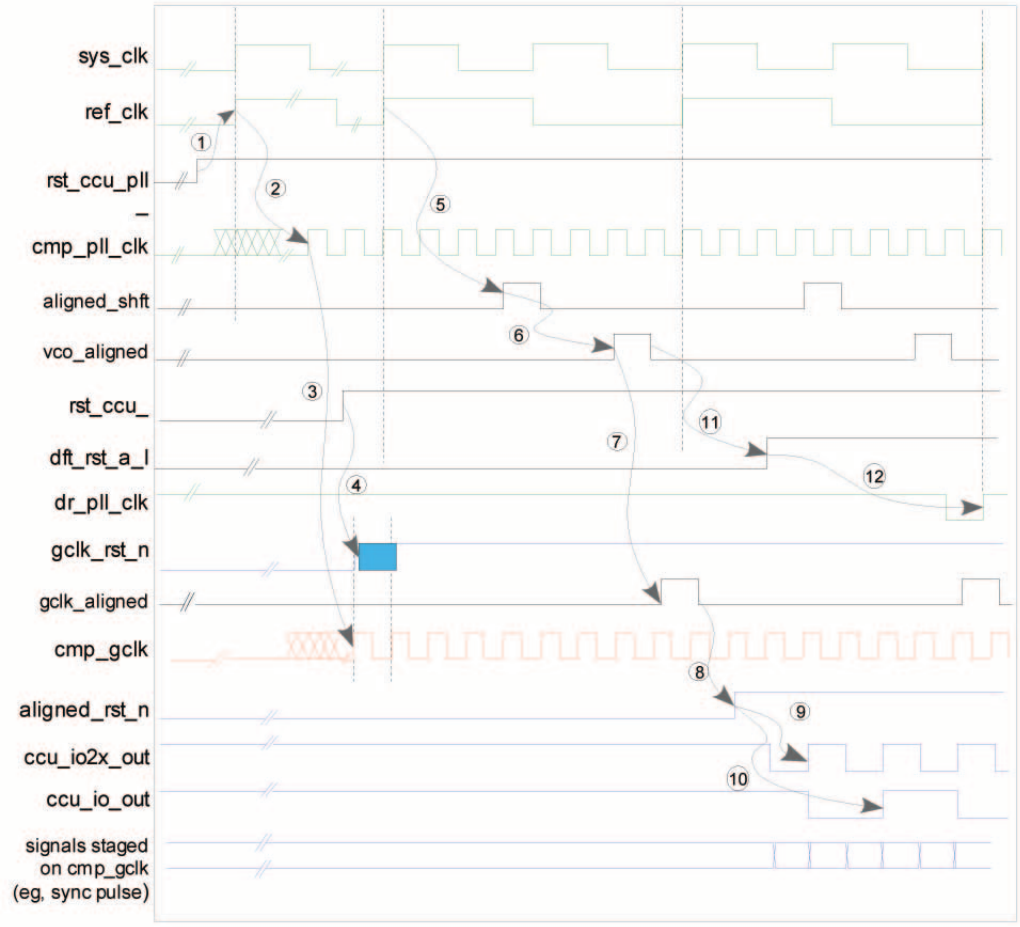


FIGURE 5-9 Initialization Sequence for CCU Clocks



5.4 SYNC Pulses

The main application of generating synchronization pulses in OpenSPARC T2 is to allow low latency, deterministic data transfer between *ratioed synchronous* clock domains. The key requirements for this scheme to work are:

A single reference clock source.

PLLs that have similar behavior, in particular a known input-output phase relationship.

The clock frequencies need to be rational multiples of each other, or *ratioed synchronous*

Jitter, skew, and other PVT mismatches are taken into account to ensure setup and hold requirements are met during domain crossing.

Clock domains that are of primary concern are the CMP and DR domains. Synchronization between cmp and IO, or IO2X domains is a simpler problem, but handled similarly.

5.4.1 Proposed Scheme

The following circuit shows the proposed scheme for clock domain transfers.

FIGURE 5-10 CMP to DR Synchronization

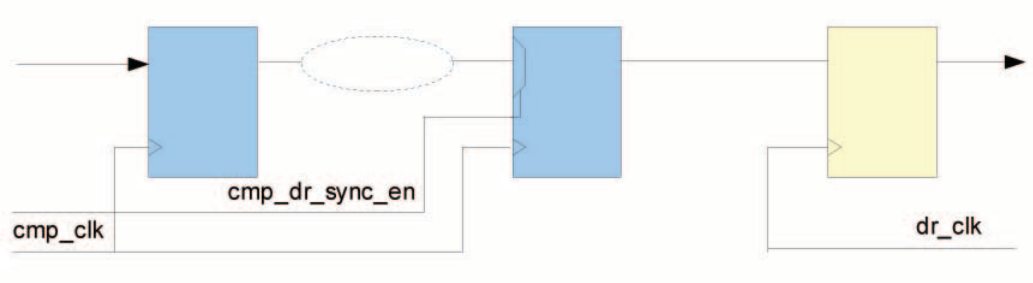
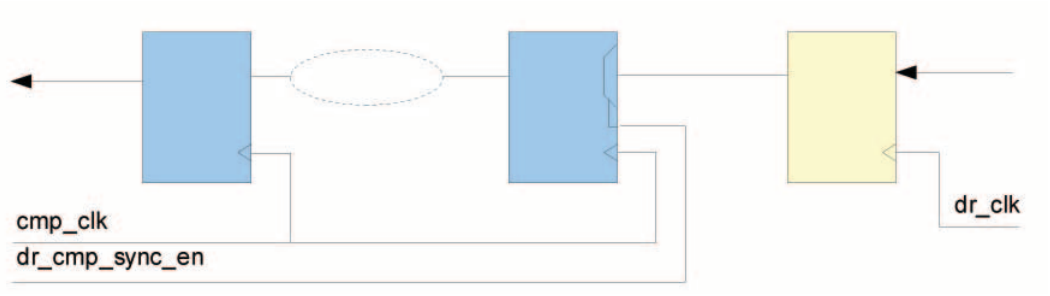


FIGURE 5-11 DR to CMP Synchronization



It has been borrowed from past designs and modified. All it does is allow data to cross one domain to another during a safe interval, avoiding setup and hold problems. The mechanism for operation for fast clock (eg, cmp) to slow clock (eg, dr) domain is as follows:

Mux enable to launch flip-flop is generated on cmp_clk.

Next cmp rising edge, data is launched.

Data is captured on dr_clk.

For slow clock to fast clock transfers, the procedure is:

Data is launched on rising edge of dr_clk.

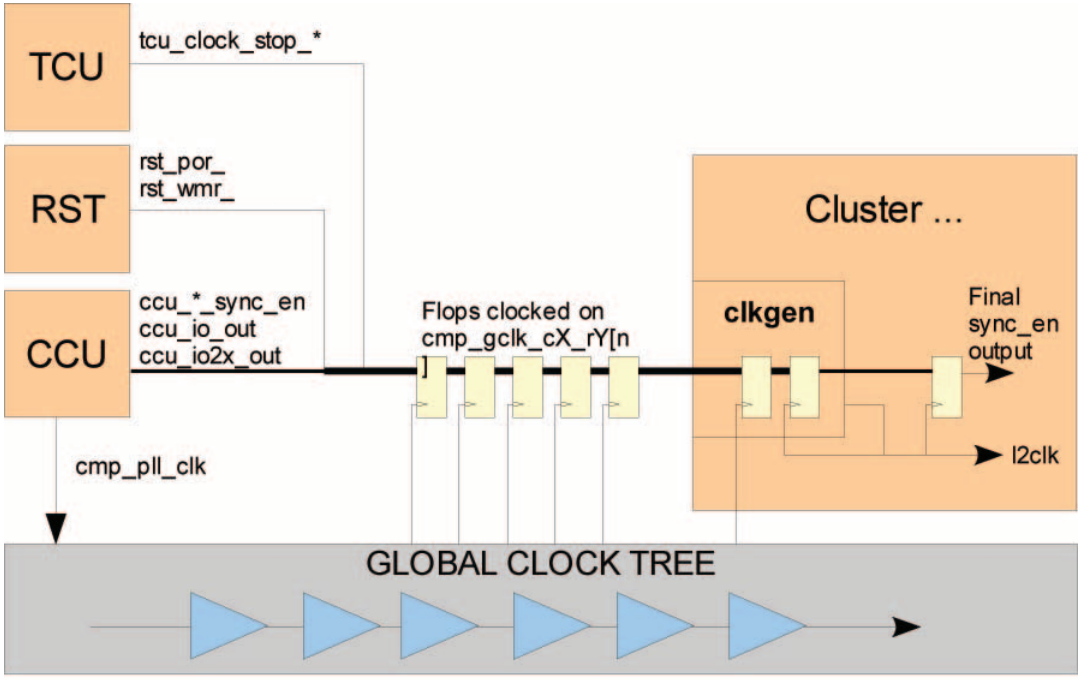
Mux enable to capture flip-flop is generated on cmp_clk.

Next cmp rising edge, data is captured.

In both cases, the rate of communication is limited by the slower clock frequency, so the enable is generated once every slow clock cycle. The main challenge is to determine the ideal intervals between pulse generation for robust operation. For a discussion on determining the positions, refer to the Appendices.

5.4.2 Sync Pulse Distribution

FIGURE 5-12 Logical Representation of Sync Pulse Global Distribution



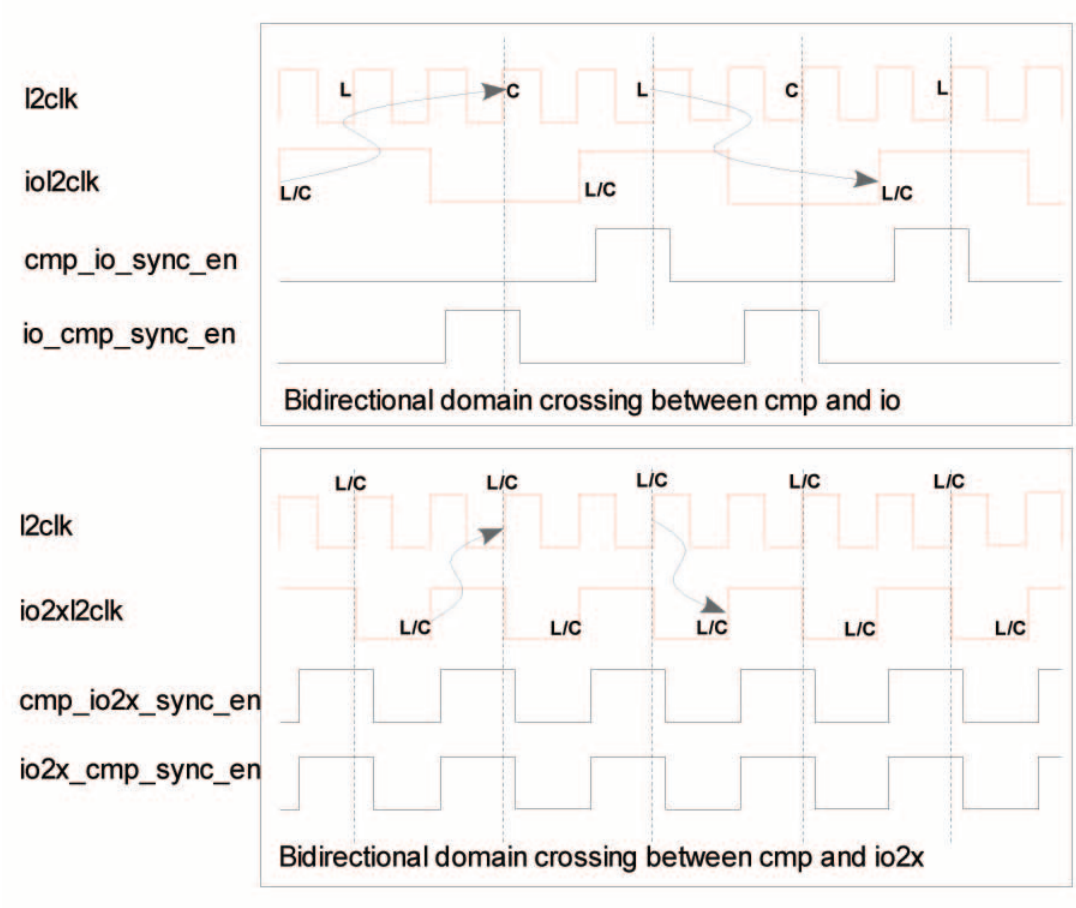
Sync pulses will be generated in the CCU on the cmp_gclk domain, and be distributed (along with other control signals) in 5 stages of pipeline in mini-clusters to each cluster header. In the cluster headers, there will be one more stage of latching the data on the gclk domain. From there, each cluster will flop the enables on the l2clk domains before local distribution. In effect, there will be 7 stages of cmp_cycle before sync pulses are output from cluster headers, and then flopped one last time within clusters.

5.4.3 CMP to IO/IO2X Waveforms

Domain crossing between CMP and IO/IO2X domains is a special, and simpler case of CMP to DR communication because cmp_clk is an integer multiple of io_clk and io2x_clk, and both io_clk and io2x_clk are directly derived from cmp_clk

FIGURE 5-13 shows the **actual** usage, ie, the ***final sync_en output*** (refer to FIGURE 5-9).

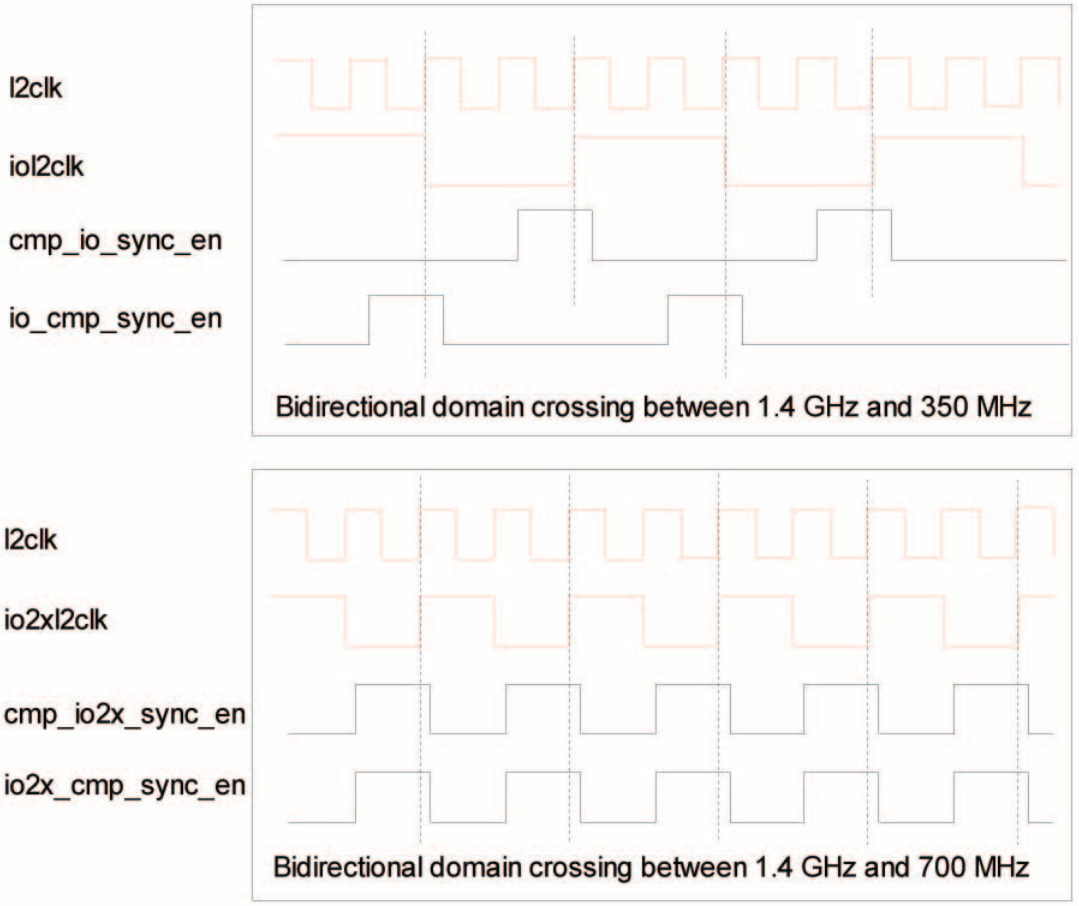
FIGURE 5-13 Actual Usage of Sync Pulses at Enable Pin of Transfer Flops (all transfer arrows not shown)



Note – Since *cmp_io2x_sync_en* and *io2x_cmp_sync_en* are shown at the point of usage; however, they would both be driven by a single source – *cluster header->io2x_sync_en ->flop output*.

For clarity, the outputs of cluster headers are also shown. These are, as expected from FIGURE 5-9, one l2clk cycle early.

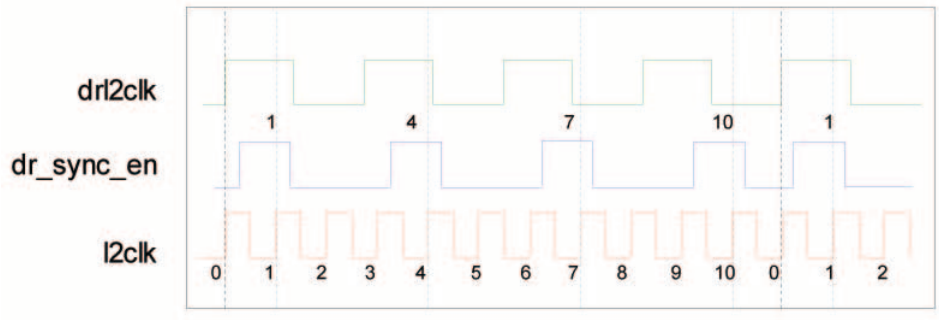
FIGURE 5-14 Sync Enable Positions at the Outputs of Cluster Headers (prior to being latched)



5.4.4 CMP/DR Pulses

CMP to DR pulse positions are determined by the amount of uncertainty that can exist between cmp_clk and dr_clks. A discussion on the procedure of determining the positions appears in the Appendix. There are several documents detailing the sync pulse schemes and timing budgets that have been created to ensure robustness. An example of the positions of the dr sync pulses is shown in FIGURE 5-15.

FIGURE 5-15 Sync Pulse Example for fCMP:fDR = 11:4



The convention is to describe the sync pulse position in terms of cmp clk phases, with phase 0 being set to the nominal alignment of cmp and dr clocks. The sync pulse positions at the point of domain crossing are given in TABLE 5-6.

TABLE 5-6 DR<->CMP Sync Pulse Positions

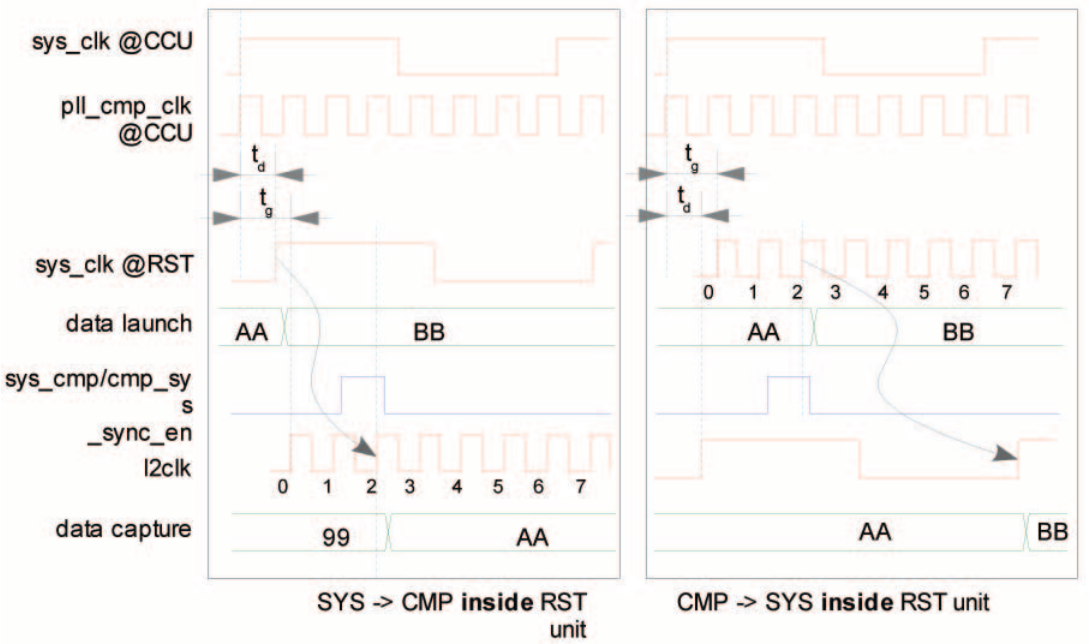
				CMP<->DR Transfer Edge K - > clk cycles				Transfer phase (normalized for 4 dr=2pi) K - > clk cycles			
N	M	Meff	N/M	0	1	2	3	0	1	2	3
8	4	1	2.00	1	1	1	1	1	3	5	7
9	4	4	2.25	1	3	6	8	1	3	6	8
10	4	2	2.50	1	4	1	4	1	4	6	9
11	4	4	2.75	1	4	7	10	1	4	7	10
12	4	1	3.00	1	1	1	1	1	4	7	10
13	4	4	3.25	2	5	8	11	2	5	8	11
14	4	2	3.50	2	5	2	5	2	5	9	12
15	4	4	3.75	2	6	9	13	2	6	9	13
16	4	1	4.00	2	2	2	2	2	6	10	14
17	4	4	4.25	2	6	11	15	2	6	11	15
18	4	2	4.50	2	7	2	7	2	7	11	16
19	4	4	4.75	2	7	12	17	2	7	12	17
20	4	1	5.00	2	2	2	2	2	7	12	17
21	4	4	5.25	3	8	13	18	3	8	13	18

5.4.5 CMP/SYS Pulses

There are a pair of sync pulses between CMP and SYS_CLK strictly for the RST unit. These pulses are not staged on the global clock tree, and not taken in through cluster headers. However, to account for fanout, the signals are flopped twice inside the RST cluster. The scheme relies on the RST block being placed close to the CCU; there is tolerance built in for skew between the CMP and SYS_CLK up to a couple of CMP cycles.

The active position of the sync pulse ("1" on rising edge of cmp_clk) will be on phase 2 of l2clk. This will provide ample margin, > 1 *fast* cmp cycle for setup or hold. Illustrations of data transfers in both directions are shown in FIGURE 5-16. For quanitfication of the amount of margin available, refer to the appendix.

FIGURE 5-16 Domain Crossing using Sync Pulses in RST



5.5 RNG Description

The random number generator (rng) generates random numbers from 3 noise cells. There is one rng block (and LFSR) to be shared amongst the 8 processor cores. Only one of the cells may be active at a time, all three may be active, or none of them may be active. Any other combination defaults to selecting all three noise cells. The following encoding applies:

TABLE 5-7 Encoding for Noise Cell Selection

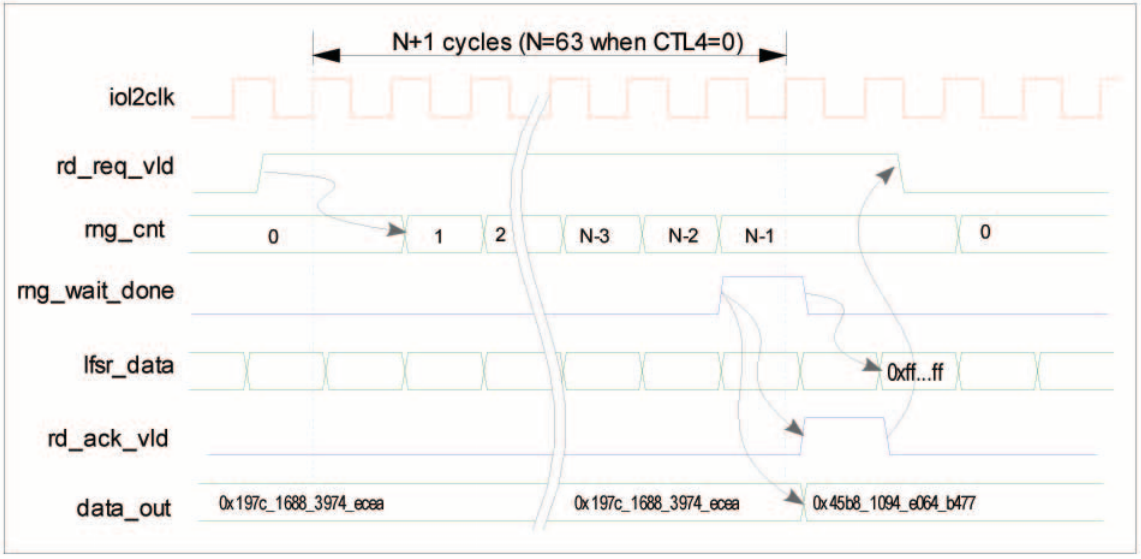
CTL3	CTL2	CTL1	Effect
0	0	0	Deselect all noise cells (feeds 0 into LFSR)
0	0	1	Select noise cell 1
0	1	0	Select noise cell 2
1	0	0	Select noise cell3
011, 101, 110, 111			Select all 3 noise cells

Every clock cycle, the XOR of the outputs of the selected noise cells is fed into a 64-bit register. Under functional mode, the register generates data by implementing the CRC-polynomial

$$P(x) = x^{64} + x^{61} + x^{57} + x^{56} + x^{52} + x^{51} + x^{50} + x^{48} + x^{47} + x^{46} + x^{43} + x^{42} + x^{41} + x^{39} + x^{38} + x^{37} + x^{35} + x^{32} + x^{28} + x^{25} + x^{22} + x^{21} + x^{17} + x^{15} + x^{13} + x^{12} + x^{11} + x^{7} + x^{5} + x + 1$$

After each read request, it is important to not maintain any correlation with the past generated values, so the LFSR will be flushed after every read acknowledge. The register will be flushed with a non-zero state 0xFFFF_FFFF_FFFF_FFFF. Also, multiple requests for rng_data are automatically separated by N+2 cycles, where N can be programmed by writing to the 16-bit field **rng_wait_cnt** in the CSR register.

FIGURE 5-17 Read Access Operation of rng_data via Memory Mapped Address

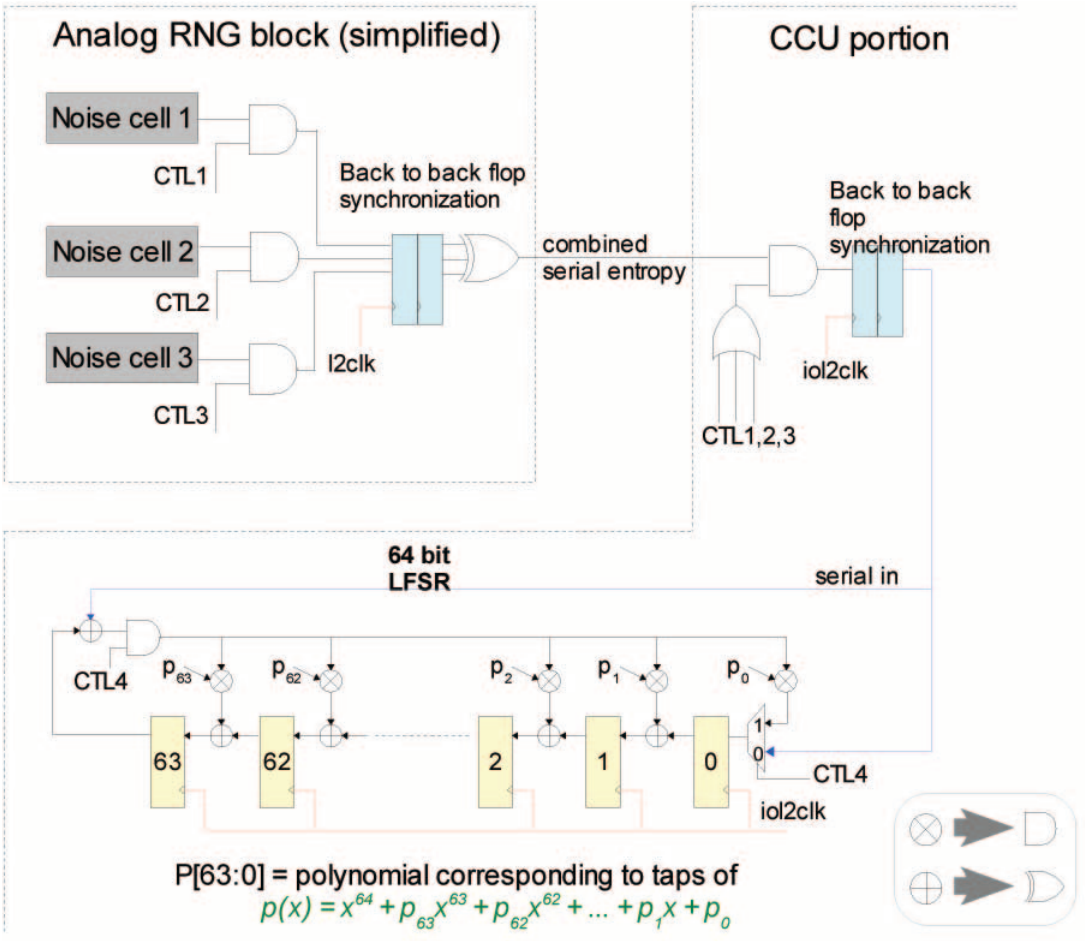


In diagnostic mode (CTL4 = 0), the LFSR acts as a simple shift register capturing the noise cell output directly, determined independently by CTL1 CTL2 and CTL3 as per encoding. The additional constraint in this mode is that successive read requests for the *RNG_DATA* will be delayed by 64 iol2clk cycles. Also, flushing the LFSR after every read will be disabled in this mode.

The nominal frequency of the osciallator in each noise cell can be set independently by programming the **rng_vco_ctrl[1:0]** field. There are 4 settings that correspond to 4 different frequencies; however, each cell must be programmed one at a time. As an example, consider the following desired configuration: noise cell1 -> 00 setting, cell2 -> 10 setting, cell 3 --> 01 setting, and observe all 3 cells. One would proceed as follows:

1. Set CTL3,CTL2,CTL1 = 001 and set RNG_VCO_CTRL = 00
2. Set CTL3,CTL2,CTL1 = 010 and set RNG_VCO_CTRL = 10
3. Set CTL3,CTL2,CTL1 = 100 and set RNG_VCO_CTRL = 01

FIGURE 5-18 Entropy Generator Design



5.6 CSR Block

The CSR block consists of registers that can be used for programming other CCU blocks, and for accessing information. This includes both functional and test related data. The other part of the CSR block communicates with the standard UCB interface.

Note that values written into the PLL_CTL register will not take effect immidiately (even though reading them back will show the new values). A warm reset needs to be applied to affect clocks.

5.6.1 PLL_CTL (*0x83_0000_0000)*

TABLE 5-8 PLL Control Register

Field Name	Bits	Default	WMR Protected	R/W	Description
Reserved	63:37	0x0	N/A	R	Reserved
pll_clamp_fltr	36	0x0	YES	R/W	PLL clamp filter setting
st_delay_dr	34:35	0x0	YES	R/W	DR stretch delay setting (40ps intervals) 00 –> 40 , 01 -> 80 , 10 -> 120 , 11 -> 160
pll_char_in	33	0x0	YES	R/W	PLL characterization test input
change	32	0x1	YES	R/W	PLL frequency to be changed
align_shift	30:31	0x0	YES	R/W	Shift align detect point by [-1:1] cmp cycle. Affects dr_sync pulse generation. All other sync pulses unchanged. 00 -> no shift , 01 -> +1 cycle, 10 -> -1 cycle, 11 -> no shift.
serdes_dtm2	29	0x0	YES	R/W	Mode 2 – causes **ccu_serdes_dtm** to be asserted during reset. io, io2x set to DR rate. Used by DBG1/MIO for selecting setting mux controls.
serdes_dtm1	28	0x0	YES	R/W	Mode 1 – causes **ccu_serdes_dtm** to be asserted during reset. io, io2x set to DR rate. Used by DBG1/MIO for selecting setting mux controls.
st_delay_cmp	27:26	0x0	YES	R/W	CMP stretch delay setting (40ps intervals) 00 –> 40 , 01 -> 80 , 10 -> 120 , 11 -> 160
st_phase_hi	25	0x0	YES	R/W	High or low phase of clk to be stretched 0 indicates low phase.
pll_div4	24:18	0x8	YES	R/W	PLL VCO divider (D4) for dr. Refer to PLL programmation section.
pll_div3	17:12	0x1	YES	R/W	PLL VCO divider (D3) for cmp. Rrefer to PLL progrmmation section.
pll_div2	11:6	0x7	YES	R/W	PLL feedback divider (D2). Refer to PLL programmation section.
pll_div1	5:0	0x1	YES	R/W	PLL pre-scalar (D1). Refer to PLL progrmmation section.

5.6.2 RNG_CTL (0x83_0000_0020)

TABLE 5-9 RNG Control Register

Field Name	Bits	Default	WMR Protected	R/W	Description
Reserved	63:25	0x0	N/A	R	Reserved
rng_wait_cnt	24:9	0x003E	NO	R/W	Minimum wait time before successive RNG data is sent
rng_bypass	8	0x0	NO	R/W	rng_bypass=0 sets noise cell vco control voltage = output of feedback amplifier rng_bypass=1, sets noise cell vco control voltage = output of bias generator
rng_vcoctrl_sel	7:6	0x0	NO	R/W	pmos diode D/A setting bus. Controls VCO rate for each noise cell. Refer to RNG section for programming.
rng_anlg_sel	5:4	0x0	NO	R/W	Analog mux select for characterization
rng_ctl4	3	0x1	NO	R/W	Enables using LFSR or plain shift register. Set to LFSR mode by default.
rng_ctl3	2	0x1	NO	R/W	Control for using noise cell 3. Refer to RNG section for programming.
rng_ctl2	1	0x1	NO	R/W	Control for using noise cell 2. Refer to RNG section for programming.
rng_ctl1	0	0x1	NO	R/W	Control for using noise cell 1. Refer to RNG section for programming.

5.6.3 RNG_DATA (0x83_0000_0030)

TABLE 5-10 RNG Data Register

Field Name	Bits	Default	WMR Protected	R/W	Description
rng_data	63:0	x	N/A	R	64 bits of rng data

5.7 CCU TESTABILITY

This section deals with the testability of the CCU logic and the PLL.

5.7.1 CCU ATPG

The CCU logic is scannable only during ATPG testing. In mission mode, the scan chain input to the CCU is short-circuited to scan_out, and the following signals are set to zero: tcu_aclk, tcu_bclk, tcu_scan_en.

When testmode is set to 1, the TCU gets full control of the CCU, and treats the CCU just like any other logic on the chip being scanned. The TCU also controls the clock muxes within the PLL via tcu_ccu_mux_sel (refer to FIGURE 5-15. When tcu_ccu_mux_sel == 2b'10, the external clocks tcu_ccu_ext_cmp_clk and tcu_ccu_ext_dr_clk are muxed into the cmp_gclk and dr_gclk buffer trees respectively. These external clocks in turn are muxed inside the TCU with TCK, such that TCK can be forced onto both lines, or be controlled by the tester independently.

The only portion of the CCU that is not scannable is logic that does not run on the regular l2clk or iol2clk. This results in about a dozen flops that are kept out of the scan chain at all times.

PLL testability is covered in detail within the PLL spec, so it is not duplicated here.

The CCU PLL may be put in testmode, asserting the signal mio_pll_testmode via an external pin. This signal is independent of the testmode signal used for ATPG. However, **tcu_atpg_mode** has higher priority than **mio_pll_testmode.** For example, when both are asserted, **ccu_pll.pll_arst_l** will be set to 0.

With pll_testmode == 1, the CCU provides access to the PLL directly through the following signals from the MIO:

```
mio_ccu_vreg_selbg_l
mio_ccu_pll_clamp_fltr
mio_ccu_pll_div2
mio_ccu_pll_div4
mio_ccu_pll_trst_l
mio_ccu_pll_char_in
```

Likewise, it is possible to observe the internal signals from the PLL through a pair of muxed (internal to PLL) outputs ccu_mio_pll_char_out.[1:0].

When pll_testmode is active, rst_ccu_pll_ has no effect on resetting the PLL, and the CSR values for PLL control are overridden. The exception is the pll_char_in signal which is OR'ed with the PLL_REG bit 33 output.

5.8 Full Chip Testability

Section 10.0 describes the role of the CCU, and its features for supporting full chip testability.

5.8.1 Full Chip ATPG

The support provided by the CCU for full chip ATPG is no different from setting the CCU itself in ATPG mode. The same procedure for setting the CCU for ATPG mode is followed, while the CCU becomes merely a conduit for forwarding clocks.

The custom global clock tree does not perform any clock gating, so test clocks injected onto the main line are never blocked. Within the cluster header, in testmode, the clock stop signal is permanently disabled, ensuring that the test clocks into any cluster are free running with direct tester control. In addition, parts of the cluster header are fully scannable, while tcu_clk_stop and ccu_div_ph intputs of the header are observable.

Note that the sync pulses between all ratio'ed synchronous domains in each cluster would have to be set to a logic 1 to allow scan capture to take place consistently. This control is outside the scope of the CCU.

5.8.2 Transition Fault Test

During transition fault testing, the CCU needs to be fully functional, as do the cluster headers, since it would not be possible to apply at-speed scan capture pulses through the external clock ports from the tester.

They will be kept out of the scan chain by ensuring the external signal testmode = 0. The operation of tcu_clock_stop will be critical in ensuring t-fault testing is programmable to provide 2 or more high-speed pulses. The staging flops in the global clock tree macro, of course will be free running on gclk, and have no scope of blocking clock stop. TCU will have full control of the cluster and domain that will be tested.

5.8.3 Clock Stretch

5.8.3.1 Clock Stretch Requirements

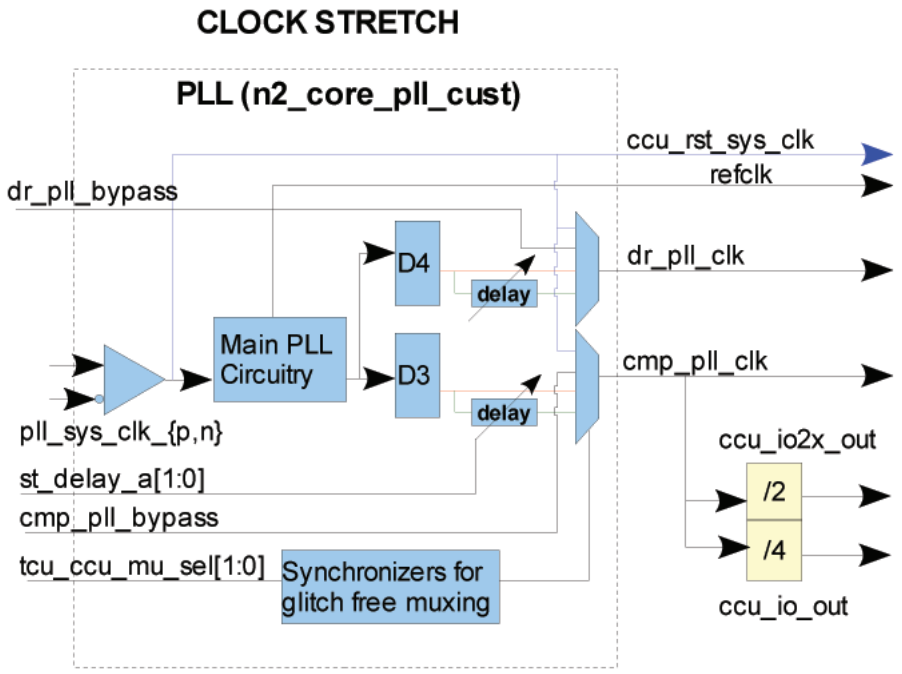
This section describes the clock stretch implementation on OpenSPARC T2 . For clock stretch designs on OpenSPARC T1 and Panther, refer to the “requirements” document. OpenSPARC T2 clock stretch operates within the following guidelines and requirements as defined by SPTE.

- Clock period stretch is needed only on the “cmp” domain.
- Frequency modulation due to stretch is restricted to 1 cycle.
- The amount of shift is in the interval (0, $T_{vco}/2$)
- This actual shift is implemented using an RC delay line with reasonable granularity.
- Assertion of clock stretch is controlled by test registers programmed through the JTAG interface. These registers also control the amount of shift in RC delay line.
- There is no latency requirement, measured from the time clock stretch is asserted to the time clock shift occurs.
- Core clocks can then be stopped and state element values can be shifted out via scan chains.

5.8.3.2 PLL Support for Pulse Stretching

Clock stretching capability is built into the PLL because of the analog RC delay line. The mechanism for shifting the positive or negative edge edge is simple. The VCO output is muxed with a delayed version of itself as shown in the simplified FIGURE 5-19.

FIGURE 5-19 Clock Stretching Capability in PLL



Ports on the CCU that are relevant to clock stretch are tcu_ccu_mux_sel[1:0]. It determines which leg of the mux in the PLL is selected for output. 2'b11will select the leg for clock stretch.

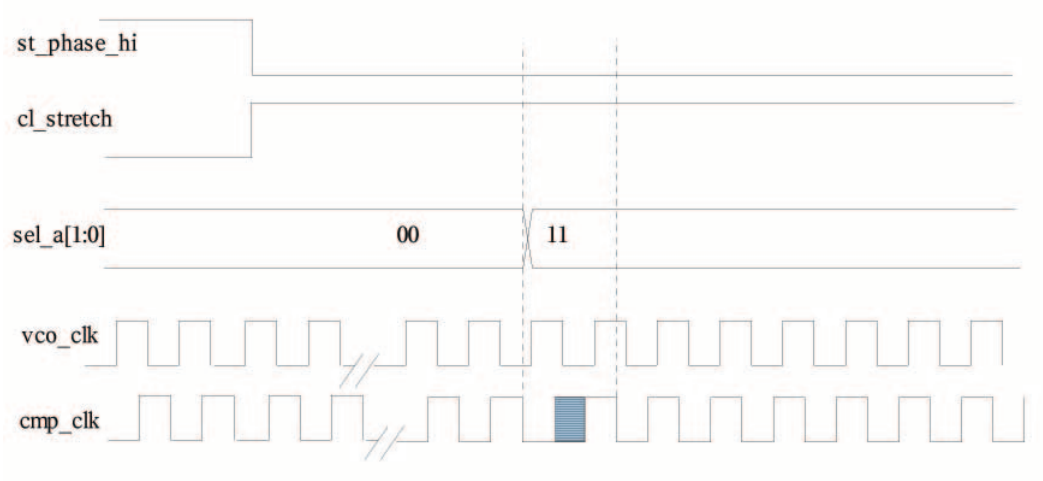
The amount of pulse shift (st_delay_a), and the phase to be stretched (st_phase_hi), are programmed in the CSR prior to the actual clock stretch event. The other 2 inputs to the CCU activate the stretch mux.

These signals can be asynchronous to the cmp_clk domain. They are synchronized appropriately in the PLL, depending on whether the high phase or the low phase is stretched. Stretching on the low phase (shifting the positive clock edge) requires synchronizing the mux selects to the negative edge of VCO clock. Conversely, stretching on the high phase requires changing the mux selects on the positive edge of VCO clock. This is illustrated in the next section.

5.8.3.3 Timing Diagrams

The following waveforms illustrate the operation of the pulse stretching circuitry for shift during the low phase (ie, rising edge).

FIGURE 5-20 Clock Stretch Timing Events



Note the synchronization of cl_stretch internally to generate the inputs to sel_a[1:0] on the falling edge of cmp_clk. This ensures the select lines to the mux change in the low phase.

The approach is similar for st_phase_hi = 1, the main difference being that sel_a[1:0] inputs are generated on the rising edge of cmp_clk.

5.8.3.4 Programmability

In PLL_REG, 5 bits can be programmed for the following clock stretch fields.

TABLE 5-11 Clock stretch fields in CSR block

CSR Field	Bits	Description
st_phase_hi	25	Stretches high phase if true, else stretches low phase
st_delay_cmp	27:26	cmp clock stretch delay settings [00, 01, 10, 11] => [40, 80, 120, 160] ps under nominal PVT
st_delay_dr	35:34	dr clock stretch delay settings [00, 01, 10, 11] => [40, 80, 120, 160] ps under nominal PVT

5.8.4 SerDes Deterministic Test Mode (DTM)

5.8.4.1 Basic Requirements

DTM is a strategy for running tests for SerDes in a repeatable, deterministic manner. It allows testers to sweep the SPARC core clock frequencies without breaking PLL lock, and perform traditonal functonal testing using the serial link interface.

In a nutshell, the tester goes through an initialization process to calibrate the RX lanes, and place data such that the outcomes on the blunt side are known and controllable. However, the TX data cannot be observed deterministically, so a workaround is to observe this TX data via the debug interface. The basic requirements are:

- All reference clocks to PLL inputs should come from the same source
- This applies to the core PLL, PSR, and FSR. ESR is excluded from DTM testing
- Convert clock domains from mission mode as follows:
  - IO -> DR
  - PC -> DR
  - CMP and DR domains unchanged
- Sync pulses between IO <-> CMP now are equivalent to DR <-> CMP
- Clock rates changed as follows
  - ref1 = ref2 = ~75-100 MHz
  - cmp = ~600-1500 MHz
  - dr = io = pc = ~75-100 MHz
  - cmp:dr ratio = 1:8, 1:11 or 1:15

5.8.4.2 Supported Clock Frequencies

The ideal scenario is to be able to perform a schmoo of OpenSPARC T2 across the entire operating range of core frequencies, ie, from 600 Mhz – 1.5 Ghz. However, because of PLL characteristics, no single divider setting will allow this, and a minimum of 3 *gear ratios* is needed.

A gear ratio corresponds to the CCU core divider configuration (in this scheme, affected only by one divider, D2). Link rate for serial links indicates the data transfer rate which may be equal to or fractional multiples of internal clock speeds.

5.8.4.3 Clocking Scheme

FIGURE 5-21 CCU PLL Configuration for DTM

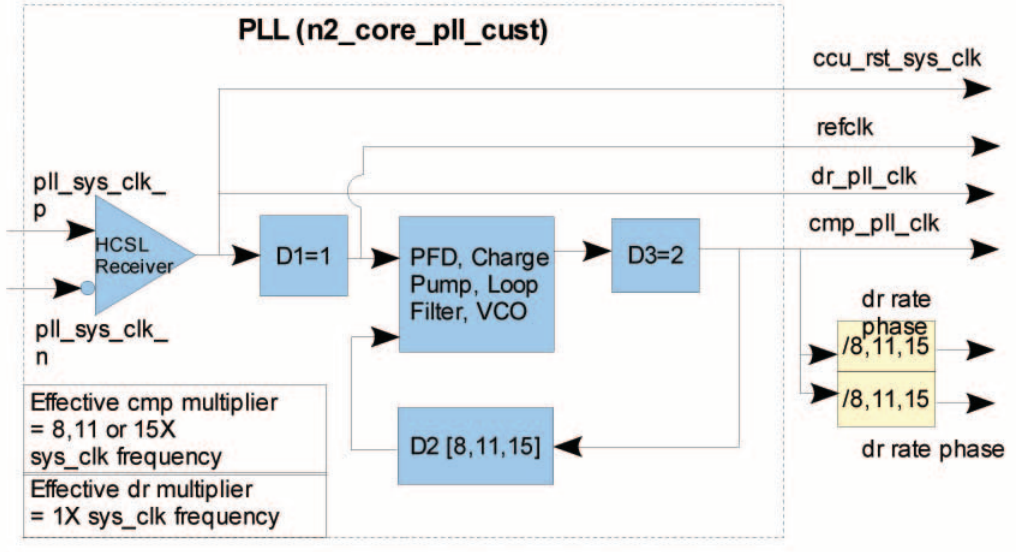


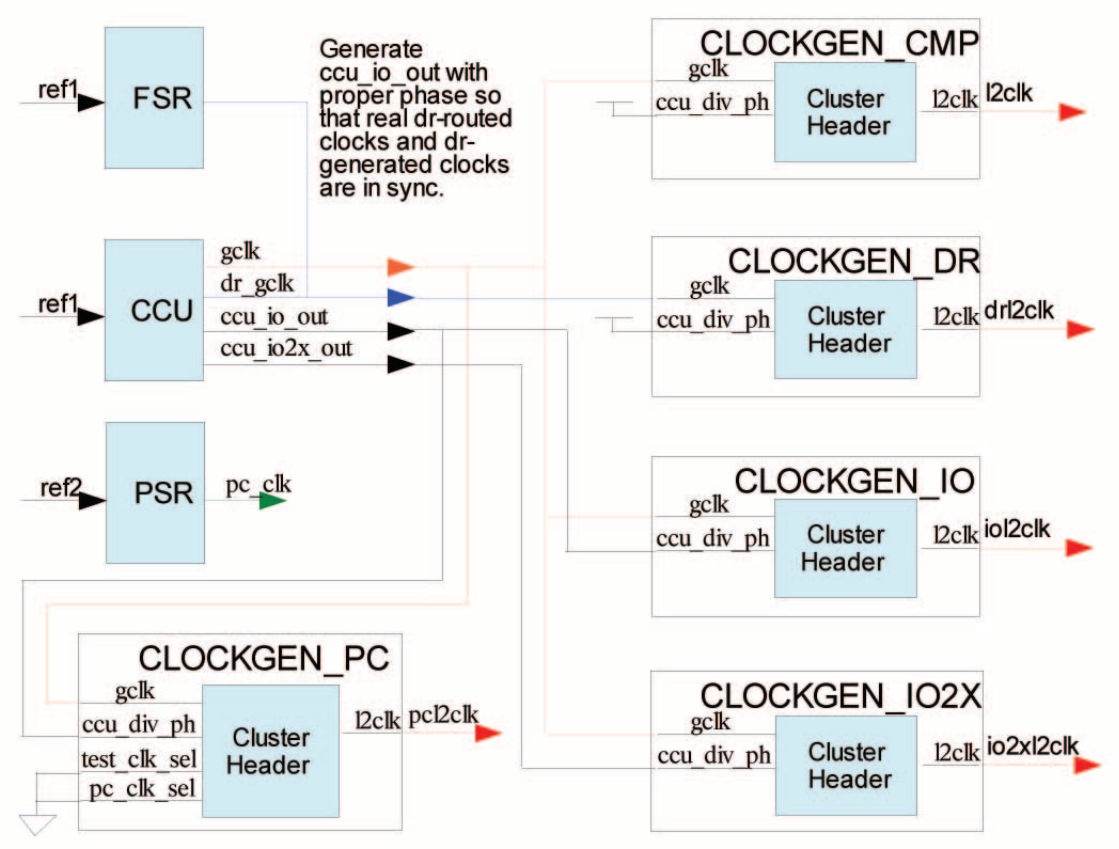
FIGURE 5-21 shows how the CCU PLL would be configured for DTM. Note that the muxes are not shown. They will configured such that the mux for the cmp clock output would be in functional mode whereas the mux for the dr clock would be in bypass mode.

As shown in FIGURE 5-21 and FIGURE 5-22, clusters that will get IO, IO2X, and PC clocks will operate at DR rate by dividing down from the cmp clock. This is different from the actual DR clock from the distribution tree. The former is dubbed **virtual dr clock** as opposed to the **real dr.** The ramificaton of this approach is simplification of cluster header muxing controls, easier gclk distribution, and more robust timing since there are no new timing paths for analysis. ***However, the real dr clock will have a 50-50 duty cycle, while the virtual dr clock will have a duty cycle of 50/50, 55/45 and 53/47 respectively for D2=8, 11, and 15.*** This is not expected to be an issue.

Even though PC, IO, IO2X and DR clocks operate at the same frequency albeit with perhaps different duty cycles during DTM, direct data crossing between these clocks needs to be handled with care due to high possibility of hold-time violations. In normal mode, there is no direct communication between the 4 domains, or is handled via asynchronous fifos, so these paths are false. The min-time issues encountered in DR<->IO crossings are addressed by using lock-up latches in the MCU, while for PC<->IO they are pre-emptively addressed by inverting the phase of PC clock (inside the cluster header) with respect to the IO clock in the PEU.

Sync pulse positions for domain transfers between CMP<->IO, CMP<->IO2X, and CMP<->DR are shown in FIGURE 5-23. They depend on the divider value D2. The sync pulse pairs, sys_cmp and cmp_sys are unchanged. However, all other sync pulses appear in different positions depending on the value of D2.

FIGURE 5-22 Chip Level DTM Clocking Scheme



5.8.4.4 Programmation and Sequencing

After power on reset, the PLL_REG has to be programmed to set either of the DTM bits to '1', and set the PLL divider values to match one of the 3 acceptable gear ratios. The "CHANGE" field in PLL_REG should also be set to indicate frequency will change. (Depending on whether DTM1 or DTM2 is selected, DBG and MIO mux controls will be affected. However, from the CCU's perspective, it is only one mode. The only check CCU will perform that if both DTM1 and 2 will are asserted, mode 1 will be considered active).

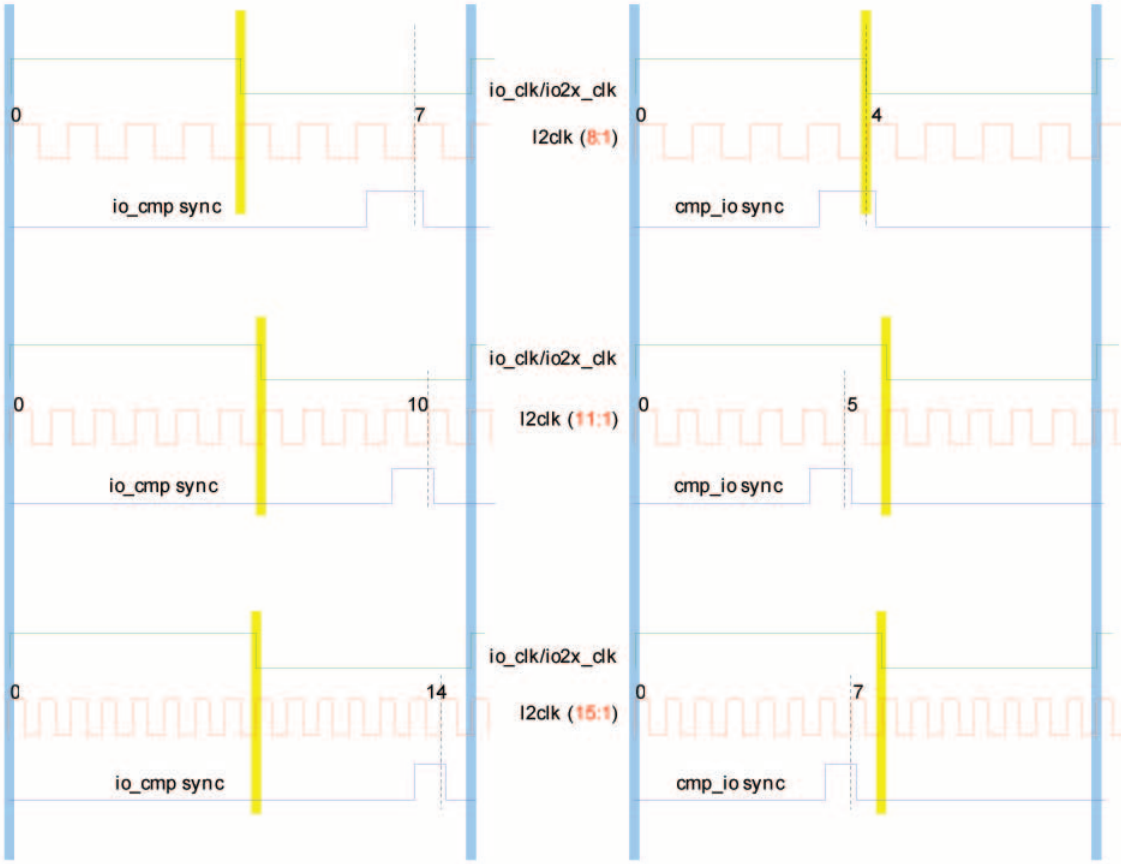
The RST block will see the "ccu_rst_change" signal asserted and issue a warm reset.

The PLL will load the divider values upon de-assertion of reset and begin to lock for DTM to provide new clock frequencies (cmp and dr).

During reset active, the cluster header will see a change on the ccu_serdes_dtm signal which will be be used to disable PC clock select. All other clockgen modules will require **no connectivity change** as a result of DTM; the process of generating a different clock frquency will be handled transparently by the CCU and cluster header.

Everything will function normally here on.

FIGURE 5-23 New Sync Pulse Positions for DTM **(All locations are at final destination after being flopped once in cluster)**



5.9 Appendix A.1 – Sync Pulse Design Procedure

This appendix focuses on the design methodology for sync pulses between CMP and DR domains which are non-integer multiples of one another. Consider the near ideal scenario for synchronizing between two such domains. We make the following assumptions:

- There is no jitter, skew, or PVT mismatches.
- At some point in time, both positive clock edges are perfectly aligned.

- There is zero phase offset between the PLL input reference and output.
- The setup and hold requirements on flip-flops are small[1].
- Propagation delay is experienced only by data (through wires, and clock-Q).

No matter which direction data is crossing domains, it makes sense to maximize the amount of time available between data launch and capture. This is illustrated in FIGURE 5-24 and FIGURE 5-25.

For cmp_clk to dr_clk transfers, the launch edge should be the first positive edge of cmp_clk ***after*** a dr_clk sampling point. The enable control generation would then occur a cmp cycle before launch.

On the other hand, for dr_clk to cmp_clk synchronization, the capture edge would be the last cmp_clk rising edge ***prior*** to a dr_clk sampling event. This time, the enable control would assert a cmp cycle before capture.

The cycle repeats when both the launch and capture clocks are perfectly aligned on the rising edge, or every *M* cycles of slow clock (equivalently *N* cycles of fast clock).

TABLE 5-12 Waveform Parameters for Ideal Case

Parameter	Description
N	Multiplication factor of fast clock
M	Multiplication factor of slow clock
T	Fast clock period (cmp_clk)
Tref	Reference clock period = *N.T*
Tslow	Slow clock period (dr_clk) = *M.T*
k	Cycle count of slow clock starting with 0

1.

FIGURE 5-24 Synchronization from Fast to Slow Clock

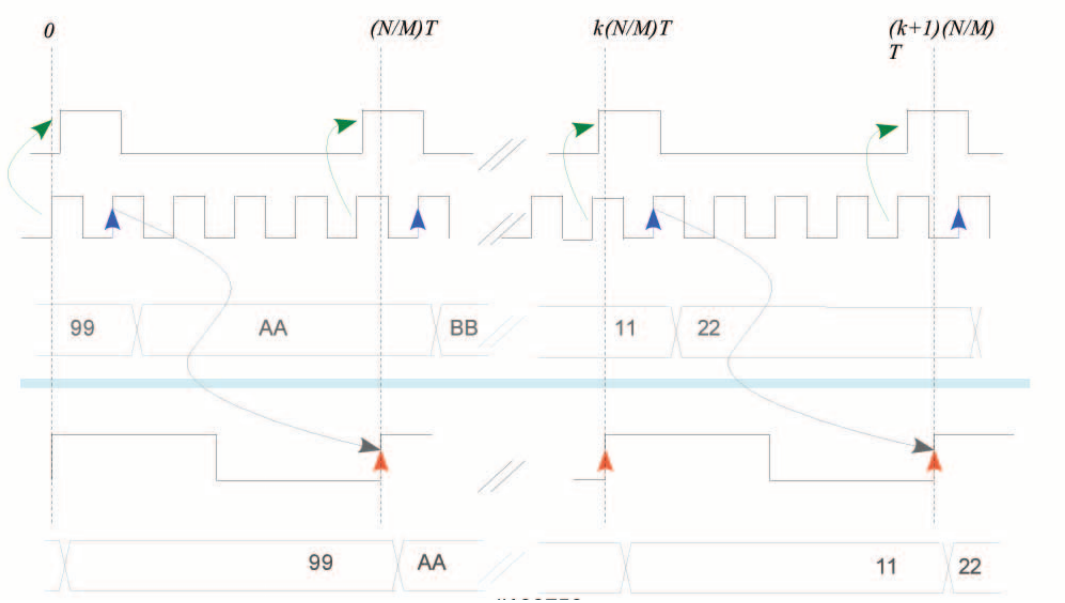
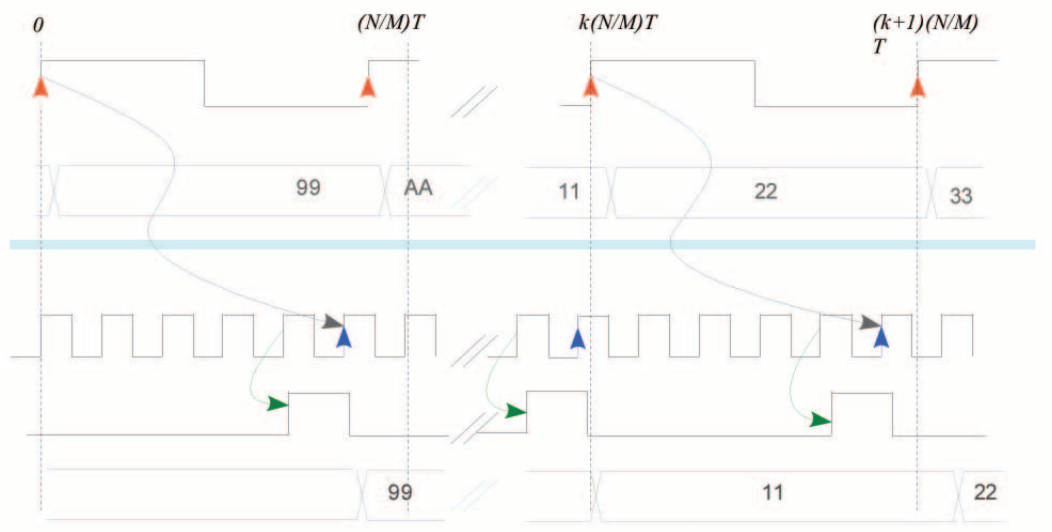


FIGURE 5-25 Synchronization from Slow to Fast Clock



5.10 Appendix A.2 – Sync Pulse Timing Analysis

Looking at the idealized timing diagrams and parameters in Appendix A.1, we can analyze the conditions one at a time for *min* and *max* conditions.

5.10.1 Fast to Slow Clock Synchronization

As per the proposal, the capture edge for cycle *k* is at *(k+1)(N/M)T.* Working our way backwards, the corresponding launch edge is the first edge on cmp_clk after *k(N/M)T*. This works out to be *FLOOR(k(N/M)+1)T.* The pulse would have to be generated a cycle before at *FLOOR(k(N/M)T.*

Amount of time available for setup,

tmax = (k+1)(N/M)T – FLOOR(k(N/M)+1)T

Similarly, the launch edge lags the last capture edge by

tmin = FLOOR(k(N/M)+1)T – k(N/M)T

There is a subtle difference between *FLOOR(k(N/M)+1)T* and *CEILING(k(N/M)T)* which shows up when k= 0.

5.10.2 Slow to Fast Clock Synchronization

The situation is reversed, where we need to launch data on the dr_clk as early as possible, ie, at *k(N/M)T*, and capture on cmp_clk as late as possible, at *FLOOR((k+1)(N/M)T).* Enable pulse generation thus occurs at *FLOOR((k+1)(N/M)-1)T.*

Therefore, setup margin is given by:

tmax = FLOOR((k+1)(N/M)T) – k(N/M)T

And the lag from last capture is

tmin = k(N/M)T – FLOOR(k(N/M))T

5.10.3 Modifications for Non-Ideal Scenario

This time, we revisit the approach while factoring in real conditions. Some other parameters that need to be considered are:

TABLE 5-13 Additional Parameters for Non-ideal Scenario

Parameter	Description
tsu	Setup time of capture flip-flop.
th	Hold time of launch flop.
tcq?*	Clock-to-Q time in flip flop.
tdata	Data delay from launch flop's Q output to capture flop D-pin
tskew	Skew and static phase offsets between slow and fast clocks.
tjitter	Jitter (cycle to cycle and long-term) between the clocks.

*

The constraints for max and min timing under non-ideal conditions for data launch and capture to work correctly are:

```
tmax >  tcq + tsu + tdata + tskew + tjitter
tmin > th + tskew + tjitter - tcq
```

Corresponding timing margins are given by:

```
tmargin,max = tmax – tdata – tcq – tsu – tskew - tjitter
tmargin,min = tmin – th – tskew – tjitter + tcq
```

Both sets of equations hold true, regardless of synchronization direction. Only the parameters *tmax* and *tmin* are derived differently as in the past section.

5.10.4 Computation and Selection of Sync Pulses

Now that a scheme has been proposed, and sync pulse generation formalized, here is the algorithm for the complete solution:

1. Compute which phase of slow clock the pulses should be generated (under ideal conditions) for fast to slow clock.
2. Find the corresponding timing margins available *tmax* and *tmin* (also ideal).
3. Estimate the amount of *skew, jitter, tdata, tcq, tsu* and *th*, and calculate tmargin,*max* and *tmargin,min*.

4. If *tmargin,max < 0 OR tmargin,min < 0* for any sync pulse, adjust phase for that pulse and repeat steps 2 through 4.
5. Repeat steps 1 through 4 for all ratios of *N/M.*
6. Repeat steps 1 through 5 for slow to fast clock.
7. Repeat steps 1 through 6 for all 3 refclk frequencies.

We are done when *tmargin,max > 0 AND tmargin,min > 0* for every ratio, otherwise for any particular ratio **if either *tmargin < 0,* this scheme will not work**.

CHAPTER 6

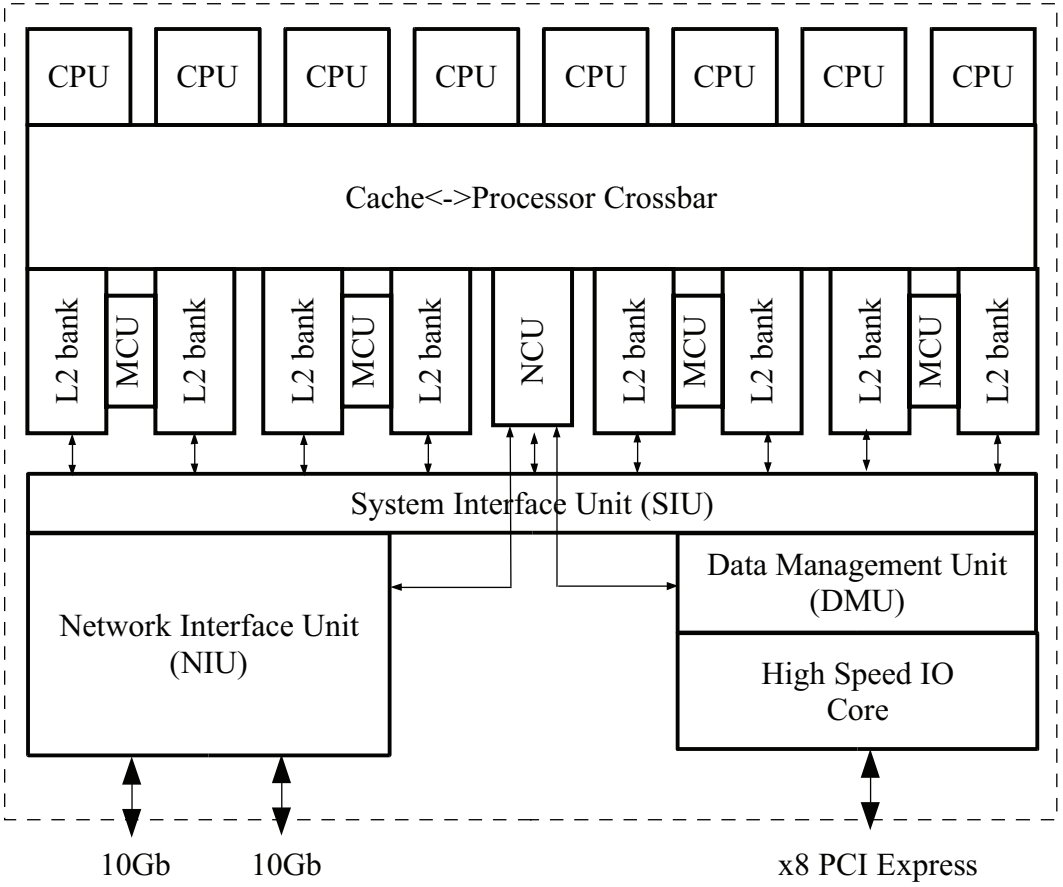
System Interface Unit (SIU)

This chapter contains the following sections:

6.1 Overview

OpenSPARC T2 has on chip multiple system I/O subsystems. OpenSPARC T2 integrates Fire's high speed IO core and connect directly to a x8 PCI Express channel (2GB/s/direction). OpenSPARC T2's integrated network I/O unit includes two 10Gb Ethernet MACs (2.5GB/s/direction). The System Interface Unit (SIU) provides 12GB/s of raw bandwidth per direction and has flexible interfaces for the Network Interface Unit (NIU) and Data Management Unit (DMU) to access memory via 8 secondary level cache (L2) banks. SIU supports Fire's PCI Express. For the NIU, SIU was architected with the ability to allow write traffic to bypass other posted write traffic. SIU does not support coherency.

FIGURE 6-1 SIU Top Level Block Diagram



The SIU also provides a data return path for reads to the Peripheral I/O subsystems. The data for these PIO Reads and interrupt messages generated by the PCI Express subsystem are ordered in the SIU prior to delivery to the NonCacheable Unit (NCU).

6.2 Terminology

Cacheable: Can be stored in the L2 cache.

Cacheline: 64Byte

CSR: Configuration Status Register. A storage element for holding status or configuration information. They can exist either on OpenSPARC T2. or off OpenSPARC T2..

Core clock domain : reference to the speed of the sparc processor core and L2 cache. Target is 1.4GHz

DMA : Direct Memory Access. A load or store originating from the IO subsystem that targets the memory subsystem.

Inbound : Logically toward the CPU and memory subsystem and NCU, away from the IO subsystems (NIU or DMU).

IO clock domain : references to the speed of the internal I/O interfaces units. Either 1/3 or core clock frequency. Also refered to as System clock domain

JBUS : Jalapeno bus. A coherency bus used in Niagara1 and other Sun processors.

Nonposted : A transaction in which the sender does want and require an acknowledge of delivery. A read is always nonposted.

Outbound : Logically toward the IO subsystems (NIU or DMU) and away from the CPU, L2 caches and the NCU

Packet : A structure for transfering information between interfaces. A Packet can consists of just a header or a header followed by a payload.

PIO : Peripherial Input Output. A load or store originating from the cpu that does not target the caches and memory. The target of a PIO can either be onchip or offchip.

Posted : A transaction in which the sender does not want nor require an acknowledgement of delivery

RDD : Read and Discard – All DMA accesses are noncoherent. Thus the data for a DMA read should be treated as use once and discard. All DMA reads are converted into RDD's by L2 caches and do not allocate in the L2 cache.

WR8 : Write 8 bytes – A WRM is decomposed into up to 8 WR8 when the WRM is forwarded to an L2 bank. The L2 does a read modify write operation for each 8 byte store. The 8 byte enables for a WR8 can be randomly on or off.

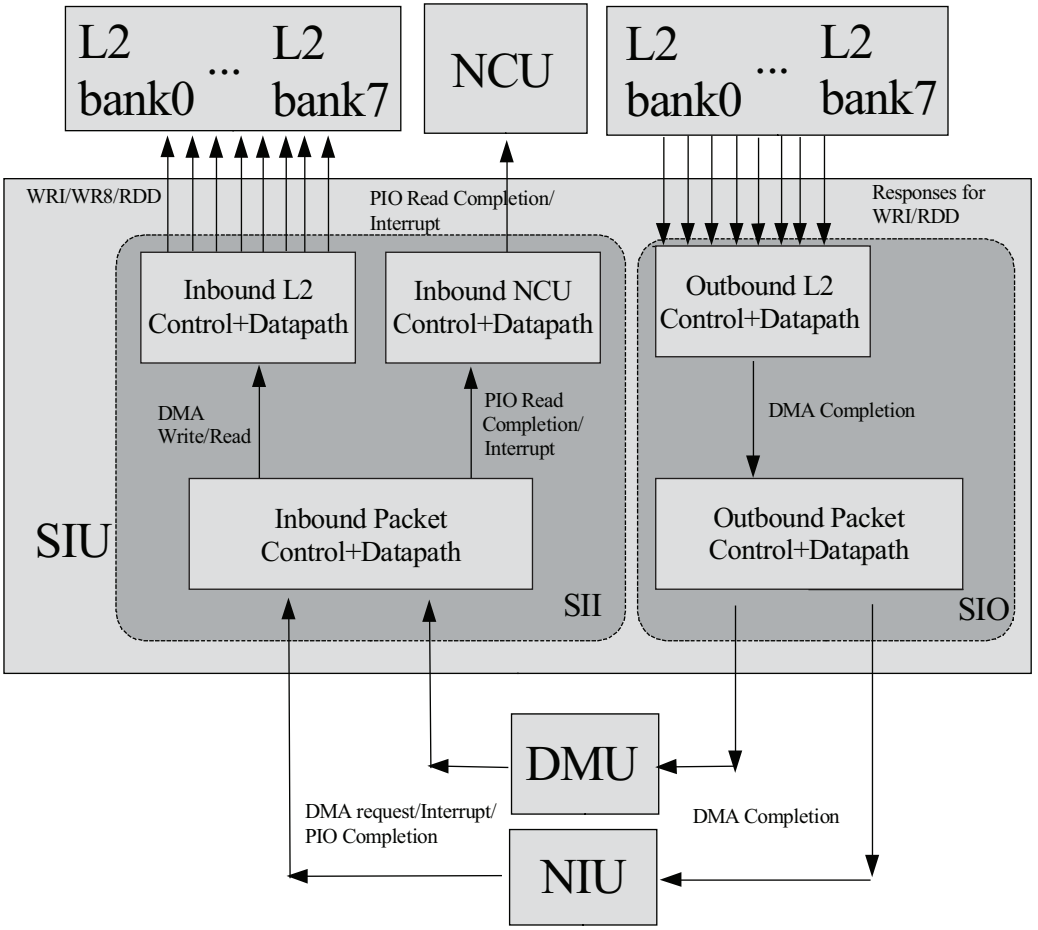
WRI: Write Invalidate – all DMA Writes that are aligned to a cacheline address boundary and writes 64 Byte of data will invalidate any matching address in the L2 cache tag array prior to data being forwarded to memory. (Terminology comes from the JBUS architecture)

WRM: Write Merge – a DMA Write with 1 or more bytes of the 64 byte payload not enabled. (Terminology comes from the JBUS architecture). A WRM may not cross cacheline boundary.

6.3 SIU Top Level Logical Block Diagram

The SIU is partitioned physically and logically into 2 parts based on flow direction – SIU Inbound (SII) for inbound traffic and SIU Outbound (SIO) for outbound traffic.

FIGURE 6-2 SIU Logical Block Diagram



All inbound traffic continues inbound through SIU until it reaches NCU or an L2 bank. All outbound traffic from NCU or L2 must leave SIU in the outbound direction. NCU and L2 banks cannot send traffic to each other through the SIU.

DMU and NIU cannot send traffic toward each other through the SIU. Because the L2 banks have their own paths through the memory controllers to memory, the SIU sees each L2 bank as a slave device. SIU assumes L2 never initiates requests to SIU. Likewise, Network blocks are always seen as master devices pulling from and pushing data to L2 only.

SIU does not support coherency.

All traffic uses a packet transfer interface. Each packet is 1 or 2 consecutive address/header cycles immediately followed by 0 or more consecutive data/payload cycles. SIU follows L2's addressing convention : big endian where the databytes for the lowest address are transferred first. Where applicable, byte enables are positional where byte_enable[0] always refer to databits[7:0] for all interfaces.

The interfaces between SIU and L2 are in the core clock domain - 1.5GHz. The interfaces between SIU and DMU, NIU, NCU are in the IO clock domain – 350 MHz or 1/4 core clock frequency.

TABLE 6-1 shows the packet types from DMU and NIU that are supported by SIU:

TABLE 6-1 Supported Packet Types from NIU and DMU

Source	Packet type	Posted	Queue in which packet may enter
NIU	RDD	Nonposted	Ordered
			Bypass
	WRI	Posted	Ordered
			Bypass
	WRI	Nonposted	Ordered
			Bypass

TABLE 6-1 Supported Packet Types from NIU and DMU *(Continued)*

DMU	RDD	Nonposted	DMA/INT (Ordered)
	WRI	Posted	DMA/INT (Ordered)
	WRM	Posted	DMA/INT (Ordered)
	INT (Mondo)	Nonposted	DMA/INT (Ordered)
	PIO Rd Completions	Posted	
			PIO (Bypass)

TABLE 6-2 shows how the mapping between inbound DMA addresses from DMU and NIU to the L2 bank number. This is to support partial L2 banks when entire set(s) of L2 banks are disabled.

5 bits (PM, BA01, BA23, BA45, BA67) are used to indicate partial mode active, and which of the 4 pairs of banks are available. X is a don't care. When PM is on, it is illegal for only 3 of the BAs to be asserted and illegal if all 4 BAs are deasserted.

TABLE 6-2 Partial L2 Bank Mapping

PM	BA67	BA45	BA23	BA01		L2Bank[2]	L2Bank[1]	L2Bank[0]
0	X	X	X	X		PA[8]	PA[7]	PA[6]
1	0	0	0	0	Illegal	0	0	PA[6]
1	0	0	0	1		0	0	PA[6]
1	0	0	1	0		0	1	PA[6]
1	0	0	1	1		0	PA[7]	PA[6]
1	0	1	0	0		1	0	PA[6]
1	0	1	0	1		PA[7]	0	PA[6]
1	0	1	1	0		PA[7]	~PA[7]	PA[6]
1	0	1	1	1	Illegal	0	PA[7]	PA[6]
1	1	0	0	0		1	1	PA[6]
1	1	0	0	1		PA[7]	PA[7]	PA[6]
1	1	0	1	0		PA[7]	1	PA[6]

TABLE 6-2 Partial L2 Bank Mapping *(Continued)*

1	1	0	1	1	Illegal	0	PA[7]	PA[6]
1	1	1	0	0		1	PA[7]	PA[6]
1	1	1	0	1	Illegal	1	PA[7]	PA[6]
1	1	1	1	0	Illegal	1	PA[7]	PA[6]
1	1	1	1	1		PA[8]	PA[7]	PA[6]

SIU along with the gasket block inside each of the sparc cores implement the same index hashing algorithm for the L2 cache. The purpose is to improve performance for certain software application – to reduce the thrashing of certain L2 indexes. Software can enable this feature by writing to a CSR in NCU.

If hashing is enabled and PA[39]==0, SIU converts the PA[39:0] to a different PA for L2 :

L2_PA[39:18] = PA[39:18];

L2_PA[17:13] = PA[32:28] ^ PA[17:13];

L2_PA[12:11] = PA[19:18] ^ PA[12:11];

L2_PA[10:0] = PA[10:0];

6.4 Logical Subblocks

The SIU is partitioned physically and logically into 2 parts based on flow direction – SIU Inbound (SII) for inbound traffic and SIU Outbound (SIO) for outbound traffic. SIU is also partioned into 2 clock domains.

A SIU subunit's name consists of 3 - 5 characters. All subunits are listed below:

SII subunits:

ILC0, ILC1, ILC2, ILC3, ILC4, ILC5, ILC6, ILC7,

ILD0, ILD1, ILD2, ILD3, ILD4, ILD5, ILD6, ILD7,

IPCC, IPCS0, IPCS1, IPD, INC, IND,

SIO subunits:

OLC0, OLC1, OLC2, OLC3, OLC4, OLC5, OLC6, OLC7,

OLD0, OLD1, OLD2, OLD3, OLD4, OLD5, OLD6, OLD7,

OPCC, OPCS0, OPCS1, OPDS, OPDC

The first letter of a logical subunit's name indicates direction: **I**nbound or **O**utbound

The second letter of a logical subunit's name indicates either the destination for inbound data or the source object for outbound data:

L = L2 cache

N = NCU

P = Packets (DMU & NIU)

The third letter of a logical subunit's name indicates **C**ontrol path or **D**atapath

The optional last 1 or 2 character represents either an instance number (i.e. L2 bank number) or the subunit's clock domain (***C***ore or ***S***ystem).

6.4.1 Clocks

Target operating frequencies, CPU:IO = 1500MHz:350MHz

Supported operating frequencies:

4:1 CPU:IO synchronous clock ratio

L2 and NCU @ 1500MHz,1400MHz,1300MHz, 1200MHz

NIU and DMU @ 350MHz, 325MHz, 300MHz

6.4.2 Interface Datapath Access Mechanism

TABLE 6-3 Interface Datapath Access Mechanism

Datapath	Mechanism	Comment
SII to L2	Credits	SIU initially given : 2 request credits, 4 64B-write-invalidate-data credits
From L2 to SIO	None	SII must not send request to L2 if SIO will not have space to receive response from L2
SII to NCU	Request/Grant Arbitration	Receiver of packets schedules resources and asserts Grant for 1 cycle to winning Requestor. Requestor must send packet the cycle after its Grant is asserted. Winning requestor may reassert Request for subsequent packet while delivering current packet. Requestor's Request must stay asserted until its Grant is received. SII can buffer 16 PIO Read data returns SII can buffer 4 interrupt mondo data or 4 x 16B
From DMU to SII, From NIU to SII	Credits	SII provides each IO subsystem 2 dedicated inbound packet queues. Each inbound queue can hold a maximum of 16 requests + 64 Byte data payload per request. SII notifies each IO subsystem when an request has been dequeued from either of the 2 dedicated inbound packet queue.
From SIO to DMU, From SIO to NIU	None	DMU or NIU must not send request to SII if it will not have space to receive response.
Internal SIU Datapaths	Request/Grant Arbitration	Receiver of packets schedules resources and asserts Grant for 1 cycle to winning Requestor. Requestor must send packet the cycle after its Grant is asserted. Winning requestor may reassert Request for subsequent packet while delivering current packet. Requestor's Request must stay asserted until its Grant is received.

6.4.3 Inbound

The parity protected interfaces between SIU and the DMU and NIU are 128 bit wides with side band signals for packet control. Having a 128 bit for header allows SIU to provide a rich set of transaction types and allows SIU to provide a uniform

and generic but flexible enough for most IO architectures.

The inbound packet interface protocol works as follows : (replace 'ext' with 'niu' or 'dmu')

Cycle 1: Header Cycle

- ext_sii_hdr_vld asserts for 1 cycle to indicate ext is sending the packet header.
- ext_sii_reqbypass indicates that ext is sending this packet to the SIU's 'bypass' inbound queue.
- ext_sii_datareq indicates this packet has a payload following the header cycle.
- ext_sii_datareq16 indicates this packet has only 1 cycle (16 Byte maximum)of payload. If datareq is asserted and datareq16 is deasserted, this packet has 4 cycles of payload for a maximum transfer size of 64Bytes.
- ext_sii_data[127:0] contains a valid header.

Cycle 2-5: Payload Cycle(s)

- ext_sii_hdr_vld is deasserted.
- ext_sii_data[127:0] contains the payload data
- ext_sii_parity[3:0] contains the parity for each 32 bit of data. Parity[N]= xor(data[32N+31 : 32N])
- ext_sii_be[15:0] contains the byteenables for each byte of data if applicable. BE[N]==1 implies write data[8N + 7 : 8N]

Each IO subsystem must keep track of number of available entries in the SIU Inbound queues and not overflow the SIU. Each time SIU Inbound forwards a request from its packet queue to its inbound L2 or NCU queue, SIU Inbound returns a credit back to the appropriate IO subsystem via sideband signals.

Each of the SIU Inbound queue allows for a maximum of 16 packets.

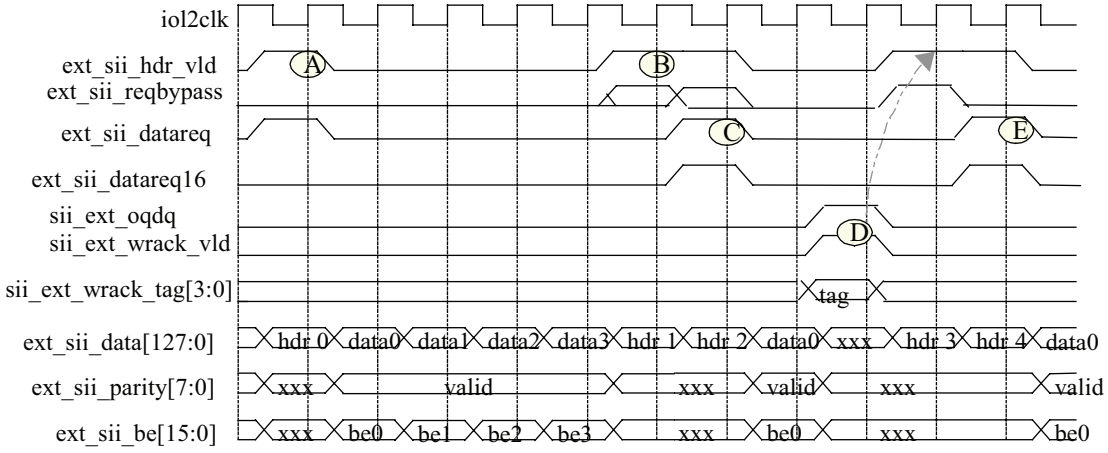
SIU supports back-to-back packet transfers with no dead cycle in between packets.

SIU architecturally supports PIO Read returns, DMA read requests, Interrupt Writes, DMA Write full cacheline (posted and nonposted), DMA write merge 64 bytes (posted only). The same logic is instantiated twice. One for each IO subsystem interface.

Another type of DMA access come from TCU/JTAG interface. The Read/Write access from the JTAG interface is 8 bytes. The address need to be 8-bytes alinged (addr[2:0] = 3'b000). There is only 1 outstanding request allowed from the JTAG interface. The signals sending from JTAG interface should be running on cmp clock domain. When SII sending TCU read request, the adddr[2] need to be zero, so that L2 will return most critical 8-bytes first.

6.4.4 Interface Timing Diagrams and Protocols

FIGURE 6-3 Inbound Packet Interface Timing Diagram



A: *A 64-byte DMA write request with 4 cycles of data payload.*
If from Fire-DMU, this must be destined for the Ordered Queue in SIU.
If from NIU, ext may choose which queue in SIU is appropriate for the behavior wanted.
This example shows the DMA write request is for the Ordered Queue.

B: *A read request, no payload.*
If from Fire-DMU, this must be destined for the Ordered Queue in SIU.
If from NIU, ext may choose which queue in SIU is appropriate for the behavior wanted.
If this was the 16th outstanding credit, ext must stop issuing transactions to this Queue.

C: *A 16-byte PIO read data return; 1 cycle of data payload following the header.*
If from NIU, this must be destined for the Ordered Queue in SIU.
If from Fire-DMU, this must be destined for the Bypass Queue in SIU.
If this was the 16th outstanding credit, ext must stop issuing transactions to this Queue.

D: *SIU returns a credit after forwarding a DMA write data from the Ordered Queue to the Inbound L2$ Queue.*
If write request was from Fire-DMU, sii_dmu_wrack_vld asserts with tag information and the Fire-DMU can add this credit back to the credit list,
If write request was from NIU and the write request was in the Ordered Queue, sii_ext_oqdq asserts.
Ext may now resume sending transaction to the SIU – this example has a DMA read request following credit.

E: *(Only applicable for Fire-DMU) INT header plus 1 data beat of data payload, the SIU checks the header to distinquish PIO read completion from INT payload;*

6.4.4.1 From NIU to SIU

Single and back-to-back DMA read request from NIU to SIU

For each DMA read request, NIU must always guarantee it has buffer space to receive the DMA read response that will return from SIU outbound..

A DMA read does not allocate in the cache.

This describes the protocol for a single DMA read request from NIU to SIU.

1. NIU first checks that it has a credit available for the packet transfer.

   If NIU does not have credit for SIU's Ordered Queue and wishes to send a DMA read request to SIU's Ordered Queue, NIU must wait for sii_niu_oqdq to assert.

   If NIU does not have credit for SIU's Bypass Queue and wishes to send a DMA read request to SIU's Bypass Queue, NIU must wait for sii_niu_bqdq to assert.

   Once NIU has guaranteed that it will not overflow SIU, NIU can send the DMA read request packet on the interface.

2. Send packet. This transfer takes one IO clock cycle.

   a. NIU asserts header valid signal (niu_sii_hdr_vld) high,

   b. NIU drives all the header bits appropriately on niu_sii_data[127:0]

      Note that SIU does not require the byte address field in the header to be aligned to a cacheline boundary. Memory will always return the critical 32 bit word first and wrapped back to the beginning cacheline address boundary. Because the current software ethernet driver model has the ethernet transmit/control information structures in memory aligned to 64B address boundary, NIU sets DMA Read address[5:0] set to 0.

   c. NIU drives data request signals (niu_sii_datareq and niu_sii_datareq16) low.

   d. NIU drives the destination queue signal (niu_sii_reqbypass) to high for the bypass queue or low for the ordered queue.

   e. The parity lines (niu_sii_parity) are a don't care for the header cycle.

   f. SIU does not support byte enables for reads. If byteenable wires exist at at the top level interface (niu_sii_be), then they are a don't care for the header cycle..

3. NIU must reduce the appropriate credit counter.

SIU supports back-to-back transfers from NIU. A 2^{nd} packet may be sent immediately the cycle after the 1^{st} DMA Read packet if there is credit available for the 2^{nd} transfer.

Single and back-to-back DMA write request from NIU to SIU

For each DMA write request that NIU requires an acknowledgement of completion, NIU must always guarantee it has buffer space to receive the DMA write completion response that will return from SIU outbound. For Neptune's NIU, this is always true. A DMA write packet that needs an completion ack returned must be marked nonposted in the packet header.

A DMA Write does not allocate in the cache.

This describes the protocol for a single DMA write request from NIU to SIU.

1. NIU first checks that it has a credit available for the packet transfer.

   If NIU does not have credit for SIU's Ordered Queue and wishes to send a DMA write request to SIU's Ordered Queue, NIU must wait for sii_niu_oqdq to assert.

   If NIU does not have credit for SIU's Bypass Queue and wishes to send a DMA write request to SIU's Bypass Queue, NIU must wait for sii_niu_bqdq to assert.

   Once NIU has guaranteed that it will not overflow SIU, NIU can send the DMA write request packet on the interface.

2. Send packet. This transfer takes 5 IO clock cycle.

   a. On the first cycle,

      i. NIU asserts header valid signal (niu_sii_hdr_vld) high.

      ii. NIU drives all the header bits appropriately on niu_sii_data[127:0].

      Note that SIU does not require the byte address field in the header to be aligned to a cacheline boundary and allows for byte mask field in the header to be set for a WRM, because the current software ethernet driver model has the ethernet receive/control information structures in memory aligned to 64B address boundary, NIU sets DMA Write address[5:0] to 0, and command field to be write full 64 bytes, byte mask active to 0.

      iii. NIU drives data request signals (niu_sii_datareq high and niu_sii_datareq16 low).

      iv. NIU drives the destination queue signal (niu_sii_reqbypass) to high for the bypass queue or low for the ordered queue.

      Writes to the ordered queue will always be be issued by the SIU to L2 after the youngest write in the bypass queue and after all prior writes to L2 has been sent from L2 to MCU. The writes in the bypass queues are not ordered with respect to other writes.

      v. The parity lines (niu_sii_parity) are don't cares for the header cycle.

      vi. If byte enables wires exist at the top level interface, the byteenable lines (niu_sii_be) are don't cares for the header cycle.

b. On the 2nd, 3rd, 4th, and 5th cycle,

i. NIU drives header valid signal (niu_sii_hdr_vld) low.

ii. NIU drives data payload on niu_sii_data[127:0] in big endian format.

iii. The data request signals (niu_sii_datareq and niu_sii_datareq16) are don't cares for nonheader cycles.

iv. The destination queue signal (niu_sii_reqbypass) is a don't care for nonheader cycle.

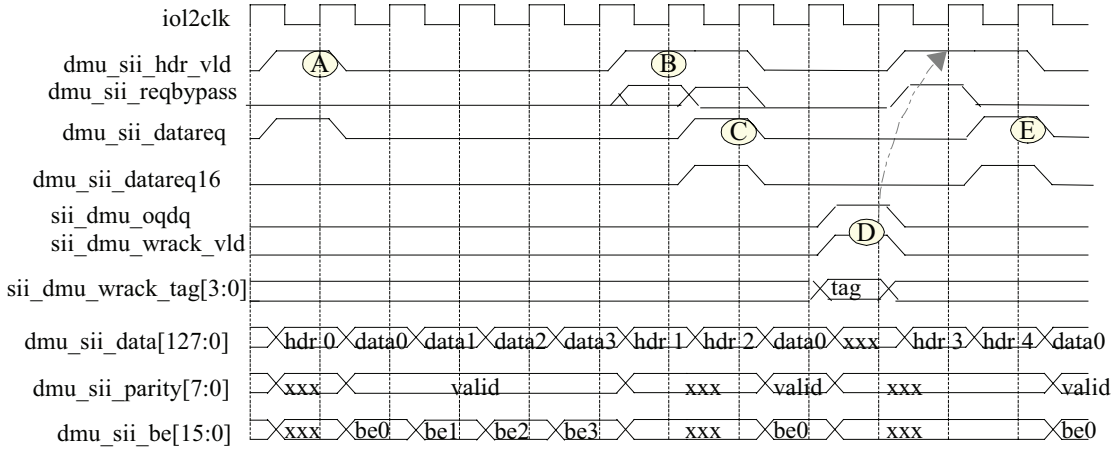
v. The parity lines (niu_sii_parity[3:0]) are driven.

vi. If byte enables wires exist at the top level interface, the byteenable lines (niu_sii_be[15:0]) should be all 1's to be safe but are treated as don't cares if during the header cycle, the byte mask active field was 0.

3. NIU must reduce the appropriate credit counter.

SIU supports back-to-back transfers from NIU. A 2nd packet may be sent immediately the cycle after the 1st DMA Write packet if there is credit available for the 2nd transfer.

FIGURE 6-4 Timing Diagram for SIU Inbound Packet from DMU



***A**: A 64-byte DMA write request with 4 cycles of data payload.*
If from Fire-DMU, this must be destined for the Ordered Queue in SIU.
If from HT-DMU, dmu may choose which queue in SIU is appropriate for the behavior wanted.
This example shows the DMA write request is for the Ordered Queue.

***B**: A read request, no payload.*
If from Fire-DMU, this must be destined for the Ordered Queue in SIU.
If from HT-DMU, dmu may choose which queue in SIU is appropriate for the behavior wanted.
If this was the 16th outstanding credit, dmu must stop issuing transactions to this Queue.

***C**: A 16-byte PIO read data return; 1 cycle of data payload following the header.*
If from HT-DMU, this must be destined for the Ordered Queue in SIU.
If from Fire-DMU, this must be destined for the Bypass Queue in SIU.
If this was the 16th outstanding credit, dmu must stop issuing transactions to this Queue.

***D**: SIU returns a credit after forwarding a DMA write data from the Ordered Queue to the Inbound L2$ Queue.*
If write request was from Fire-DMU, sii_dmu_wrack_vld asserts with tag information and the Fire-DMU can add this credit back to the credit list,
If write request was from HT-DMU and the write request was in the Ordered Queue, sii_ext_oqdq asserts.
DMU may now resume sending transaction to the SIU – this example has a DMA read request following credit.

***E:** INT header plus 1 data beat of data payload, the SIU checks the header to distinquish PIO read completion from INT payload;*

6.4.4.2 From a Fire-PCI Express-DMU to SIU

For SIU to support Fire's version of a DMU that connects to PCI Express, SIU Inbound must adapt to a stricter form of credit management where not only is a credit returned when SII dequeues a packet, but the corresponding id for the packet that was dequeued.

With respect to packet types, Fire does not support nonposted DMA writes and does not allow DMA Writes to pass other writes so all DMA's would effectively go into the SIU's Ordered Queue. The packet format differences between a prior DMU<->SIU and Fire-DMU<->JBC is handled in a thin layer within the new DMU called the DSN.

Fire's PCI-Express DMU requires SIU to be able to accept without flow control all completions for all PIO reads that had originated from NCU.

SIU's Inbound architecture has a 16 deep FIFO for its Ordered Queue and a 16 deep FIFO for its Bypass Queue. Fire's PCI-Express DMU supports 16 'credits' of DMAs+Interrupts and 16 credits of PIOs.

A DMA write credit id may be reused once the write has been posted (dequeued) from SIU's Inbound Packet Ordered Queue. A DMA read credit id may only be reused after the DMA read data response has returned from SIU's Outbound to DMU. Interrupt (Mondo type only) credit id may only be reused after NCU has acked or nacked the Interrupt. NCU must adapt to Fire's PCI-Express DMU and manage 16 PIO credits.

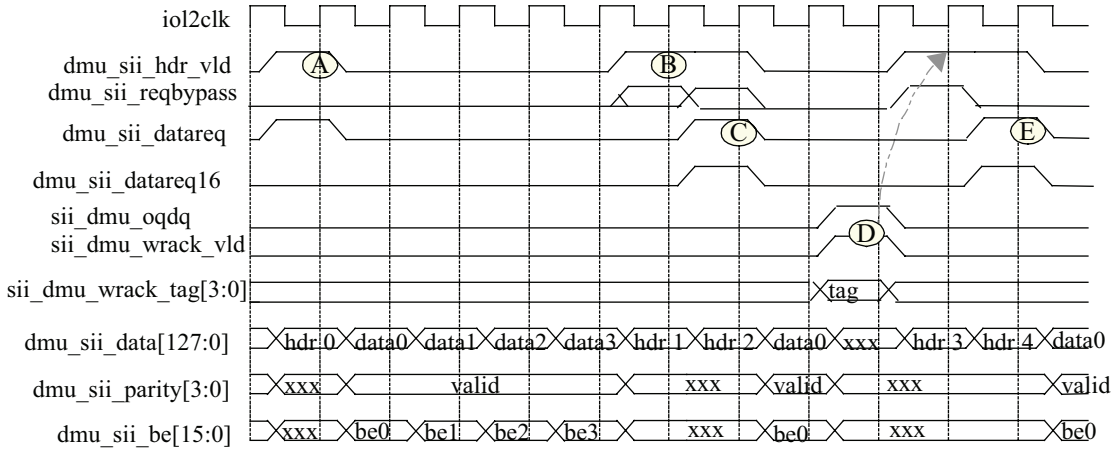
Without a design change, SIU's ordered Queue can support 16 outstanding DMA+Interrupt. But expanding that to an 32 deep ordered queue and gutting the design of the bypass queue to accommodate the 16 PIO completions would significantly impact schedule and not allow for code reuse. The estimated net area savings from gutting the design and extending the queue depth was determined to be small (at most couple hundred square microns in Epic9).

The solution proposed and implemented is conditioned on the fact that SIU already has dependency pointers for each of the 16 entries in the Inbound Packet Bypass Queue and dependency pointers for each of the 16 entries in the Inbound Packet Ordered Queue. An internal mode wire is added to indicates the interface is connected to PCI-Express DMU instead of NIU. Effectively, the 'Bypass Queue' becomes an ordered PIO completion queue. When in PCIExpress mode, rather than using the results of address cams to set up dependencies for a new packet entering the Bypass Queue, SIU forces that new packet to wait for the youngest packet existing in the Ordered Queue. Note that new packets entering the Ordered Queue by default (even for NIU) depends on the youngest packet existing in the Bypass Queue. Forcing this dependency when in PCIExpress mode converts the Bypass Queue into another ordered queue IF ALL the packets in the Bypass Queues drains to the same place. If there are multiple DMA Writes in the bypass queue, the existing ordering mechanism used for NIU would not guarantee a younger DMA Write from the bypass queue entry would complete later than an older DMA Write from the bypass queue that targets a different L2 bank.

Therefore, ALL DMA Writes (and interrupts) from Fire's PCI-Express DMU must be steered into the Ordered Queue. Likewise, because there is only 1 drain from SIU for all PIO completions (one FIFO path from SIU to NCU), when ALL PIO Completions from Fire's PCI-Express DMU are steered in the Bypass Queue, the ordering

requirement that PIO completions must pull all prior DMAs writes and all prior PIO completions is satisfied and SIU now achieved a 16 entry PIO completion queue at the cost of a few muxes added the original design. If physical areas become more critical, the cam logic remaining may be optimized out during physical implementation.

FIGURE 6-5 Timing Diagram for SIU Inbound Packet from DMU



***A**: A 64-byte DMA write request with 4 cycles of data payload.*
If from Fire-DMU, this must be destined for the Ordered Queue in SIU.
If from HT-DMU, dmu may choose which queue in SIU is appropriate for the behavior wanted.
This example shows the DMA write request is for the Ordered Queue.

***B**: A read request, no payload.*
If from Fire-DMU, this must be destined for the Ordered Queue in SIU.
If from HT-DMU, dmu may choose which queue in SIU is appropriate for the behavior wanted.
If this was the 16th outstanding credit, dmu must stop issuing transactions to this Queue.

***C**: A 16-byte PIO read data return; 1 cycle of data payload following the header.*
If from HT-DMU, this must be destined for the Ordered Queue in SIU.
If from Fire-DMU, this must be destined for the Bypass Queue in SIU.
If this was the 16th outstanding credit, dmu must stop issuing transactions to this Queue.

***D**: SIU returns a credit after forwarding a DMA write data from the Ordered Queue to the Inbound L2$ Queue.*
If write request was from Fire-DMU, sii_dmu_wrack_vld asserts with tag information and the Fire-DMU can add this credit back to the credit list,
If write request was from HT-DMU and the write request was in the Ordered Queue, sii_ext_oqdq asserts.
DMU may now resume sending transaction to the SIU – this example has a DMA read request following credit.

***E:** INT header plus 1 data beat of data payload, the SIU checks the header to distinquish PIO read completion from INT payload;*

Single and Back-to-Back DMA Read Request from Fire-DMU to SIU

For each DMA read request, DMU must always guarantee it has buffer space to receive the DMA read response that will return from SIU outbound. For Fire-DMU, this is always true.

A DMA read does not allocate in the cache.

This describes the protocol for a single DMA read request from Fire-DMU to SIU.

1. Fire-DMU first checks that it has a credit available for the packet transfer.
    a. If Fire-DMU does not have credit for SIU's Ordered Queue and wishes to send a DMA read request to SIU, Fire-DMU must wait for sii_dmu_wrack_vld to assert, a prior DMA Read to complete from SIU Outbound or an interrupt credit to return from NCU.
    b. Once Fire-DMU has guaranteed that it has a DMA credit, Fire-DMU can send the DMA read request packet on the interface.
2. Send packet. This transfer takes one IO clock cycle.
    a. Fire-DMU asserts header valid signal (dmu_sii_hdr_vld) high.
    b. Fire-DMU drives all the header bits appropriately on dmu_sii_data[127:0]
        i. Fire-DMU always generate a 64 Byte aligned address and never cross a cacheline boundary. Fire-DMU sets DMA Read address[5:0] set to 0.
    c. Fire-DMU drives data request signals (dmu_sii_datareq and dmu_sii_datareq16) low.
    d. Fire-DMU drives the destination queue signal (dmu_sii_reqbypass) to low for the ordered queue.
    e. The parity lines (dmu_sii_parity) are a don't care for the header cycle.
    f. The byteenable lines (dmu_sii_be) are a don't care for the header cycle.
3. Fire-DMU must reduce the DMA credit counter.

SIU supports back-to-back transfers from Fire-DMU. A 2^{nd} packet may be sent immediately the cycle after the 1^{st} DMA Read packet if there is credit available for the 2^{nd} transfer.

Single and Back-to-Back DMA Write Request from Fire-DMU to SIU

For Fire-DMU, all DMA Writes are posted and address aligned to cacheline boundary although byte(s) may be deasserted at the beginning or the end. this is always true.

A DMA Write does not allocate in the cache.

This describes the protocol for a single DMA write request from Fire-DMU to SIU.

1. Fire-DMU first checks that it has a credit available for the packet transfer.

a. If Fire-DMU does not have credit for SIU's Ordered Queue and wishes to send a DMA write request to SIU, Fire-DMU must wait for sii_dmu_wrack_vld to assert, a prior DMA Read to complete from SIU Outbound or an interrupt credit to return from NCU.

b. Once Fire-DMU has guaranteed that it has a write credit, Fire-DMU can send the DMA write request packet on the interface.

2. Send packet. This transfer takes 5 IO clock cycle.

   a. On the first cycle,

      i. Fire-DMU asserts header valid signal (dmu_sii_hdr_vld) high.

      ii. Fire-DMU drives all the header bits appropriately on dmu_sii_data[127:0].

      For Fire-DMU, DMA Writes with or without bytemask active has address aligned to 64-Byte boundary. Fire-DMU set DMA Write address[5:0] to 0. Fire-DMU does not support nonposted writes.

      iii. Fire-DMU drives data request signals (dmu_sii_datareq high and dmu_sii_datareq16 low).

      iv. Fire-DMU drives the destination queue signal (dmu_sii_reqbypass) to low for the ordered queue.

      Writes to the ordered queue will always be be issued by the SIU to L2 after the youngest PIO completion in the bypass queue and after all prior writes to L2 has been sent from L2 to MCU.

      v. The parity lines (dmu_sii_parity) are don't cares for the header cycle.

      vi. The byteenable lines (dmu_sii_be) are don't cares for the header cycle.

   b. On the 2^{nd}, 3^{rd}, 4^{th}, and 5^{th} cycle,

      i. Fire-DMU drives header valid signal (dmu_sii_hdr_vld) low.

      ii. Fire-DMU drives data payload on dmu_sii_data[127:0] in big endian format.

      iii. The data request signals (dmu_sii_datareq and dmu_sii_datareq16) are don't cares for nonheader cycles.

      iv. The destination queue signal (dmu_sii_reqbypass) is a don't care for nonheader cycle.

      v. The parity lines (dmu_sii_parity[3:0]) are driven.

      vi. The byteenable lines (dmu_sii_be[15:0]) should be driven. They are treated as don't cares if during the header cycle, the byte mask active field was 0.

3. Fire-DMU must reduce the DMA credit counter.

SIU supports back-to-back transfers from Fire-DMU. A 2nd packet may be sent immediately the cycle after the 1st DMA Write packet if there is credit available for the 2nd transfer.

Single and Back-to-Back Interrupt Request from Fire-DMU to SIU

PCI-Express requires that an interrupt it send to the CPU via whatever path is delivered after the youngest corresponding DMA write has been sent to memory (interrupt from Fire-DMU must guarantee that prior writes sent to SIU has left L2). Fire-DMU take advantage of the ordering maintained by SIU by simply send an interrupt to NCU via SIU's ordered queue.

Because these interrupt types are mondo, Fire-DMU must be capable of retrying NAcked mondo and Fire-DMU must have an interrupt response path from NCU.

This describes the protocol for a single interrupt from Fire-DMU to SIU.

1. Fire-DMU first checks that it has a credit available for the packet transfer.
   a. If Fire-DMU does not have credit for SIU's Ordered Queue and wishes to send a DMA write request to SIU, Fire-DMU must wait for sii_dmu_wrack_vld to assert, a prior DMA Read to complete from SIU Outbound or an interrupt credit to return from NCU.
   b. Once Fire-DMU has guaranteed that it has a DMA-INT credit, Fire-DMU can send the interrupt request packet on the interface.
2. Send packet. This transfer takes 2 IO clock cycle.
   a. On the first cycle,
      i. Fire-DMU asserts header valid signal (dmu_sii_hdr_vld) high.
      ii. Fire-DMU drives all the header bits appropriately on dmu_sii_data[127:0].
      iii. Fire-DMU drives data request signals (dmu_sii_datareq high and dmu_sii_datareq16 high).
      iv. Fire-DMU drives the destination queue signal (dmu_sii_reqbypass) to low for the ordered queue.

          Interrupt in the ordered queue will always be be issued by the SIU to NCU after the youngest PIO completion in the bypass queue and after all prior writes to L2 has been sent from L2 to MCU.
      v. The parity lines (dmu_sii_parity) are don't cares for the header cycle.
      vi. The byteenable lines (dmu_sii_be) are don't cares for the header cycle.

b. On the 2nd cycle,

i. Fire-DMU drives header valid signal (dmu_sii_hdr_vld) low.

ii. Fire-DMU drives mondo data payload on dmu_sii_data[127:0].

This is the 1st 16Bytes of the mondo data.

iii. The data request signals (dmu_sii_datareq and dmu_sii_datareq16) are don't cares for nonheader cycles.

iv. The destination queue signal (dmu_sii_reqbypass) is a don't care for nonheader cycle.

v. The parity lines (dmu_sii_parity[3:0]) are driven.

vi. The byteenable lines (dmu_sii_be[15:0]) should be all 1's to be safe.

3. Fire-DMU must reduce the DMA-INT credit counter.

SIU supports back-to-back transfers from Fire-DMU. A 2nd packet may be sent immediately the cycle after the 1st Interrupt packet if there is credit available for the 2nd transfer.

Single and Back-to-Back PIO Read Data Return from Fire-DMU to SIU

PCI-Express requires that an interrupt it send to the CPU via whatever path is delivered after the youngest corresponding DMA write has been sent to memory (interrupt from Fire-DMU must guarantee that prior writes sent to SIU has left L2). Fire-DMU take advantage of the ordering maintained by SIU by when in PCI-Express mode by simply sending the PIO Completion to NCU via SIU's Bypass Queue.

This describes the protocol for a single PIO Read data return from Fire-DMU to SIU.

1. Send packet. This transfer takes 2 IO clock cycle. Because NCU guarantees that NCU will stop sending PIO request if all 16 PIO credits are used, Fire-DMU does not need to check for credit available prior to PIO read completion transfers.

a. On the first cycle,

i. Fire-DMU asserts header valid signal (dmu_sii_hdr_vld) high.

ii. Fire-DMU drives all the header bits appropriately on dmu_sii_data[127:0].

iii. Fire-DMU drives data request signals (dmu_sii_datareq high and dmu_sii_datareq16 high).

iv. Fire-DMU drives the destination queue signal (dmu_sii_reqbypass) to high for the bypass queue.

A PIO completion in the bypass queue will always be be issued by the SIU to NCU after the youngest PIO completion in the bypass queue and after all prior DMA writes to L2 has been sent from L2 to MCU and all prior interrupts are on its way to NCU.

v. The parity lines (dmu_sii_parity) are don't cares for the header cycle.

vi. The byteenable lines (dmu_sii_be) are don't cares for the header cycle.

2. On the 2nd cycle,

a. Fire-DMU drives header valid signal (dmu_sii_hdr_vld) low.

b. Fire-DMU drives data payload on dmu_sii_data[127:0] in big endian format.

i. The CPU knows how many bytes it wanted.

ii. Note that if this PIO read is for a CSR, Fire-DMU must replicate the CSR's 64 bit value on the 128 bit payload. The CPU expects the lower 64 data bits to be the same as the upper 64 bits.

iii. The data request signals (dmu_sii_datareq and dmu_sii_datareq16) are don't cares for nonheader cycles.

iv. The destination queue signal (dmu_sii_reqbypass) is a don't care for nonheader cycle.

v. The parity lines (dmu_sii_parity[3:0]) are driven.

vi. The byteenable lines (dmu_sii_be[15:0]) should be all 1's.

SIU supports back-to-back transfers from Fire-DMU. A 2nd packet may be sent immediately the cycle after the 1st PIO read completion packet if there is credit available for the 2nd transfer.

6.4.4.3 From SIU to L2

Back to Back Read Requests to L2

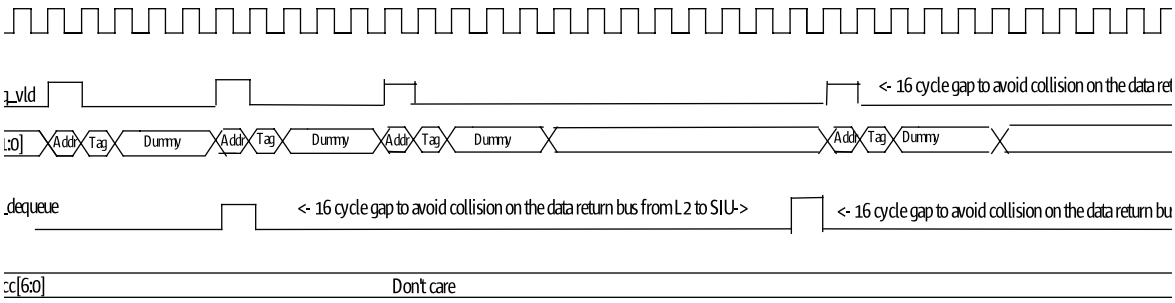
A read request packet (on sii_l2t_req[31:0]) consists of 2 header cycles followed by 3 dummy cycles. The first 2 dummy cycles are required by L2 for pipeline alignment - so the initial pipeline stages of a read request looks like the stages of a write 8 byte request. The last dummy cycle is for turnaround required by L2. The signal sii_l2t_req_vld asserts for 1 cycle to indicate the 1st cycle of the packet transfer. SIU only has 2 request tokens so SIU can burst only up to 2 back to back read requests to the L2 Tag. After 2 outstanding requests without an indication of a dequeue from L2

(l2t_sii_iq_dequeue asserting high for 1 cycle), SIU must wait for an entry in the input queue in L2Tag to drain. FIGURE 6-6 shows the best case of back to back read requests. SIU bursts 2 reads, then sees l2t_sii_iq_dequeue asserting during the 2nd transfer and proceeds to send out a 3rd read request. After which point, resource constraints prevent further back to back requests. According to the L2 pipeline, the earliest l2t_sii_iq_dequeue asserts is 2 cycles after L2Tag receives the 2nd dummy data cycle from the 1st read request. The next possible assertion of l2t_sii_iq_dequeue must be a minimum of 16 clock cycles after the 1st assertion. This minimum pulse period of once every 17 cycles avoids the bus contention on the data return path from L2 Bank to SIU (1 header cycle + 16 payload cycles).

A read request inbound to L2 will generate a response packet from L2 on the outbound path. An inbound read request from SIU should not overflow SIU's receive header and data buffers in the outbound direction. SIU's outbound L2 subunit sends dequeue signals to the inbound L2 subunit to communicate buffer resource availability and SIU's inbound L2 subunit increments and decrements its credit counters.

For a read request, there is no data payload to protect, so the ECC lines (sii_l2b_ecc) are a don't care.

FIGURE 6-6 SIU to L2 : Back to Back Reads



Because the L2 Tags are physically way across the chip from SIU or a one way distance of 9 to 10mm, 2 cycles of delay staging flops per direction will most likely be required to accomodate the paths between SIU and L2. The timing diagrams shown in this specification do not push out the signals to account for the delay stages.

Back to Back WR8 Request follow by WRI Request

A Write 8 byte is a partial store in L2 cache and the packet transfer consists of 2 cycles of header followed by 2 cycles of payload and 1 dummy turnaround cycle. The bytemask is encoded in the header. In the current L2 implementation, a WR8

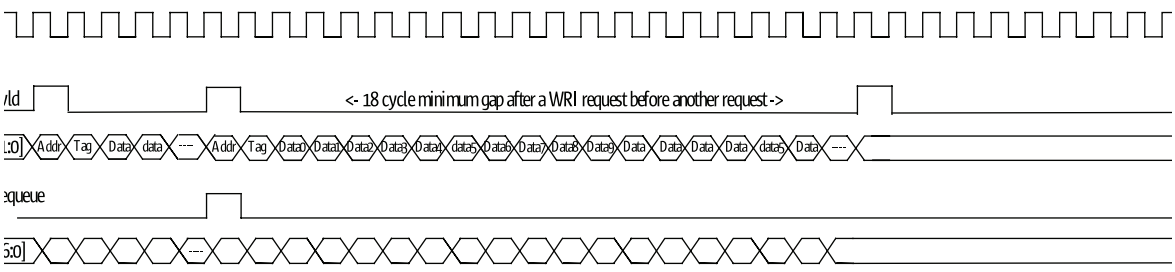
request does not consume any data I/O write buffer entry. Instead, L2 pumps the 64 bit data into pipeline stages of 64 flops. Like the read request to L2, SIU must have a request token available before it can send a write request. L2 asserts l2t_sii_iq_dequeue for 1 cycle when it sends the WR8 down its pipeline during the first pass. Note that for a WR8, SIU does not need a data token and should not decrement its data credit for a WR8 transfer. For performance purpose, SIU does not issue any WR8 with all bytes off.

A Write Invalidate (WRI) invalidates L2 if there's a tag match and moves all 64 bytes to memory. A WRI packet transfer consists of 2 cycles of header followed by 16 cycles of payload and 1 dummy turnaround cycle. L2 does not move the write data to the data cache array (for either WR8 or WRI) until L2 has accumulated the entire data payload. Like the read request to L2, SIU must have a request token available before it can send a write invalidate request. SIU will also need a data token (initially set to 4 to match the 4 I/O write buffer entries in L2 Tag). SIU decrement its data credit for a WRI transfer. Although not shown in FIGURE 6-7, L2 assserts l2t_sii_iq_dequeue for 1 cycle when it sends the WRI request down its pipeline and L2 also asserts l2t_sii_wib_dequeue for 1 cycle when L2 moves the 64 byte write data out of the I/O write buffer.

ECC (sii_l2b_ecc[6:0]) is generated to protect the content of each data cycle. The ECC algorithm used by SIU is the same as used by L2 for its data array and produces 7 check bits for a set of 32 data bit. Note that because ECC algorithm used by L2 is different from memory, L2 will check the ECC from SIU and will regenerate new ECC for memory for a WRI request.

Any nonposted write request inbound to L2 will generate an ack packet from L2 on the outbound path. An inbound write request from SIU should not overflow SIU's receive header buffers in the outbound direction. SIU's outbound L2 subunit sends dequeue signals to the inbound L2 subunit to communicate buffer resource availability and SIU's inbound L2 subunit increments and decrements its credit counters.

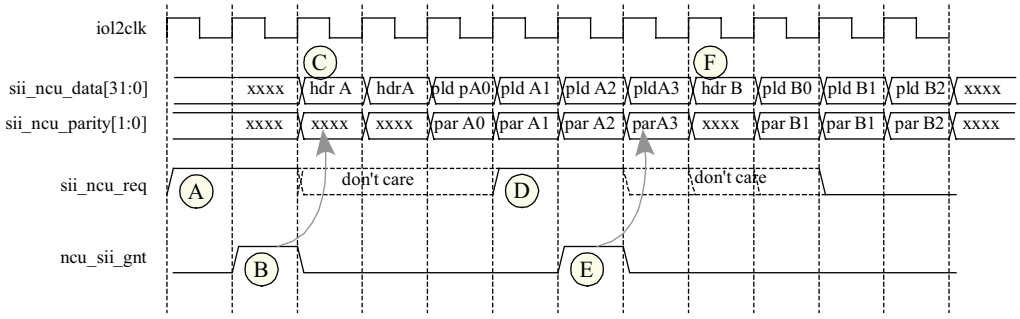
FIGURE 6-7 SIU to L2 : Back to Back Writes (WR8 followed by WRI)



6.4.4.4 From SIU to NCU

FIGURE 6-8 shows fastest possible back to back transfer from the SIU to NCU. This could be for Interrupt or PIO completion. SIU signals NCU it wants to transfer a packet. The request signal is held high until SIU sees ncu_sii_gnt asserts (it only asserts for 1 clock cycle). The cycle after grant, SIU drives for 5 cycles always, beginning with the header and 4 cycles of payload/parity. While SIU is driving the data lines, SIU may reassert the request line if it has more work to do. NCU is expected to ignore the request line until 2 cycles before the current packet to check if another transfer is requested. If NCU has space, it will reassert grant for 1 cycle again for the next transfer. The earliest the 2^{nd} grant can assert is on the 4^{th} payload cycle of the 1^{st} transfer. This would allow for back to back transfer with no bubbles on the sii_ncu_data bus.

FIGURE 6-8 Timing Diagram for Packet from SIU to NCU (Back to Back Transfer)



Ⓐ : *SIU signals NCU for transaction request;*
Ⓑ : *NCU sends grant signal to indicate ready to accept packet in next cycle;*
Ⓒ : *SIU starts new packet;*

Ⓓ : *2 cycles before finishing the current packet, NCU checks if there is another request;*
Ⓔ : *NCU sends grant signal to indicate ready to accept new packet in next cycle;*
Ⓕ : *SIU starts new packet..*

6.4.4.5 From TCU to SIU

There will be 2 bits interface (tcu_sii_vld, tcu_sii_data) from TCU to SII for DMA read/write access.

tcu_sii_data is a 128/64 bit data stream with 64-bit header, 64-bits of data in case of write. tcu_sii_vld asserted at the 1^{st}, 64^{nd} cycle on valid tcu_sii_data. There will be 128 bits (header+data) for DMA write and 64-bits for DMA read.

Header format :

bit[63:56] = 0x81 for read , 0x82 for write

bit[55:40] = 0x00 reserved

bit[39:0] = 8 byte aligned physical address (bit[5:0])

6.4.5 SIU's Inbound Pipeline

6.4.5.1 Major Pipeline Stages

There are 4 major pipeline stages in the inbound transfer. The best case total latency is 8 cmp clock cycles + 4 IO clock cycles. The worst case total latency to NCU is 15 cmp clock cycles + 15 IO clock cycles. The worst case total latency to L2 Tag is 15 cmp clock cycles +32 cmp clock cycles. The following subsections will discuss the the latencies of each stages in details. Please refer to FIGURE 6-9.

Stage1 : Interface (3-7 IO clock cycles)

The interface latency between DMU/NIU to SIU is between from 3 to 7 cycles. Request and grant arbitration costs 2 cycles. Although that latency can be hidden when there are back to back requests and the first request transfers at least 1 payload cycle, it must be taken into account. The best case is 1 cycle of read request to either L2 or NCU. The worst case is 1 cache line write request from DMU/NIU to L2. It is 1 cycle of header plus 4 cycles of payload. The transaction on the bus will be registered and written to the fifo(register file) in this stage.

Stage2 : Write to Fifo (1 to 4 cmp clock + 1 IO clock cycles)

This stage includes the header decoding and address lookup to set dependency for DMU packets followed by write to the fifo (register file). Once the last cycle of packet has been written into the FIFO and to disallow flow through FIFO, read pointer synchronization across the clock domain takes a minimum of 1 to 4 cmp clock cycles. 1 IO cycle (3 or 4 cmp clock cycles) for header decoding and register file write, and 1 to 4 cmp clocks for the read ptr synchronization.

Stage3: Read from Fifo and arbitration (3-11 cmp clock cycles)

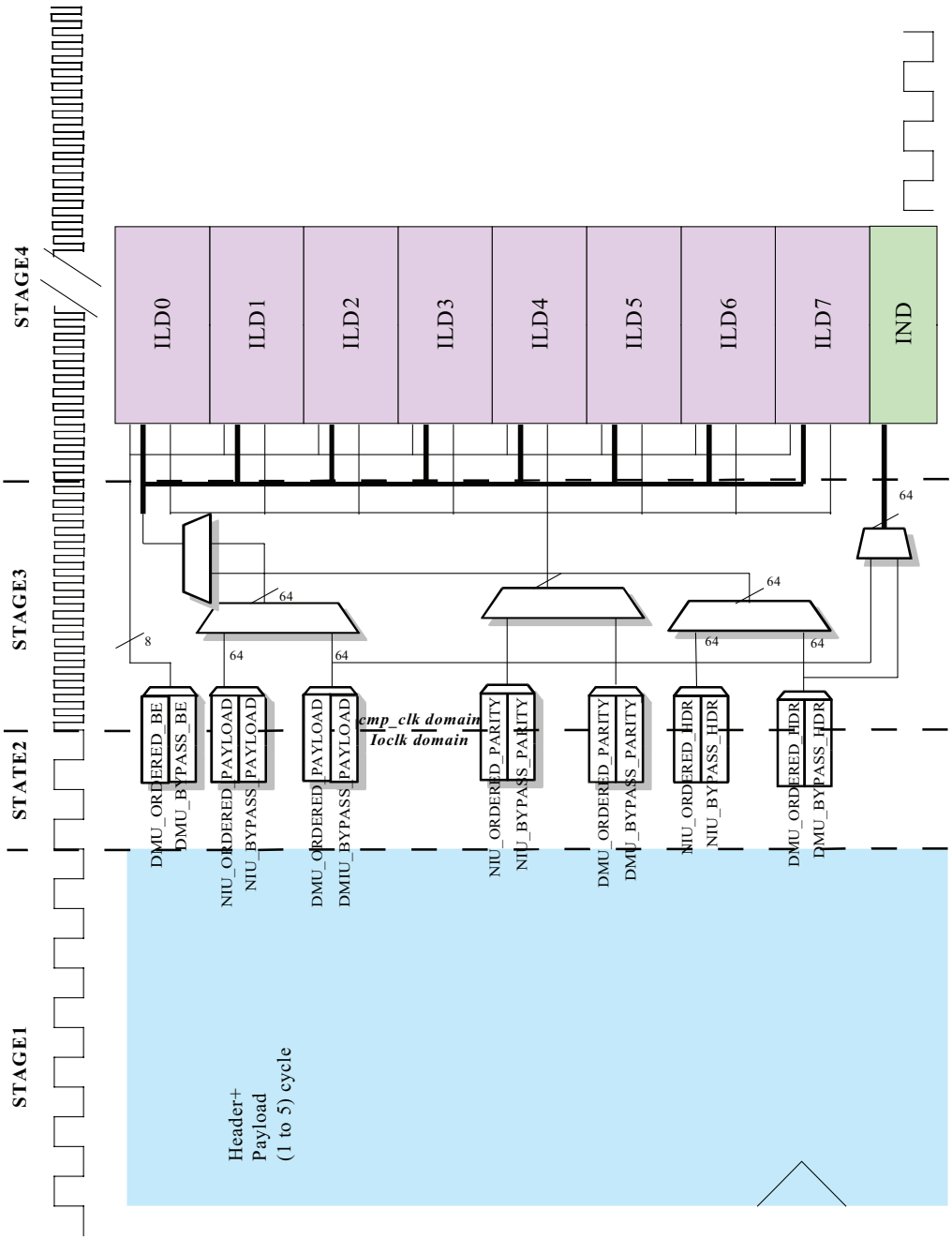
There is 2 cycles for arbitration between different fifo queues (DMU ordered, DMU bypass, NIU). In this stage, the arbitor check for the resources availabity and priorty of each queue to grant the transfer. Depending on the type of transaction, the transfer going to the queues in the inbound L2 subunits may take 1 or 9 cmp cycles. The transfer going to the queues in the inbound NCU subunit takes 1, 2 or 3 cmp cycles. Inbound toward L2 contributes to both the best case and worse case latency.

Stage4: L2 interface (4-32 cycles), NCU interface (4 to 7 IO clock cycles)

This stage includes latency of either the ILDs or IND. In the IND, SIU crosses back from the cmp clock domain into the IO domain. That pointer synchronization plus writing and reading from the width conversion FIFO take 2 IO clock cycles. The subsequent transfer to NCU takes 2 to 5 IO clock cycles. In the ILDs, there are 2

cycles of header (Addr Tag) and (0 to 16 cycles) for regular RDD, WRI, WR8 transaction. However, when there is a WRM request, since L2 only merges 8 bytes, a WRM will be broken down to a maximum of 8 WR8 transfers. With each WR8 being 4 cmp cycles (2cycles of header and 2 cycles of payload), and assuming L2 can stream the merge pipeline, the worse case is 32 cmp cycles for 8 transfers. The best case is RDD (2 cycles of header + 3 dummy data cycles) and the worse case is WRM (32 cycles).

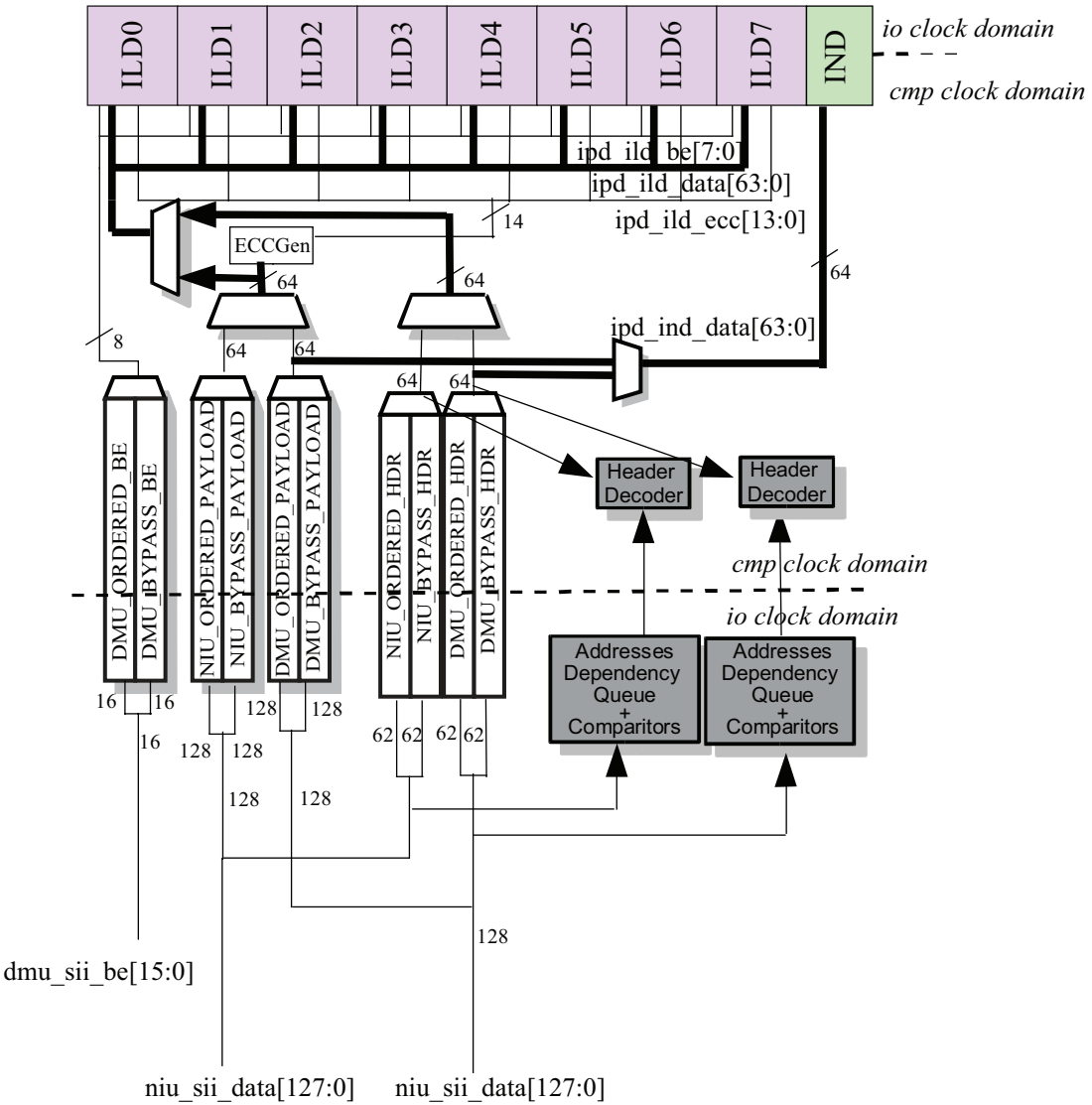
FIGURE 6-9 Inbound Pipeline Diagram



6.4.6 Block Diagrams of SIU Inbound

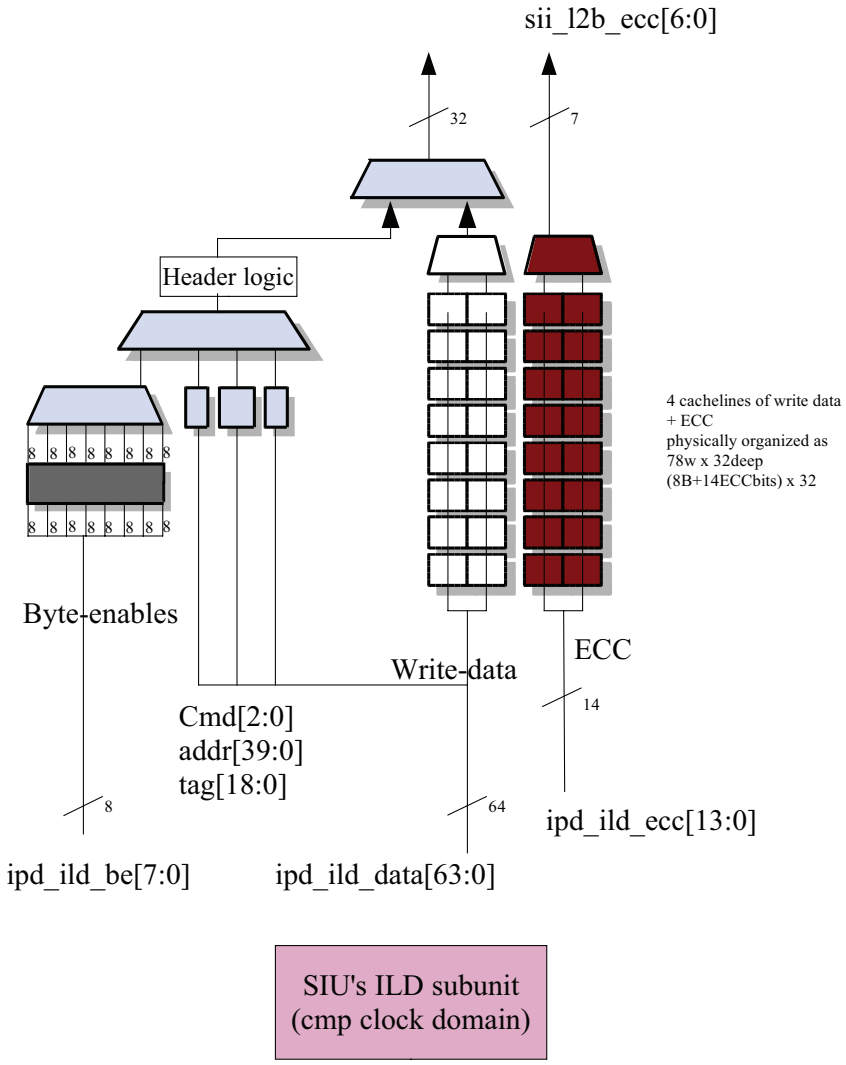
6.4.6.1 Top

FIGURE 6-10 SIU Inbound Top Level



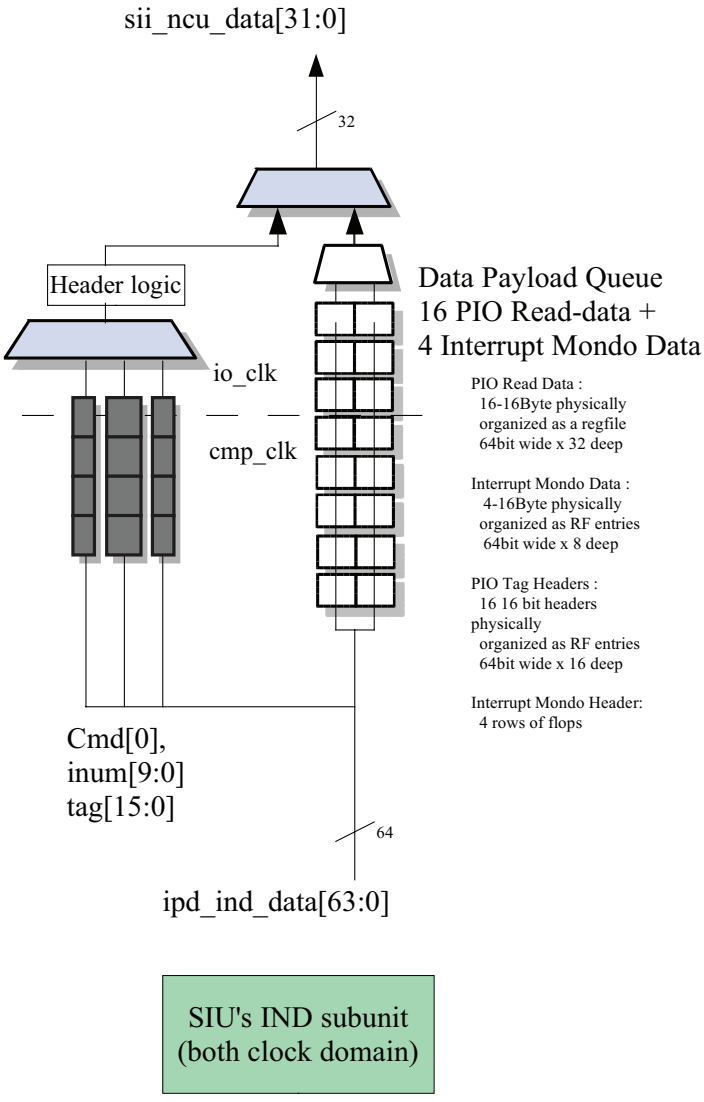
6.4.6.2 Sub-Blocks - ILD

FIGURE 6-11 SIU Inbound L2 Datapath (ILD) Subunit



6.4.6.3 Sub-Block - IND

FIGURE 6-12 SIU Inbound NCU Datapath (IND) Subunit



6.4.6.4 Sub-Block Descriptions

Top

The logical top level of the SIU inbound logic consist of control sub-blocks and data

path sub-blocks. The datapath include the following sub-blocks:

ILDs : There are 8 identitcal copies of ILD instantiated in the inbound logic. Each ILD can hold 4 requests and 4 cachelines of write data and associated byte-enables, ecc and header information. ILDs handle transaction going to L2 cache, it has 1 unified data queue 78bits wide x 32 deep to hold the 4 cachelines - writing in 64 bit data + 14 bit ecc per cycle. It acts as buffer for assembling of SIU_L2 packets.

IND : IND handles transfer going to NCU. It acts as a buffer for assembling SIU_NCU packets. It can buffer 16 PIO Read 16Byte responses and 4 interrupt 16Byte mondo write requests (2 32-bit x 40 data queues, 4x26-bit register for Interrupt header storage, 16 16-bit register for PIO Read Return header storage). IND runs in both clock domains. It downconverts the cmp clock frequency of IPD into the IO clock frequency because NCU's interface to SIU runs in the IO clock domain.

DMU_ORDERED header queue : it is a 62-bit wide x 16 depth register file with input and output registered. The write operation is running at Io_clock domain, and the read operation is running at L2_clock domain.

DMU_BYPASS header queue : it is a 62-bit wide x 16 depth register file with input and output registered. The write operation is running at Io_clock domian, and the read operation is running at L2_clock domain.

NIU_ORDERED header queue : it is a 62-bit wide x 16 depth register file with input and output registered. The write operation is running at Io_clock domain, and the read operation is running at L2_clock domain.

NIU_BYPASS header queue : it is a 62-bit wide x 16 depth register file with input and output registered. The write operation is running at Io_clock domian, and the read operation is running at L2_clock domain.

DMU_ORDERED parity queue : it is a 1-bit x 16 register. It hold the parity bit of the whole packet including header and payload.

DMU_BYPASS parity queue : it is a 1-bit x 16 register. It hold the parity bit of the whole packet including header and payload.

NIU_ORDERED parity queue : it is a 1-bit x 16 register. It hold the parity bit of the whole packet including header and payload.

NIU_BYPASS parity queue : it is a 1-bit x 16 register. It hold the parity bit of the whole packet including header and payload.

DMU_ORDERED_BE queue : it is a 16-bit x 16 register file with input and output registered. The write operation running at Io_clock doman, and the read operation is running at L2 clock domain.

DMU_BYPASS_BE queue : it is a 16-bit x 16 register file with input and output registered. The write operation running at Io_clock doman, and the read operation is running at L2 clock domain.

DMU_ORDERED payload queue : it is a 128-bit x 64 depth queue logically. It is implemented with 1 128-bit x 32 depth register file with input and and output registered. The write operation runs at Io_clock domain and the read operation runs at L2_clock domain.

DMU_BYPASS payload queue : it is a 128-bit x 64 depth queue logically. It is implemented with 1 128-bit x 64 depth register file with input and and output registered. The write operation runs at Io_clock domain and the read operation runs at L2_clock domain.

NIU_ORDERED payload queue : it is a 128-bit x 64 depth queue logically. It is implemented with 1 128-bit x 32 depth register file with input and and output registered. The write operation runs at Io_clock domain and the read operation runs at L2_clock domain.

NIU_BYPASS payload queue : it is a 128-bit x 64 depth queue logically. It is implemented with 1 128-bit x 64 depth register file with input and and output registered. The write operation runs at Io_clock domain and the read operation runs at L2_clock domain.

The control path include the following sub-blocks: IPC, ILC, please refer to the next subsections for the details of IPC and ILC sub-blocks.

ILC Sub-block

There are 8 identical copies of ILCs instantiated in SIU. ILCs are running at core clock domain. There are 2 major functions of ILC sub-block:

1. It checks the L2 bank's availability. There are 2 counters in each ILC to keep track of outstanding L2 transactions. One is the transaction counter, which keep track of outstanding L2 requests which are issued to L2 and no acknowledgments come back yet. The second counter keeps track of the WRI requests. The requirements from L2 stated that L2 allow 2 outstanding requests (WR8, RDD, WRM) and 4 outstanding WRI requests. As long as the counters' value satisfy the L2 requirements, the particular L2 bank is considered as available. The availability information will be passed to IPC for arbitration purposes.
2. ILC will drive the SIU-L2 interface bus according to the protocol defined in the previous sections. Also it will assemble the packets with header and payload formats according to the SIU-L2 packet format defined in the previous sections.

Note – For the WR8 transfer with no bit set in the byte-mask, SIU will just discard the transaction.

For WRM of 64 byte merge, ILC will break it down to 8 WR8 requests and assemble 8 packets for it.

Should a WRM of 64 bytes have no bytes on, SIU will not send out any WR8 packets to L2 but will wait for all outstanding acks from L2 to return and then inject a single fake write response into the outbound L2 subblock for return to DMU if needed.

INC Sub-block

The INC sub-block is running at core clock domain, however part of the logic is running at IO clock domain for the purpose of driving the SIU-NCU interface signals. There is no flow control between SIU and NCU for read returns, NCU is considered as always available with respect to read returns. However INC need to keep track of outstanding interrupts, there are 4 outstanding interrupts allowed in SIU. Once it reached that number, it will signal the IPC to block further interrupt requests to NCU. Another function of INC is to assemble the packet following the SIU-NCU packet format and drive the SIU_NCU interface accordingly.

IPC Sub-block

There are 2 identical copies of IPCs instantiated in SIU, one for DMU and one for NIU. IPCs are mainly running at IO clock domain. However, part of the logic is running at core core clock domain to handle the cross clock domain situation. The logic will be partitioned into 2 parts (IPCC and IPCS) according their clock domain.

IPCS sub-block implement 2 major functions:

1. IPCS will drive the DMU/NIU – SIU interface bus according to the protocol defined in previous section. It check for the availability of different queues to maintain flow control for the interface.

2. IPCS maintains the ordering rules for the input side from the DMU and NIU. It uses the sideband signals and header information to dispatch requests from DMU to DMU_ORDERED queue and DMU_BYPASS queue and from NIU to NIU_ORDERED queue and NIU_BYPASS queue.

IPCS has access to 2 FIFOs containing addresses and write/read bit duplicating the addresses and command type in the ordered and bypass FIFOs. Assuming the register-file FIFOs are not cam-able, the 2 FIFOs must be made of flops for address comparison. IPCS maintains 5 pointers – the location of the youngest write entry in the bypass queue, the youngest entry in the bypass queue, the youngest entry in the

ordered queue, the oldest entry in the ordered queue, the oldest entry in the bypass queue. IPCS and IPCC communicates with each other to maintain these pointers. All packets from an interface updates the duplicate address FIFOs for that interface.

For the newest write entering the ordered queue, IPCS tags it as dependent on the younger of the 2 entries: youngest write in the bypass queue or the youngest matching cacheline address in the bypass queue. This is done by storing a pointer to the bypass queue and setting a dependency pointer valid bit.

For the newest read entering the ordered queue, IPCS tags it as dependent on the youngest entry with the same address in the bypass queue. This is done by storing a pointer to the bypass queue and setting a dependency pointer valid bit.

For the newest write or read entering the bypass queue, IPCS tags it as dependent on the youngest entry with the same address in the ordered queue. This is done by storing a pointer to the ordered queue and setting a dependency pointer valid bit.

In PCIExpress mode, IPCS tags each newest entry as dependent on the youngest entry in the opposite queue.

IPCC sub-block implements 2 major functions:

1. IPCC will drive the output buses of the inbound packet fifos according to the protocol defined in previous section. It check for the availability of different queues in IND and ILDs to maintain flow control.

2. IPCC maintain the ordering rules for the output side for DMU packet and does a 2 level arbitration between NIU and DMU. The top level arbitration is between NIU and DMU packets on a deficit round robin basis. NIU packets have no ordering requirement with respect to other packets from DMU. DMU packets have no ordering requirement with respect to packets from NIU. The 2nd level arbitration is between the the 2 DMU FIFOs.

IPCC maintains 2 counters – a bypass write counter and an ordered write counter. The bypass write counter counts the number of writes sent from the bypass queue but have not received their acknowledgements. The ordered write counter counts the number of writes or interrupts that were sent from the ordered queue but have not received their acknowledgements. An ordered target ID tracks which of the 9 targets (8 L2, 1 NCU) was the last issued write/interrupt from the ordered queue.

When a write/interrupt/PIO read return reaches the top of the ordered queue and has its dependency pointer valid bit set, IPCC first checks if that entry in the bypass queue has been dequeued. If it has and the bypass write counter reaches zero, then a 2nd check is made. After the bypass write counter has reached zero, if it's a read return then it may continue, else the following is decided. A comparison between the ordered target ID against the destination of the write/interrupt is made. If they are the same, then it may continue. If they are not the same, then the write/interrupt

must wait until the ordered write counter has reached zero before it can continue. When the write/interrupt packet dequeues, the ordered write counter is incremented and the ordered target ID is updated.

When a read reaches the top of the ordered queue and has its dependency pointer valid bit set, IPCC first checks if that entry in the bypass queue has been dequeued. If it has and the bypass write counter reaches zero, then a 2nd check is made. A comparison between the ordered target ID against the destination of read is made. If they are the same, then it may continue. If they are not the same, then the read must wait until the ordered write counter has reached zero before it can continue. No counter is incremented when the read dequeues.

When a write or read reaches the top of the bypass queue and has its dependency pointer valid bit set, then IPCC first checks if that entry in the ordered queue has been dequeued. If it has, then the write may continue. Once dequeued and if its a write, then the bypass write counter is incremented. No counter is incremented when a read dequeues.

When a flush reaches the top of the ordered queue and has its dependency pointer valid bit set, IPCC checks if that entry in the bypass queue has been dequeued. If it has and the bypass write counter reaches zero, then a 2nd check is made. After the bypass write counter has reached zero, must wait until the ordered write counter has reached zero before it is dequeued. When a flush dequeues and it's nonposted, a response packet is injected into the outbound path to return to DMU.

6.4.6.5 RAS

Syndrome format of sii_ncu_syn_data[63:0]

sii_ncu_syn_vld will cover the 16 io cycles of syndrome.

Cycle0 : send sii_ncu_syn_data[3:0]

Cycle1 : send sii_ncu_syn_data[7:4]

.

.

Cycle15 : send sii_ncu_syn_data[63:60]

```
==================================
>> bit[63:62]     = 2'b00         (in case of future changes !)
>> bit [61]      = niud_pe
>> bit [60]      = niua_pe
>> bit [59]      = niuctag_ue
>> bit [58]      = dmud_pe
>> bit [57]      = dmua_pe
>> bit [56]      = dmuctag_ue
```

```
>> bit [55:40]  = ctag[15:0]
>> bit [39:0]   = Physical Address[39:0]
>> ================================
```

6.5 Outbound

6.5.1 Interface Timing Diagrams

6.5.1.1 From L2 to SIU

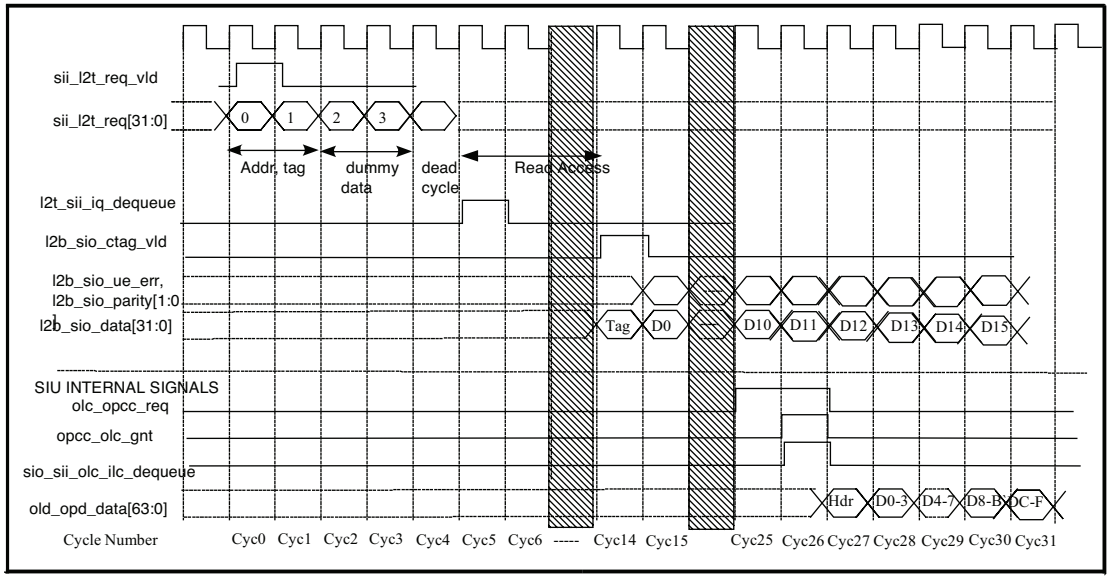
Single Read Response from L2

FIGURE 6-13 shows the quickest a read request inbound to an L2 bank will return outbound to SIU. The greyed bars in the diagram mean some clock cycles are not shown.

After L2Tag has accumulated the read request packet with the dummy cycles for pipeline alignment (shown as sii_l2t_req[31:0]), the read request can be dispatched down L2's pipeline. The l2t_sii_iq_dequeue signal asserts high for 1 cycle when L2 dispatches the read. That signal is used by SIU's Inbound L2 Control subunit for credit based flow control. In the current L2 pipeline, the earliest L2 can assert l2t_sii_iq_dequeue is 2 cycles after the last dummy data cycle of the request packet.

L2 Bank asserts the l2b_sio_ctag_vld signal high for 1 cycle to indicate the completion of the read. On the cycle that l2b_sio_ctag_vld asserts, l2b_sio_data has a read response header. The subsequent 16 cycles of l2b_sio_data[31:0] contain the 64B read data. Parity is generated on l2b_sio_parity[1:0] and corresponds to the parity of each data word of l2b_sio_data. The l2b_sio_ue_err signal is also active during the data cycles to indicate that L2 had detected an uncorrectable error for that data word.

FIGURE 6-13 L2 Read Data Return Timing Diagram (Fastest case is shown)



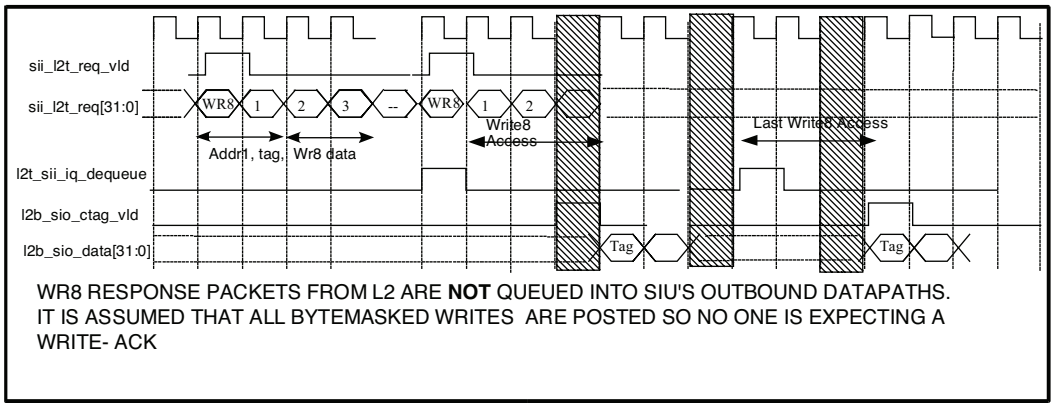
The bottom part of the timing diagram represents the read response continuing outbound internally through SIU. Read requests can be pipelined to L2 and the responses pipelined back to SIU. However because there is no direct flow control to stop a response from L2 from overflowing SIU's outbound response buffers, SIU's Outbound L2 Control subunit asserts internal signal sio_sii_olc_ilc_dequeue high for a cycle when a credit is returned to SIU's Inbound L2 Control subunit. The diagram shows SIU forwarding the read response packet to the outbound packet data subunit on the next data bus (old_opd_data[63:0]) as early as possible. The earliest this can happen is after SIU has received 3/4 the response payload from L2. The assumption made here is that SIU can forward the accumulated result of all the parity checks and uncorrectable errors later. Otherwise, SIU pays the latency of accumulating the full 64B response payload. Working backward from the databus old_opd_data are SIU's outbound internal arbitration request-grant signals olc_opc_req and opc_olc_gnt. The dequeue signal sio_sii_olc_ilc_dequeue for the inbound path can simply be a buffered version of the grant signal opc_olc_gnt as shown, if the timing can be made. Otherwise it will assert a cycle later.

Due to stall conditions such as a cache miss, the latency between the assertion of l2t_sii_iq_dequeue and l2b_sio_ctag_vld can exceed the 9 cycles shown in the above diagram. Likewise, due to back stall conditions further downstream in the outbound paths, the latency between the assertion of sii_l2t_req_vld and sio_sii_olc_ilc_dequeue can exceed the 26 cycles shown in the above diagram.

Write 8 Responses from L2 to SIU

FIGURE 6-14 shows 2 WR8 requests inbound to an L2 bank followed by 3 signals to related to the WR8 responses returning outbound to SIU from L2. The greyed bars in the diagram mean some clock cycles are not shown.

FIGURE 6-14 L2 Write 8 Acknowledgement Timing Diagram



After L2Tag has accumulated the WR8 request packet (shown as sii_l2t_req[31:0]), the request can be dispatched down L2's pipeline. The l2t_sii_iq_dequeue signal asserts high for 1 cycle when L2 dispatches the write. That signal is used by SIU's Inbound L2 Control subunit for credit based flow control. In the current L2 pipeline, the earliest L2 can assert l2t_sii_iq_dequeue is 2 cycles after the last data cycle of the request packet. The l2t_sii_iq_dequeue signal guarantees write ordering has occurred in L2. However, it does not mean the write has completed.

For the WR8 acknowledgement, the latency between l2t_sii_iq_dequeue and l2b_sio_ctag_vld depends on whether the store missed the L2 or not. L2 handles the merge like an atomic read-modified-write operation. A subsequent request from SIU would not overtake the WR8 should the WR8 miss L2 on the first pass and must wait to be reissued after a fill. Note that there is no new assertion of l2t_sii_iq_dequeue on 2nd pass of a WR8.

The timing diagram shows that when l2b_sio_ctag_vld asserts, L2 Bank responds with a tag on signal l2b_sio_data[31:0]. This tag does not contain enough information for SIU to know if the WR8 response is the last of a sequence of up to 8 WR8 packets that were decomposed from a single write transaction. The assumption made in the SIU is that DMU will never need a ack response for writes with bytemasks (all memory writes with bytemask from DMU are posted) and hence SIU Inbound and Outbound paths do not need to coordinate to scoreboard the tag.

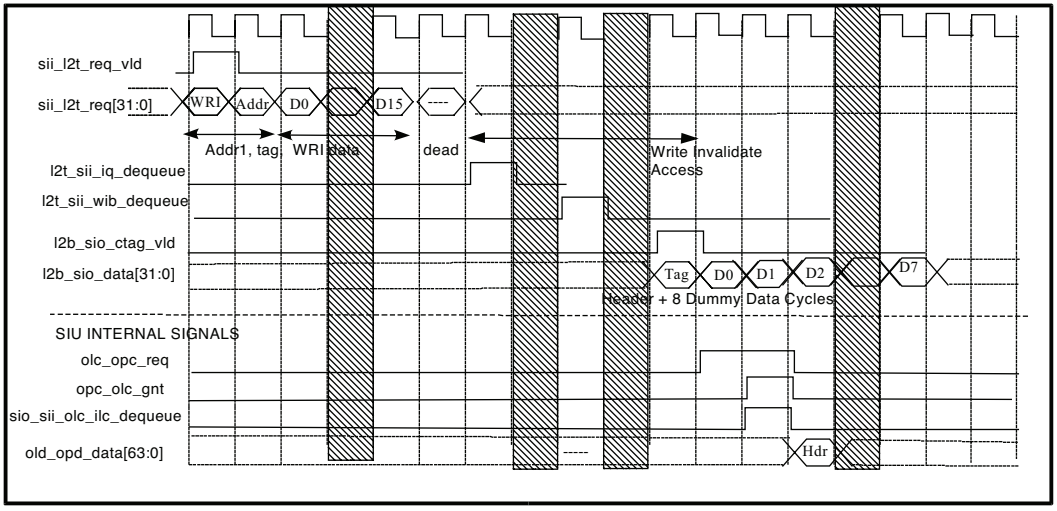
Similiarly, the assumption is that NIU will never generate bytemask writes requests to SIU and thus SIU will never generate WR8 for NIU and again SIU would not need to scoreboard the tag after SIU issue to L2.

Because the outbound path never needs to respond to a WR8 response from L2, those responses never enters any queue in the SIU outbound L2 data or control paths.

Write Invalidate Response from L2 to SIU

FIGURE 6-15 shows a Write Invalidate request inbound to an L2 bank and the outbound return of the write response. The greyed bars in the diagram mean some clock cycles are not shown.

FIGURE 6-15 L2 Write Invalidate Acknowledgement Timing Diagram



After L2Tag has accumulated the 18-cycle WRI request packet (shown as sii_l2t_req[31:0]), the request can be dispatched down L2's pipeline. The l2t_sii_iq_dequeue signal asserts high for 1 cycle when L2 dispatches the write. That signal is used by SIU's Inbound L2 Control subunit for credit based flow control. In the current L2 pipeline, the earliest L2 can assert l2t_sii_iq_dequeue is 2 cycles after the last data cycle of the request packet. The l2t_sii_iq_dequeue signal guarantees write ordering has occurred in L2. However, it does not mean the write has completed. When L2 Tag drains the 64 bytes of write data from its I/O Write Buffer and the data are enroute to the memory controller, the l2t_sii_wib_dequeue signal asserts for 1 cycle. This signal is used only for a Write Invalidate request as a Write 8 request does no consume any data buffer in L2. The SIU Inbound L2 subunit

monitors the signal l2t_sii_wib_dequeue for credit based flow control of L2's 4 I/O Write Buffers. Many cycles later, the l2b_sio_ctag_vld signal asserts for 1 cycle to indicate the completion of the write. When l2b_sio_ctag_vld asserts, l2b_sio_data contains the header of the write response packet. 8 cycles of dummy data follows to align L2's store pipeline. The bottom part of the timing diagram represents the write response continuing outbound through SIU. For flow control, the outbound L2 control subunit signals to the inbound L2 control subunit that an entry has been dequeued.

6.5.1.2 From SIU to NIU

The assumption is that the NIU has buffers to receive all DMA responses from SIU. There is no flow control from NIU to throttle SIU Outbound and thus SIU is allowed to send responses back to back without any bubble to NIU.

Writes from NIU can be either posted or nonposted. For nonposted writes, SIU must return an acknowledgement response back to NIU when the L2 has acknowledged completion of the write. There is no requirement of the SIU to return the ack response packet in the same order that NIU delivered the write request nor in the order that the write completed in memory.

The parity protected interfaces between SIU and the NIU are 128 bit wides with side band signals for packet control. Having a 128 bit for header allows SIU to provide a rich set of transaction types and allows SIU to provide a uniform and generic but flexible enough for most IO architectures. See Chapter 7.4 and 7.6 for the exact header format from SIU to DMU and from SIU to NIU. See Chapter 9 for the exact signal names.

The outbound packet interface protocol works as follows :

Cycle 1: Header Cycle

- sio_niu_hdr_vld asserts for 1 cycle to indicate SIU is sending the packet header to niu.
- sio_niu_datareq is set to 1 to indicate this packet has a 4 cycle payload following the header cycle to transfer 64 Bytes of data. It is set to 0 to indicate this packet is an write acknowledge and has no data payload.
- sio_niu_data[127:0] contains a valid header.

Cycle 2-5: Payload Cycles if sio_niu_datareq was asserted during Header Cycle.

- sio_niu_hdr_vld is deasserted.
- sio_niu_data[127:0] contains the payload data. Data is returned big endian and critical 4 Byte first and wraps back to the beginning of the cacheline when it reaches the cacheline boundary.
- sio_niu_parity[3:0] contains the parity for each 32 bit of data. Parity[N]= xor(data[32N+31 : 32N])

6.5.1.3 From SIU to DMU

The assumption is that the DMU has buffers to receive all DMA responses from SIU. There is no flow control from DMU to throttle SIU Outbound and thus SIU is allowed to send responses back to back without bubble to DMU.

Fire-DMU never issues nonposted DMA writes so all responses from SIU to Fire-DMU has a 4 cycle payload.

The parity protected interfaces between SIU and the DMU are 128 bit wides with side band signals for packet control. Having a 128 bit for header allows SIU to provide a rich set of transaction types and allows SIU to provide a uniform and generic but flexible enough for most IO architectures. See Chapter 7.4 and 7.6 for the exact header format from SIU to DMU and from SIU to DMU. See Chapter 9 for the exact signal names.

The outbound packet interface protocol works as follows :

Cycle 1: Header Cycle

- sio_dmu_hdr_vld asserts for 1 cycle to indicate SIU is sending the packet header to dmu.
- sio_dmu_datareq is set to 1 to indicate this packet has a 4 cycle payload following the header cycle to transfer 64 Bytes of data.
- sio_dmu_data[127:0] contains a valid header.

Cycle 2-5: Payload Cycles if sio_dmu_datareq was asserted during Header Cycle.

- sio_dmu_hdr_vld is deasserted.
- sio_dmu_data[127:0] contains the payload data. Data is returned big endian and critical 4 Byte first and wraps back to the beginning of the cacheline when it reaches the cacheline boundary.
- sio_dmu_parity[3:0] contains the parity for each 32 bit of data. Parity[N]= xor(data[32N+31 : 32N])

6.5.1.4 From SIO to TCU

There will be 2 bits interface (sio_tcu_vld, sio_tcu_data) from SIO to TCU for DMA read/write response.

sio_tcu_data is a 64-bit data stream for read request , and 1bit of data=0 for write request.

Header format :

bit[63:0] = 8-bytes of read return data for read request

bit[0] = 0x0

6.5.2 Outbound Pipeline

6.5.2.1 From L2

L2 Bus Cycle Packets (Write Acknowledge)

1. L2B-OLD Header Enqueue (1 L2 cycle : 32 bit bus)
2. OLD-OPD Request (1+ L2 cycle)
3. OLD-OPD Grant (1 L2 cycle)
4. OLD-OPD Transmit/Muxing Wire Delay (1 to 2 L2 cycles)
5. OLD-OPD Header Enqueue (1 L2 cycle : 64 bit bus)
6. OPD-OPD Domain Crossing (1-3 L2 cycles)
7. OPD-DMU/NIU Header Enqueue (1 IO cycle : 128 bit bus)

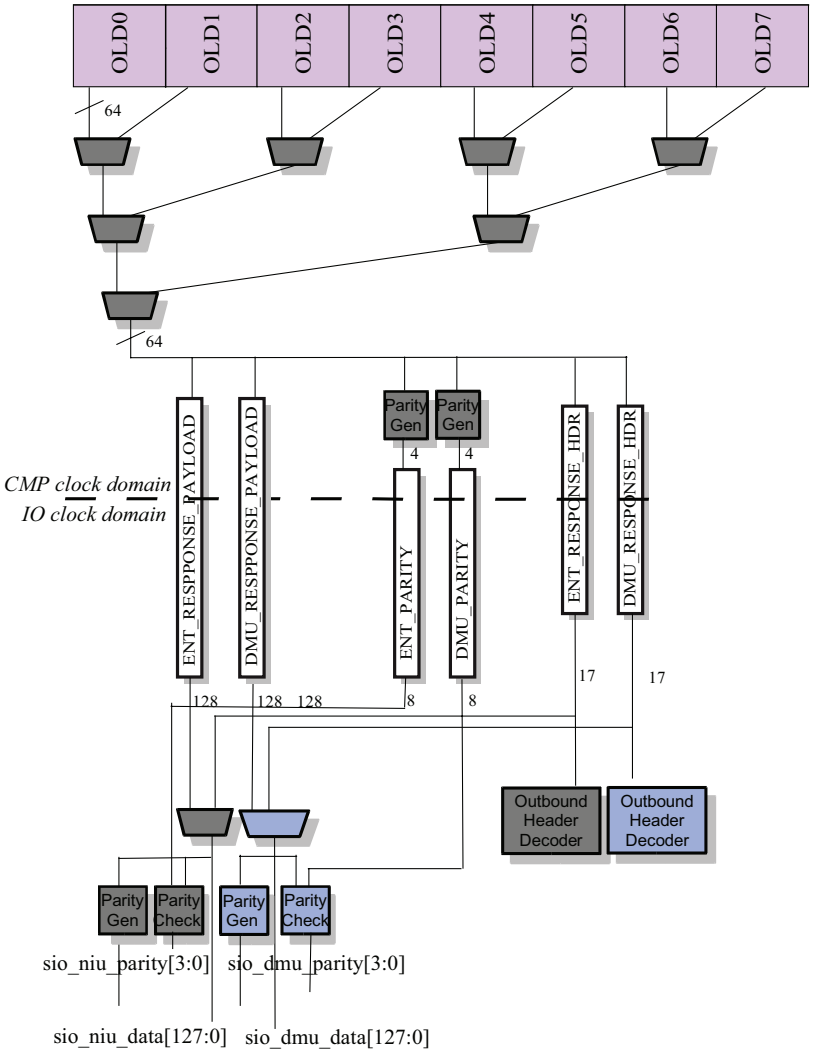
L2 Bus Cycle Packets (Read Response)

1. L2B-OLD Header Enqueue (1 L2 cycle : 32 bit bus)
2. L2B-OLD Data Payload Enqueue (16 L2 cycles : 32 bit bus)
3. OLD-OPD Request (1+ L2 cycles)
4. OLD-OPD Grant (1 L2 cycle)
5. OLD-OPD Transmit/Muxing Wire Delay (1 to 2 L2 cycles)
6. OLD-OPD Header Enqueue (1 L2 cycle : 64 bit bus)
7. OLD-OPD Data Payload Enqueue (8 L2 cycles : 64 bit bus)
8. OPD-OPD Domain Crossing (1-3 L2 cycles)
9. OPD-DMU/NIU Header Enqueue (1 IO cycle : 128 bit bus)
10. OPD-DMU/NIU Data Payload Enqueue (4 IO cycles : 128 bit bus)

6.5.3 SIU Outbound Block Diagram

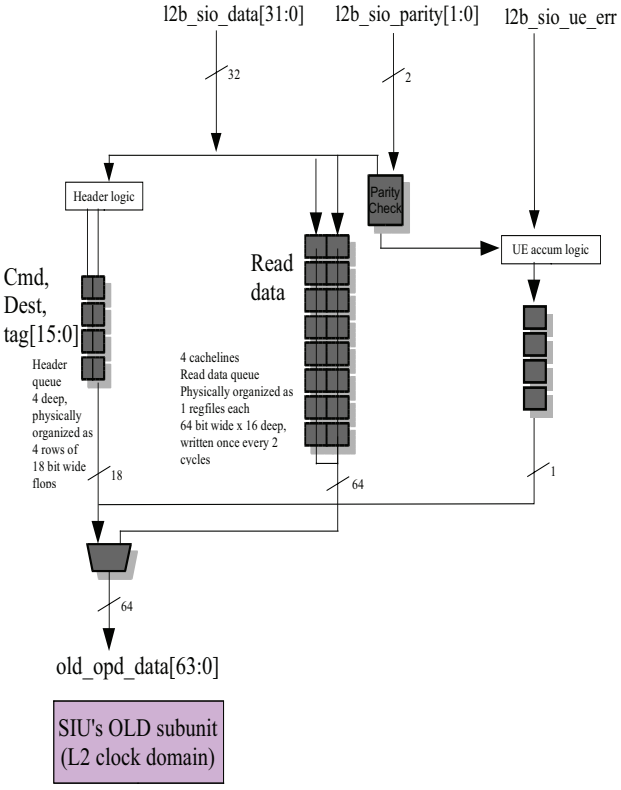
6.5.3.1 OPD : Outbound Packet Datapath

FIGURE 6-16 SIU Outbound Packet Datapath (OPD) Subunit



6.5.3.2 OLD : Outbound L2 Datapath

FIGURE 6-17 SIU Outbound L2 Datapath (OLD) Subunit



6.5.4 SIU Outbound Subunit Descriptions

6.5.4.1 Datapath

OLD0 to OLD7 receive and store response packets from L2 buffer. Each interface has a 64Byte 4 deep payload buffers physically organized as 2 regfiles. Each regfile is 32bit wide x 32 deep – written in at cmp clock @ 32bit / cycle, read out at cmp clock @ 64bit / cycle. There is also a header queue in each interface to hold the tag information. That queue is 18 bits wide x 4 deep.

The physical placement and organization of these OLDx queues are critical to the critical path in the outbound direction. The distance between the farthest 2 regfile's IO flops determines how far the mux-select lines would need to travel on the 1st level of 8:1 muxing of these 64 bit wire bundles.

OPD : The outbound packet datapath subunit holds packets from the OLDx subunits and stream them out to DMU and NIU. NIU and DMU each has a separate outbound packet queue 16 entry deep for responses from L2. The packet queues are physically separated into header queues, payload queues, and parity. The packets crosses clock domain from core clock to IO clock in OPD.

6.5.4.2 Control Path

OLC0 to OLC7 are indentical instantiations of the control logic to enqueue and dequeue from the OLD FIFOs. Each makes a request to OPCC and waits for a grant before it ships the DMA response over to the Ooutbound packet datapath and FIFOs.

OPCC : The outbound packet control logic in the core clock domain monitors all the 8 request lines from OLC0 to OLC7, checks that the destination FIFO is available and then drives grant and controls the mux selects for the 8:1 muxes. It also transfer FIFO write pointers to the IO clock domain so the the corresponding outbound packet control logic in the IO clock domain can handle pushing the data out to DMU or NIU. OPCC also monitors the L2 response headers to signal to the inbound side how many and type of responses received so IPCC can do bookkeeping.

OPCS: The outbound packet control logic in the IO clock domain monitors the pessimistic FIFO write pointers from OPCC and communicates with DMU or NIU.

6.6 Packet Formats

6.6.1 Inbound To L2

6.6.1.1 WRI Packet

Write Invalidate (WRI) request must be 64 byte aligned and a full 64 bytes is written to memory. The bottom 6 bits of address for a WRI request are set to zero.

FIGURE 6-18 Write Invalidate Request

V	*ECC[6:0]*	*Packet Data [31:0]*			
	6 5 4 3 2 1 0	31 30 29 28 27 26 25 24	21 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
1	CtagECC[5:0]	J O P E S 1 0 0	Tag[15:0]		Address[39:32]
	Invalid	Address[31:6], 6'b000000			
	Valid ECC	DataByte0	...	...	DataByte3
	Valid ECC	DataByte4	...	...	DataByte7
	Valid ECC	DataByte8	...	...	DataByte11
	Valid ECC	DataByte12	...	...	DataByte15
	Valid ECC	DataByte16	...	...	DataByte19
	Valid ECC	DataByte20	...	...	DataByte23
	Valid ECC	DataByte24	...	...	DataByte27
	Valid ECC	DataByte28	...	...	DataByte31
	Valid ECC	DataByte32	...	...	DataByte35
	Valid ECC	DataByte36	...	...	DataByte39
	Valid ECC	DataByte40	...	...	DataByte43
	Valid ECC	DataByte44	...	...	DataByte47
	Valid ECC	DataByte48	...	...	DataByte51
	Valid ECC	DataByte52	...	...	DataByte55
	Valid ECC	DataByte56	...	...	DataByte59
0	Valid ECC	DataByte60	...	...	DataByte63

J :Jtag access : 1= Jtag access from tcu, 0= regular dma packet from IO

O:Ordered bit : 1=From SIU Inbound Ordered Queue. Needed by SIU.

P:Posted bit : 1=Posted => Completion Ack by SIU to the source NOT needed,

0=Nonposted => Completion Ack by SIU to the source NEEDED

E: Error bit : (parity or uncorrectable error in header)

S:Source : 1=DMU, 0=NIU

T:Tag[15:0] generated by the source to track the transaction.

If the P bit is zero, the response packet will contain this 16-bit tag.

For this implementation, DMU and NIU will both guarantee packets with address errors (like unmapped address or illegal access) will not be sent to SIU. This was different than in OpenSPARC T1 where OpenSPARC T1's JBI set the error bit and changed the Physical Address to all zeros.

Parity error will be logged in the header if SIU or a prior unit had detected an uncorrectable error in the data payload.

Legal WRI Packet Encodings:

L2 does not look at the O,P,S and T fields from SIU but simply pipe them along back to SIU Outbound when L2 generates the response packet.

For this implementation, NIU never sends bytemasked writes which means all DMA writes from NIU will become WRI's not WR8. In NIU's case, DMA writes can be issued to the SIU's ordered queue or the SIU's bypass queue. However, a high percentage of NIU's DMAs will go into the the bypass queue. NIU's writes can be posted or nonposted. Fire-DMU never issue nonposted memory writes. In the Fire-DMU implementation, all their DMA writes are restricted to the ordered queue.

So if the NIU and DMU interfaces are behaving correctly,

the following header restrictions apply for WRI:

P must be 1 if from Fire-DMU.

O must be 1 if Fire-DMU.

PA[5:0] must be all 0s.

Header cycle bits 26:24 == 3'b100

6.6.1.2 WR8 Packet

For the Write 8 bytes (WR8) request, random byte writes are supported provided at least one byte gets written . A byte mask field [7:0] is supported for the random byte writes with at least one byte mask = 1. Bytemask field is positional. The address for a WR8 request must be 8-byte aligned (the lower 3 bits of the address must be 0).

FIGURE 6-19 Write 8 Bytes Request

V	ECC[6:0]							Packet Data [31:0]																													
	6	5	4	3	2	1	0	31	30	29	28	27	26	25	24	21	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	CtagEcc[5:0]							J	O	P	E	S	0	1	0									ByteMask[7:0]							Address[39:32]							
0	Invalid							Address[31:3], 3'b000																															
0	Valid ECC							DataByte0								...								...								DataByte3							
0	Valid ECC							DataByte4								...								...								DataByte7							

J :Jtag access : 1= Jtag access from tcu, 0= regular dma packet from IO

O:Ordered bit : 1=From SIU Inbound Ordered Queue. Needed by SIU.

P:Posted bit : 1=posted. Needed by SIU

E: Error bit : (parity or uncorrectable error)

S:Source : 1=DMU, 0=NIU;

Legal WR8 Packet Encodings:

L2 does not look at the O,P,S fields from SIU but simply pipe them along back to SIU Outbound when L2 generates the response packet.

For this implementation, NIU never sends bytemasked writes which means all DMA writes from NIU will become WRI's not WR8. Fire-DMU can send DMA writes that will decompose into WR8. Fire-DMU never issue nonposted memory writes. In the Fire-DMU implementation, all their DMA writes are restricted to the ordered queue.

So if the NIU and DMU interfaces are behaving correctly,

the following header restrictions apply for WR8:

- S must be 1.
- P must be 1.
- O must be 1 if Fire-DMU.
- PA[2:0] must be 0s.
- Header cycle bits 26:24 == 3'b010

6.6.1.3 RDD Packet

FIGURE 6-20 RDD Requests

V	*ECC[6:0]*	*Packet Data [31:0]*																															
	6	5	4	3	2	1	0	31	30	29	28	27	26	25	24	21	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	CtagEcc[5:0]							J	O	P	E	S	0	0	1	Tag[15:8]								Tag[7:0]								Address[39:32]							
0	Invalid							Address[31:0]																															
0	Invalid							Dummy Cycle0																															
0	Invalid							Dummy Cycle1																															

J :Jtag access : 1= Jtag access from tcu, 0= regular dma packet from IO

O:Ordered bit : 1=From SIU Inbound Ordered Queue. Needed by SIU.

P:Posted bit : reads are nonposted so this bit should always be 0

E: Error bit : (parity or uncorrectable error)

S:Source : 1=DMU, 0=NIU

T:Tag[15:0] generated by the source to track the transaction.

The response packet will contain this 16-bit tag.

Legal RDD Packet Encodings:

RDD requests will always read a full 64 byte cache line, although L2 does not require the address to be any alignment. NIU and Fire-DMU will always align to a 64 Byte address boundary. Note there must be 2 dummy data cycles in the read request from SIU to L2 to match the pipeline format for WR8.

L2 does not look at the O,P,S and T fields from SIU but simply pipe them along back to SIU Outbound when L2 generates the response packet.

For this implementation, NIU can issue reads to either the SIU's ordered or bypass queue. However, a high percentage of NIU's DMAs will go into the the bypass queue. For Fire-DMU implementation, all their DMA reads are restricted to the ordered queue.

So if the NIU and DMU interfaces are behaving correctly,

the following header restrictions apply for RDD:

- P must be 0
- O must be 1 if Fire-DMU
- PA[5:0] must be all 0s if from Fire-DMU or NIU
- Header cycle bits 26:24 == 3'b001

6.6.2 Outbound from L2

6.6.2.1 RDD Response Packet

L2 drives back a packet with a 64 Byte cacheline payload, critical 32-bits first. Should the address sent on the SIU-L2 interface be not aligned to a 64Byte boundary, L2 will align the responses to a 4-word (32 bit) boundary. The initial 32-bit doubleword within the 64Byte line is indicated by Address[5:2]. Data responses will start at the specified address, continuing sequentially to the end of the cache line and then wrap.The Read bit (bit 16) is set to 1. The same tag is returned to SIU. For this example, the requested address had addr[5:0]=101101 (or decimal 45). Note that byte 44 is returned first by L2.

Note that L2 returns the CBA (Critical Byte Address – address[2:0]) from the original RDD request. NIU and Fire-DMU always read on a 64Byte boundary so this is not an issue and data is therefore always returned at the cacheline boundary and CBA[2:0] should always be 0 when the SIU->L2 read request is legal.

L2 also pipes back to SIU Outbound the fields O,P,S, Tag as sent by SIU Inbound. The field E indicates there was an error. SIU Outbound will pass the error condition downstream to DMU or NIU.

Example FIGURE 6-21 shows PA[5:0]=101101 (decimal 45).

FIGURE 6-21 RDD Response Packet when PA[5:0] is not all zeros.

Legal RDD Response Packet Encodings:

Refer to the Legal RDD request section.

Given those restrictions, if the NIU and DMU interfaces and SIU Inbound are behaving correctly,

the following header are expected for RDD responses:

J Jtag access : 1= Jtag access from tcu, 0= regular dma packet from IO

UE : Uncorrectable error generated by L2, internal L2 error

CtagEcc : check bit for Tag[15:0]

P must be 1

O must be 1 if Fire-DMU

CBA[2:0] must be all 0s if from Fire-DMU or NIU

S must be 1 if from Fire-DMU. Must be 0 if from NIU

Header cycle bit 16 must be 1

Tag[15:0] matches Tag sent by NIU/Fire-DMU

6.6.2.2 Write Invalidate Response Packet

FIGURE 6-22 WRI Response Packet

V	*UE, Paritys*	*Packet Data [31:0]*
	UE 1 0	31 30 29 28 27 26 25 24 21 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1	Invalid	J CtagEcc[5:0] UE O P E S 0 0 0 0 Tag[15: 0]
0	Invalid	Dummy Cycle0
0	Invalid	Dummy Cycle1
0	Invalid	Dummy Cycle2
0	Invalid	Dummy Cycle3
0	Invalid	Dummy Cycle4
0	Invalid	Dummy Cycle5
0	Invalid	Dummy Cycle6
0	Invalid	Dummy Cycle7

A write invalidate response packet looks like a read response packet filled with 8 cycles of dummy data. Because of the L2 pipeline, after a write response, the next response (either read or write) cannot come until after 8 pad cycles. SIU Outbound will drop all write invalidate responses with the 'P'osted bit set.

L2 pipes back to SIU Outbound the fields O,P,S, Tag as sent by SIU Inbound. The field E indicates there was an error. SIU Outbound will pass the error condition downstream to DMU or NIU if P bit was a 0.

Legal WRI Response Packet Encodings:

Refer to the Legal WRI request section.

Given those restrictions, if the NIU and DMU interfaces and SIU Inbound are behaving correctly,

the following header bits are expected for WRI responses:

UE : Uncorrectable error generated by L2, internal L2 error

CtagEcc : check bit for Tag[15:0]

P must be 1 if from Fire-DMU.

O must be 1 if Fire-DMU

S must be 1 if from Fire-DMU. Must be 0 if from NIU

Header cycle bit 19:16 must be 0s

Tag[15:0] matches Tag sent by NIU/Fire-DMU

6.6.2.3 Write 8 Response Packet

FIGURE 6-23 WR8 Response Packet

A write 8 response packet looks like a read response packet filled with 8 cycles of dummy data. Because of the L2 pipeline, after a write response, the next response (either read or write) cannot come until after 8 pad cycles. SIU Outbound will drop all write 8 responses with the 'P'osted bit set. Because the SIU to L2 WR8 request packet header does not provide enough bits for a full 16 bit Tag field and because it is guaranteed that all writes with bytemasks from Fire-DMU are posted and because NIU does not do any writes with bytemasks, SIU outbound's implementation drops all WR8 responses. However, SIU Outbound does decode the header fields so SIU Inbound can do buffer management and therefore expects L2 to pipe back to SIU Outbound fields O,P,S as sent by SIU Inbound. The field E indicates there was an error. Currently, there is no mechanism to log an WR8 response packet with an error.

Legal WR8 Response Packet Encodings:

Refer to the Legal WR8 request section.

Given those restrictions, if the NIU and DMU interfaces and SIU Inbound are behaving correctly,

the following header bits are expected for WR8 responses:

J Jtag access : 1= Jtag access from tcu, 0= regular dma packet from IO

UE : Uncorrectable error generated by L2, internal L2 error

CtagEcc : check bit for Tag[15:0]

P must be 1.

O must be 1 if Fire-DMU

S must be 1.

Header cycle bit 19:16 must be 0s

6.6.2.4 DMA Read Request Packet from NIU to SIU

All read requests by NIU will return 64 Bytes aligned to 64B boundary. The 1 cycle packet on niu_sii_data[127:0] contains only the header. The header format is shown below with the SUPPORTED settings needed for Read Requests by NIU. The 16 bit ID is the captured into the tag field sent up to L2 by SIU and later returned back to NIU. There are no payload cycles for the request packet. During the packet transfer, NIU indicates whether the DMA Read packet will go into an ordered queue or a bypass queue. The reads do not fill in L2. A high percentage of DMA Reads go to the bypass queue. And reads are by default nonposted.

TABLE 6-4 NIU to SIU : DMA Read Request Header Format

NIU's Header Cycle niu_sii_data[msb:lsb] for a Read	Name	Usage
127:122	Command	
	127=Response bit	Must be 0
	126=Posted request bit	Must be 0
	125=Read bit	Must be 1
	124=Write ByteMask Active	Must be 0
	123=L2 bit	Must be 1
	122=NCU bit	Must be 0
121:85	Reserved	Must Be Zero

TABLE 6-4 NIU to SIU : DMA Read Request Header Format

84:83	Address parity	Bit 84 for odd addres bit Bit 83 for even address it
82:82	TimeOutError	1=This packet had Timed Out
81:81	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address
80:80	UncorrectableError	Must be 0 because read request does not carry data payload
79:64	ID[15:0]	NIU supplies an ID that it can use to track the responses later
63	Reserved	Must be Zero
62	Command Parity	Parity bit for bit127-122
61:56	CtagEcc	6bit sec-dec check bit for ID
55-40	Reserve	Must be Zero
39:0	PA[39:0]	Must be 64B aligned - PA[5:0] must be zeros

6.6.2.5 DMA Write Request Packet from NIU to SIU

NIU always sends a full cacheline of data for writes. The header format is shown below with the SUPPORTED settings needed for Write Requests by NIU. The 16 bit ID is the captured into the tag field sent up to L2 by SIU. The address is always 64 Byte aligned. All DMA Write Request from NIU can be either nonposted (requires a write response from SIU to indicate completion) or posted (no response from SIU). Prior to the packet transfer, NIU indicates whether the packet will go into an ordered queue or a 'bypass' queue. The writes do not allocate in L2. Writes to the ordered queue will always be be issued by the SIU to L2 after the youngest write in the bypass queue. The writes in the bypass queues are not ordered with respect to other writes in the bypass queue. NIU will never use byte-masks so if niu_sii_be[15:0] exists at the interface, NIU always drives them to all 1's.

TABLE 6-5 NIU to SIU Write Request Packet Format

Packet Cycle Number	Packet Content of niu_sii_data[127:0]
1	NIU's Write Header
2	Byte0, Byte1, ..., Byte15

TABLE 6-5 NIU to SIU Write Request Packet Format

3	Byte16, Byte17, ..., Byte31
4	Byte32, Byte33, ..., Byte47
5	Byte48, Byte49, ..., Byte63

The header encoding for a DMA Write from NIU is shown in TABLE 6-6.

TABLE 6-6 NIU to SIU : DMA Write Request Header Format

NIU's Header Cycle niu_sii_data[msb:lsb] for a Write	Name	Usage
127:122	Command 127=Response bit 126=Posted request bit 125=Read bit 124=Write ByteMask Active 123=L2 bit 122=NCU bit	 Must be 0 0=NIU needs an ack Must be 0 Must be 0 (all bytes on) Must be 1 Must be 0
121:85	Reserved	Must Be Zero
84:83	Address parity	Odd, even address parity
82:82	TimeOutError	1=This packet had Timed Out
81:81	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address
80:80	UncorrectableError	1=data payload has uncorrectable error
79:64	ID[15:0]	NIU supplies an ID that it can use to track the responses later
62	Command parity	Command parity[127:122]
61:56	CtagEcc[5:0]	6-bit ECC check bit for ID
55:40	Reserved	Must be Zero
39:0	PA[39:0]	Must be 64B aligned - PA[5:0] must be zeros

6.6.3 Outbound to NIU

6.6.3.1 DMA Write Response Packet from SIU to NIU

If NIU had requested a response (write-ack), SIU will return a packet to indicate the completion of the write. The write responses can return out of order.

The write response is a one cycle packet containing only a header. Immediately following a write response can be another write or read response.

TABLE 6-7 SIU to NIU : DMA Write Response Header Format

SIO's Header Cycle sio_niu_data[msb:lsb] for Write Response	Name	Usage
127:122	Command 127=Response bit 126=Posted request bit 125=Read bit 124=Write ByteMask Active 123=L2 bit 122=NCU bit	 Must be 1 Must be Ignored (driven to 0) Must be 0 Must be Ignored (driven to 0) Must be Ignored (driven to 1) Must be Ignored (driven to 0)
121:84	Reserved	Must Be Zero
83:83	Reserved	Must Be Zero
82:82	Reserved	Must Be Zero
81:81	UncorrectableError for prior address, ctagEcc	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address
80:80	Data Error	1=data payload has a detected uncorrectable error. This could be: 1. timeout errors 2. unmapped errors 3. data ue error from L2$ or dram
79:64	ID[15:0]	ID supplied originally by NIU
63:40	Reserved	Must be Zero
39:0	PA[39:0]	Must be Ignored

6.6.3.2 DMA Read Response packet from SIU to NIU

TABLE 6-8 SIU to NIU : DMA Read Response Packet Format

Packet Cycle Number	Packet Content of sio_niu_data[127:0]
1	SIU to NIU DMA Read Response Header
2	Byte0, Byte1, ..., Byte15
3	Byte16, Byte17, ..., Byte31
4	Byte32, Byte33, ..., Byte47
5	Byte48, Byte49, ..., Byte63

The read response packet from SIU to NIU is 1 cycle of header followed by 4 cycles of data payload. The above data order assumes that NIU had set the lower 6 bits of the PA to zero. Otherwise L2 and SIU will return data critical 32-bit doubleword first and wrap around the 64 Byte boundary. The read response header encoding is defined in TABLE 6-9:

TABLE 6-9 SIU to NIU Read Response Header Format

SIO's Header Cycle sio_niu_data[msb:lsb] for Read Response	Name	Usage
127:122	Command 127=Response bit 126=Posted request bit 125=Read bit 124=Write ByteMask Active 123=L2 bit 122=NCU bit	 Must be 1 Must be Ignored (driven to 0) Must be 1 Must be Ignored (driven to 0) Must be Ignored (driven to 1) Must be Ignored (driven to 0)
121:84	Reserved	Must Be Zero
83:83	Reserved	Must Be Zero
82:82	Reserved	1=This packet had Timed Out
81:81	UncorrectableError for prior address, ctagEcc	1=This packet's address and ctagEcc err
80:80	DE data error	1=data payload has a detected uncorrectable error. This could be: 1. timeout errors 2. unmapped errors 3. data ue error from L2$ or dram

TABLE 6-9 SIU to NIU Read Response Header Format *(Continued)*

79:64	ID[15:0]	ID supplied originally by NIU
63:40	Reserved	Must be Zero
39:0	PA[39:0]	Must be Ignored

6.6.4 Inbound from DMU

See Chapter 12 (Verification Cases) to see what packet types are supported/nonsupported/illegal. They are based on assumed usage models from NIU and Fire-DMU.

6.6.4.1 Packet from =Fire-DMU to SIU

There are 4 expected/supported types of packet transfers from Fire-DMU to SIU

DMA Read Request : A packet with only a header cycle and no payload cycles. Addresses must be 64-Byte aligned. Must be steered into SIU's Inbound Ordered Queue.

Interrupt Write Request: A packet with a header cycle and fixed size of 1 payload cycle with all 16 bytes in the payload valid. Must be steered into SIU's Inbound Ordered Queue.

DMA Write Request : A packet with a header cycle and fixed size of 4 payload cycles. Addresses must be 64-Byte aligned and always posted. Must be steered into SIU's Inbound Ordered Queue.

PIO Read Data Return : A packet with a header cycle and fixed size of 1 payload cycle. SIU and NCU will transport the full 16 bytes. Only the cpu cares which byte(s) within the 16 bytes are enabled. Must be steered into SIU's Inbound Bypass Queue.

TABLE 6-10 for dmc_tag[15:0] is referred to by all packets from Fire-DMU. S ee the DMC MAS spec pg. 7.611 for more details.

TABLE 6-10 Fire-DMC Tag

Field	Bits	Description
DMA transactions		
dmc_tag[15]	type	0b-indicates DMA/Int transactions

TABLE 6-10 Fire-DMC Tag *(Continued)*

dmc_tag[14:11]	cl_tag[3:0]	Dmc transaction number for tracking credits
dmc_tag[10:6]	d_ptr[4:0]	Used for DMA Rds only-dou dma rd buffer address
dmc_tag[5:1]	pkt_tag[4:0]	Used for DMA Rds only-PSB index for building packet records
dmc_tag[0]	cl_sts	Used for DMA Rds only-indicates 1st cacheline in packet sequence
Int Transactions		
dmc_tag[15]	type	0b-indicates DMA/Int transactions
dmc_tag[14:11]	cl_tag[3:0]	Dmc transaction number for tracking credits
dmc_tag[10:3]	Rsv[7:0]	reserved
dmc_tag[2:1]	mdo_tag[1:0]	mondo_tag for mondo-reply to IMU
dmc_tag[0]	rsv	Must be 0
MMU Tablewalk Transactions		
dmc_tag[15]	type	1b-indicates MMU Tablewalk transactions
dmc_tag[14:11]	cl_tag[3:0]	Dmc transaction number for tracking credits
dmc_tag[10:6]	Rsv[4:0]	reserved
dmc_tag[5:0]	Mtag[5:0]	Used for MMU tablewalks only-MMU tag for tracking tablewalks
PIO Cpl Transactions		

TABLE 6-10 Fire-DMC Tag *(Continued)*

dmc_tag[15:13]	Rsv[2:0]	Must be 3'b100
dmc_tag[12:9]	jbc_trans_#[3:0]	Pio transaction credit id
dmc_tag[8:0]	thread_id[8:0]	Thread id of PIO read request

Note – The NCU will distinquish interrupts from PIO cpl's by using dmc_tag[15].

DMA Read Request packet from Fire-DMU to SIU

All read requests by Fire-DMU will return 64 Bytes aligned to 64B boundary. The 1 cycle packet on dmu_sii_data[127:0] contains only the header. The header format is shown below with the SUPPORTED settings needed for Read Requests by Fire-DMU. The 16 bit ID is the captured into the tag field sent up to L2 by SIU and later returned back to Fire-DMU. There are no payload cycles for the request packet. During the packet transfer, Fire-DMU must always steer DMA Reads into the ordered queue. The reads do not fill in L2. And reads are by default nonposted.

TABLE 6-11 Fire-DMU to SIU : DMA Read Request Header Format

Fire-DMU's Header Cycle dmu_sii_data[msb:lsb] for a Read	Name	Usage
127:122	Command 127=Response bit 126=Posted request bit 125=Read bit 124=Write ByteMask Active 123=L2 bit 122=NCU bit	 Must be 0 Must be 0 Must be 1 Must be 0 Must be 1 Must be 0
121:85	Reserved	Must Be Zero
84:83	Address parity	Bit 84 for odd addres bit Bit 83 for even address it
82:82	TimeOutError	1=This packet had Timed Out, used only by Fire-DMU for PIO Rd Completions

TABLE 6-11 Fire-DMU to SIU : DMA Read Request Header Format *(Continued)*

81:81	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address used only by Fire-DMU for PIO Rd Completions
80:80	UncorrectableError	used only by Fire-DMU for PIO Rd Completions
79:64	ID[15:0] (dmc_tag[15:0])	DMU supplies an ID that it can use to track the responses later. See dmc_tag TABLE 6-10
63	Reserved	Must be Zero
62	Command Parity	Parity bit for bit127-122
61:56	CtagEcc	6bit sec-dec check bit for ID
55-40	Reserve	Must be Zero
39:0	PA[39:0]	Must be 64B aligned - PA[5:0] must be zeros

DMA Write Request Packet from Fire-DMU to SIU

Fire-DMU does not always send a full cacheline of data for writes. The header format is shown below with the SUPPORTED settings needed for Write Requests by Fire-DMU. The 16 bit ID is the captured into the tag field sent up to L2 by SIU. The address is always 64 Byte aligned. All DMA Write Request from Fire-DMU are posted (no response from SIU). Fire-DMU must always steer DMA Writes into the SIU's ordered queue. The writes do not allocate in L2. Fire-DMU can send DMA writes with all bytes enabled and with 1 or more bytes at the beginning and/or the end of the 64Bytes disabled.

TABLE 6-12 Fire-DMU to SIU Write Request Packet Format

Packet Cycle Number	Packet Content of dmu_sii_data[127:0]
1	Fire-DMU's Write Header
2	Byte0, Byte1, ..., Byte15
3	Byte16, Byte17, ..., Byte31
4	Byte32, Byte33, ..., Byte47
5	Byte48, Byte49, ..., Byte63

The header encoding for a DMA Write from Fire-DMU is shown in TABLE 6-13.

TABLE 6-13 Fire-DMU to SIU : DMA Write Request Header Format

Fire-DMU's Header Cycle dmu_sii_data[msb:lsb] for a Write	Name	Usage
127:122	Command 127=Response bit 126=Posted request bit 125=Read bit 124=Write ByteMask Active 123=L2 bit 122=NCU bit	 Must be 0 Must be 1 Must be 0 0=Write full cacheline. 1=WRM Must be 1 Must be 0
121:84	Reserved	Must Be Zero
83:83	Reserved	Must Be Zero
82:82	TimeOutError	1=This packet had Timed Out, used only by Fire-DMU for PIO Rd Completions
81:81	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address used only by Fire-DMU for PIO Rd Completions
80:80	UncorrectableError	1=data payload has uncorrectable error used only by Fire-DMU for PIO Rd Completions
79:64	ID[15:0] (dmc_tag[15:0])	Fire-DMU supplies an ID that it can use to track the responses later. See dmc_tag TABLE 6-10.
63	Reserved	Must be Zero
62	Command Parity	Parity bit for bit127-122
61:56	CtagEcc	6bit sec-dec check bit for ID
55-40	Reserve	Must be Zero
39:0	PA[39:0]	Must be 64B aligned - PA[5:0] must be zeros

Interrupt Write Request packet from Fire-DMU to SIU

Fire-DMU can send an interrupt to NCU via SIU. The interrupt is always a mondo type with 16 bytes of payload. NCU decodes the ID field to determine how to process the mondo and to differentiate it from a PIO Completion. SIU transports the full 16 bit ID to NCU. Interrupts must be steered toward the ordered queue.

TABLE 6-14 Fire-DMU to SIU : Interrupt Write Request Packet Format

Packet Cycle Number	Packet Content of dmu_sii_data[127:0]
1	Fire-DMU's Interrupt Write Header
2	Mondo Byte0, Byte1, ..., Byte15

The header encoding for an Interrupt Write from Fire-DMU is shown in TABLE 6-15.

TABLE 6-15 Fire-DMU to SIU : Interrupt Write Request Header Format

Fire-DMU's Header Cycle dmu_sii_data[msb:lsb] for an Interrupt	Name	Usage
127:122	Command 127=Response bit 126=Posted request bit 125=Read bit 124=Write ByteMask Active 123=L2 bit 122=NCU bit	 Must be 0 Must be 0 Must be 0 Must be 0 Must be 0 Must be 1
121:84	Reserved	Must Be Zero
83:83	Reserved	Must Be Zero
82:82	TimeOutError	1=This packet had Timed Out used only by Fire-DMU for PIO Rd Completions
81:81	UnmappedAddressError	Must be 0 used only by Fire-DMU for PIO Rd Completions
80:80	UncorrectableError	Must be 0 used only by Fire-DMU for PIO Rd Completions

TABLE 6-15 Fire-DMU to SIU : Interrupt Write Request Header Format *(Continued)*

79:64	ID[15:0]	See NCU spec or see dmc_tag TABLE 6-10 for Interrupt ID encoding
63:40	Reserved	Must be Zero
39:0	PA[39:0]	Must be Ignored

PIO Read Completion Packet from Fire-DMU to SIU

Fire-DMU can send a PIO Read completion packet to NCU via SIU. The PIO Read completion has a 1 cycle payload. SIU transports the full 16 bit ID and 16 byte payload to NCU. PIO Read completions from Fire-DMU must be steered toward the bypass queue. Only cpu cares which byte(s) within the 16 bytes are enabled.

TABLE 6-16 Fire-DMU to SIU : PIO Read Completion Response Packet Format

Packet Cycle Number	Packet Content of dmu_sii_data[127:0]
1	Fire-DMU's PIO Read Completion Header
2	Byte0, Byte1, Byte2, Byte3, Byte4, Byte5, Byte6, Byte7, Byte8, Byte9, Byte10, Byte11, Byte12, Byte13, Byte14, Byte15

The header encoding for a PIO Read Completion from Fire-DMU is shown in TABLE 6-17.

TABLE 6-17 Fire-DMU to SIU : PIO Read Completion Packet Header Format

Fire-DMU's Header Cycle dmu_siu_data[msb:lsb] for PIO completions	Name	Usage
127:122	127=Response bit	Must be 1
	126=Posted request bit	Must be 0 (Ignored by SIU if Response bit is set)
	125=Read bit	Must be 1
	124=Write ByteMask Active	Must be 0 (Ignored by SIU if Response bit is set or if Read bit is set)
	123=L2 bit	Must be 0
	122=NCU bit	Must be 1
121:84	Reserved	Must Be Zero
83:83	Reserved	Must Be Zero

TABLE 6-17 Fire-DMU to SIU : PIO Read Completion Packet Header Format *(Continued)*

82:82	TimeOutError	1=This packet had Timed Out
81:81	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address
80:80	UncorrectableError	1=data payload has a detected uncorrectable error
79:64	ID[15:0]	for PIO read completions this is PIOID [15:13]=3'b100 [12:9] will be the credit id returned on PIO rd completions, [8:0] will be the {3'b000, cpu-thread ID[5:0]}. See NCU or DSN specification.
62	Command parity	Command parity for [127:122]
63:40	Reserved	Must be Zero
39:0	PA[39:0]	Must be Ignored.

6.6.5 Outbound to DMU

6.6.5.1 Packet from SIU to Fire-DMU

There are 1 types of packet transfers from SIU to Fire-DMU

DMA Read Response : A packet with a header cycle followed by a fixed size of 4 payload cycles with all bytes valid. Because Fire-DMU's DMA Read Request are always cacheline aligned, the data returned always starts at the cacheline boundary. Data format is big endian.

TABLE 6-18 SIU to Fire-DMU : DMA Read Response Packet Format

Packet Cycle Number	Packet Content of sio_dmu_data[127:0]
1	SIU to Fire-DMU's DMA Read Response Header
2	Byte0, Byte1, ..., Byte15
3	Byte16, Byte17, ..., Byte31
4	Byte32, Byte33, ..., Byte47
5	Byte48, Byte49, ..., Byte63

The header format is shown in TABLE 6-19.

TABLE 6-19 SIU to Fire-DMU : Outbound Packet Header Format

SIO to Fire-DMU's Header Cycle sio_dmu_data[msb:lsb]	Name	Usage
127:122	Command Legal combinations - DMA Read Response	 1010_10
	127=Response bit	Must be 1
	126=Posted request bit	Must be Ignored (driven to 0)
	125=Read bit	Must be 1
	124=Write ByteMask Active	Must be Ignored (driven to 0)
	123=L2 bit	Must be 1
	122=NCU bit	Must be 0
121:84	Reserved	Must Be Zero
83:83	Reserved	Must Be Zero
82:82	TimeOutError	1=This packet had Timed Out
81:81	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address
80:80	UncorrectableError	1=response has a detected uncorrectable error. This could be: 1. timeout errors 2. unmapped errors 3. data ue error from L2$ or dram
79:64	ID[15:0] (dmc_tag[15:0])	For Response, this is DMU's ID
63:62	Reserved	Must be Zero
61:56	CtagEcc[5:0]	6-bit ECC check bit for ID
55:40	Reserved	Must be Zero
39:0	Bus Address[39:0]	For Responses, SIU does not return the Address.

6.6.6 Inbound to NCU

6.6.6.1 Packet from SIU to NCU

There is 1 cycle of header followed by 4 cycles of payload data. The format of the header is shown in TABLE 6-20.

TABLE 6-20 SIU to NCU : Inbound Packet Header Format

Header Cycle sii_ncu_data[msb:lsb]	Name	Usage
31:31	TimeOutError	1=This packet had Timed Out
30:30	UnmappedAddressError	1=This packet's address mapped to an nonexistent, reserved, or erroroneous address
29:29	UncorrectableError	1=packet has an uncorrectable error
28	SIU Ctag Uncorrectable Error	1= ctag uncorrectable error
27-22	Reserve	Must be Zero
21:16	ctag	Ecc check bit for ID [15:0]
15:0	ID[15:0] as originally sent by DMU. dmc_tag[15:0]	If Interrupt is from DMU, NCU returns the entire tag back to DMU with mondo_ack or mondo_nack signal asserted. If PIO Completion is from DMU, dmc_tag[12:9] = NCU credit id will be returned to the credit pool dmc_tag[8:0] = {3'b000, cpu-thread id[5:0]}

```
DMU sending to SIU as the following [127:0]
Byte0     Byte1...          Byte15
Byte16   Byte17...      Byte31
Byte 32  Byte33...      Byte47
Byte 48  Byte49...      Byte63

SIU send to L2 is  [31:0]
Byte0 Byte1 Byte2 Byte3                          cycle1
Byte 4 Byte5 Byte6 Byte7                         cycle2
:
:
Byte 60 Byte61 Byte62  Byte63                cycle16
```

```
SIU send to NCU will  be [31:0]
Byte0 Byte1 Byte2 Byte3                          cycle1
Byte4 byte5 Byte6  Byte7                          cycle2
:
Byte12 Byte13 Byte14 Byte15                    cycle4
```

6.7 CSR

SIU has no CSRs. Data parity errors and uncorrectable errors detected are signaled in the packet header to the receiving unit. It is assumed that the end unit will log the error.

The following is a summary of what the Eagle Team's CSR tool look like and may be a candidate for connecting/generating CSRs within Niagara2. If debug control status registers are defined in the future, then SIU will participate with whatever methodology is defined for accessing CSRs.

Control Status Register (CSR)

All the control and status registers inside the Niagara2 are generated by the CSR Tool, so that we can reduce some of the coding effort and standardize the register access among different modules within Niagara2. All the modules with CSR registers will be connected with CSR specific interface in a ring fashion.

CSR interface signals:

```
Inputs:
clk                  - clock signal for the design
rst                  - reset signal for the flops
csrbus_data_in    - data to writeto CSR
csrbus_addr          - address to send to select one CSR
csrbus_src_bus    - source bus identification
csrbus_valid         - specifies that address and data lines are
valid


Outputs:
csrbus_mapped     - asserted when CSR within module is selected
```

```
csrbus_acc_vio   - improper access is attempted
csrbus_done          - transaction is completed
csrbus_data_out  - data read from the CSR
```

CSR Read/Write access:

For read from CSR, csrbus_valid is asserted along with address and source bus information, data come back on csrbus_data_out when csrbus_done is asserted.

For write operation, csrbus_valid is asserted along with address and data info, then the carbus_done is asserted shortly after within the same cycle.

6.8 Unit Level Signals

6.8.1 SIU-L2 Interface List

TABLE 6-21 SIU-L2 Interface List

Signal Name	I/O	Size	From/To	Timing	Description
SII to L2Tag signals					
sii_l2t0_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank0
sii_l2t1_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank1
sii_l2t2_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank2
sii_l2t3_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank3
sii_l2t4_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank4
sii_l2t5_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank5
sii_l2t6_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank6
sii_l2t7_req_vld	O	1	SIU->L2T		Packet request valid (first cycle) to L2Tag for bank7
sii_l2t0_req	O	32	SIU->L2T		Packet header/data for bank0
sii_l2t1_req	O	32	SIU->L2T		Packet header/data for bank1
sii_l2t2_req	O	32	SIU->L2T		Packet header/data for bank2
sii_l2t3_req	O	32	SIU->L2T		Packet header/data for bank3
sii_l2t4_req	O	32	SIU->L2T		Packet header/data for bank4
sii_l2t5_req	O	32	SIU->L2T		Packet header/data for bank5
sii_l2t6_req	O	32	SIU->L2T		Packet header/data for bank6
sii_l2t7_req	O	32	SIU->L2T		Packet header/data for bank7
SII to L2Buffer signals					
sii_l2b0_ecc	O	7	SIU->L2T		Packet ecc for bank0

TABLE 6-21 SIU-L2 Interface List *(Continued)*

sii_l2b1_ecc	O	7	SIU->L2B		Packet ecc for bank1
sii_l2b2_ecc	O	7	SIU->L2B		Packet ecc for bank2
sii_l2b3_ecc	O	7	SIU->L2B		Packet ecc for bank3
sii_l2b4_ecc	O	7	SIU->L2B		Packet ecc for bank4
sii_l2b5_ecc	O	7	SIU->L2B		Packet ecc for bank5
sii_l2b6_ecc	O	7	SIU->L2B		Packet ecc for bank6
sii_l2b7_ecc	O	7	SIU->L2B		Packet ecc for bank7
L2Tag to SII signals					
l2t0_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank0 request
l2t1_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank1 request
l2t2_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank2 request
l2t3_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank3 request
l2t4_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank4 request
l2t5_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank5 request
l2t6_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank6 request
l2t7_sii_iq_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank7 request
l2t0_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank0 write invalidate data buffer
l2t1_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank1 write invalidate data buffer
l2t2_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank2 write invalidate data buffer
l2t3_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank3 write invalidate data buffer
l2t4_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank4 write invalidate data buffer
l2t5_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank5 write invalidate data buffer
l2t6_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank6 write invalidate data buffer
l2t7_sii_wib_dequeue	I	1	L2T->SIU		L2Tag is unloading a bank7 write invalidate data buffer
L2Buffer to SIO signals					
l2b0_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 0

TABLE 6-21 SIU-L2 Interface List *(Continued)*

l2b1_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 1
l2b2_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 2
l2b3_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 3
l2b4_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 4
l2b5_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 5
l2b6_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 6
l2b7_sio_ctag_vld	I	1	L2B->SIU		Response packet Valid (First Cycle) from L2 bank 7
l2b0_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank0
l2b1_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank1
l2b2_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank2
l2b3_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank3
l2b4_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank4
l2b5_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank5
l2b6_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank6
l2b7_sio_data	I	32	L2B->SIU		Read data/write response packet from L2 bank7
l2b0_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank0
l2b1_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank1
l2b2_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank2
l2b3_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank3
l2b4_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank4
l2b5_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank5
l2b6_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank6

TABLE 6-21 SIU-L2 Interface List *(Continued)*

l2b7_sio_ue_err	I	1	L2B->SIU		UE on Read data from L2 bank7
l2b0_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank0
l2b1_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank1
l2b2_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank2
l2b3_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank3
l2b4_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank4
l2b5_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank5
l2b6_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank6
l2b7_sio_parity[1:0]	I	2	L2B->SIU		Parity on Read data from L2 bank7

6.8.2 SIU-NCU Interface List

TABLE 6-22 SIU-NCU Interface List

Signal Name	I/O	Size	From/To	Timing	Description
NCU to SII					
ncu_sii_gnt	I	1	NCU->SIU		Grant – xfr packet to NCU starting next cycle
ncu_sii_dmuctag_uei	I	1	NCU->SIU		Inject uncorrectable error for ctag
ncu_sii_dmuctag_cei	I	1	NCU->SIU		Inject correctable error for ctag
ncu_sii_dmua_pei	I	1	NCU->SIU		Inject address parity error
ncu_sii_dmud_pei	I	1	NCU->SIU		Inject Data parity error
ncu_sii_niuctag_uei	I	1	NCU->SIU		Inject uncorrectable error for ctag
ncu_sii_niuctag_cei	I	1	NCU->SIU		Inject correctable error for ctag
ncu_sii_niua_pei	I	1	NCU->SIU		Inject address parity error
ncu_sii_niud_pei	I	1	NCU->SIU		Inject data parity error
SII to NCU					
sii_ncu_req	O	1	SIU->NCU		Packet request from SIU to NCU
sii_ncu_data	O	32	SIU->NCU		Packet header/data from SIU to NCU
sii_ncu_parity[1:0]	O	2	SIU->NCU		Parity on data from SIU to NCU
sii_ncu_dmuctag_ue	O	1	SIU->NCU		Uncorrectable error for ctag
sii_ncu_dmuctag_ce	O	1	SIU->NCU		Correctable error for ctag
sii_ncu_dmua_pe	O	1	SIU->NCU		Address parity error
sii_ncu_dmud_pe	O	1	SIU->NCU		Data parity error
sii_ncu_niuctag_ue	O	1	SIU->NCU		Uncorrectable error for ctag
sii_ncu_niuctag_ce	O	1	SIU->NCU		Correctable error for ctag
sii_ncu_niua_pe	O	1	SIU->NCU		Address parity error
sii_ncu_niud_pe	O	1	SIU->NCU		Data parity error
sii_ncu_syn_vld	O	1	SIU->NCU		Syndrome valid signal
sii_ncu_syn_data[3:0]	O	4	SIU->NCU		Syndrome bus total 16 cycle xfr syndrome
SIO to NCU					
sio_ncu_ctag_ue	O	1	SIU->NCU		Uncorrectable error for ctag
sio_ncu_ctag_ce	O	1	SIU->NCU		Correctable error for ctag

TABLE 6-22 SIU-NCU Interface List *(Continued)*

sio_ncu_data_parity	O	1	SIU->NCU		Data parity error
Partial L2 Bank Mode bits					
ncu_sii_pm	I	1	NCU->SIU		0=all 8 banks available 1=partial mode and need to look at each ncu_sii_ba* signals
ncu_sii_ba01	I		NCU->SIU		0=bank0 and bank1 unavailable 1=both banks available
ncu_sii_ba23	I		NCU->SIU		0=bank2 and bank3 unavailable 1=both banks available
ncu_sii_ba45	I		NCU->SIU		0=bank4 and bank5 unavailable 1=both banks available
ncu_sii_ba67	I		NCU->SIU		0=bank6 and bank7 unavailable 1=both banks available
L2 Index Hashing Enable bit					
ncu_sii_l2_idx_hash_en	I		NCU->SIU		1=enable hashing of PA for L2 index.

6.8.3 SIU-NIU Interface List

TABLE 6-23 SIU-NIU Interface List

Signal Name	I/O	Size	From/To	Timing	Description
NIU to SII signals					
niu_sii_hdr_vld	I	1	NIU->SIU		Asserted during the header phase of any requests from NIU to SIU. Not asserted during the data transfer phase.
niu_sii_reqbypass	I	1	NIU->SIU		Valid during the header phase only. 0: Current request is for the bypass queue 1: Current request is for the ordered queue
niu_sii_datareq	I	1	NIU->SIU		Valid during the header phase only. 0: Current request is a read, with no payload; 1: Current request is a write, with 1 or 4 cycles of data payload
niu_sii_datareq16	I	1	NIU->SIU		Valid during the header phase only. Don't care if niu_sii_datareq is 0. Otherwise should always be 0 for the supported modes expected from NIU: Current write request has 64B data payload;
niu_sii_data[127:0]	I	128	NIU->SIU		Packet header/data for L2. (Big-endian)
niu_sii_parity[7:0]	I	4	NIU->SIU		Parity of data payload cycles (127:0)
SII to NIU signals					
sii_niu_oqdq	O	1	SIU->NIU		Transaction credit for the ordered queue
sii_niu_bqdq	O	1	SIU->NIU		Transaction credit for the ordered queue
SIO to NIU signals					

TABLE 6-23 SIU-NIU Interface List *(Continued)*

sio_niu_hdr_vld	O	1	SIU->NIU		Envelops the header of any requests from SIU to NIU. Not asserted during the data transfer phase. NIU determines from the header if and how much data will follow.
sio_niu_data[127:0]	O	128	SIU->NIU		Packet header/data for NIU
sio_niu_parity[7:0]	O	4	SIU->NIU		Parity of payload cycles (127:0)

6.8.4 SIU-DMU Interface List

TABLE 6-24 SIU-DMU Interface List

Signal Name	I/O	Size	From/To	Timing	Description
DMU to SII signals					
dmu_sii_hdr_vld	I	1	DMU->SIU		Asserted during the header phase of any requests from DMU to SIU. Not asserted during the data transfer phase.
dmu_sii_reqbypass	I	1	DMU->SIU		Valid during the header phase only. Asserted for PIO rd cpl's
dmu_sii_datareq	I	1	DMU->SIU		Valid during the header phase only. 0: Current request is a read, with no payload; 1: Current request is a write, with 1 or 4 cycles of data payload
dmu_sii_datareq16	I	1	DMU->SIU		Valid during the header phase only. Don't care if dmu_sii_datareq is 0. 0: Current write request has 64B data payload; 1: Current write request has 16B data payload. (meant for NCU – int/PIO read data)
dmu_sii_data[127:0]	I	128	DMU->SIU		Packet header/data for L2/NCU. (Big-endian) For PIO read completions, the 64 bit PIO payload will be duplicated on both halves of the 128 bit data bus. Which 64 bits to replicate will be determined by dmu_sii_be[15:0]
dmu_sii_be[15:0]	I	16	DMU->SIU		Packet data byte enables/errors. Only valid during data transfer phase. (Bit position matches Byte position on the data bus.)
dmu_sii_parity[7:0]	I	4	DMU->SIU		Parity of data payload cycles (127:0)
dmu_sii_be_parity	I	1	DMU->SIU		Parity of dmu_sii_be[15:0]
SII to DMU signals					
sii_dmu_wrack_tag[3:0]	O	4	SIU->DMU		j2d_d_wrack_tag[3:0] DSN/DMU name Transaction credit id for dma wrack
sii_dmu_wrack_parity	O	1	SIU->DMU		Parity bit for the sii_dmu_wrack_tag

TABLE 6-24 SIU-DMU Interface List *(Continued)*

sii_dmu_wrack_vld	O	1	SIU->DMU		j2d_d_wrack_vld DSN/DMU name Valid signal for j2d_d_wrack_tag
SIO to DMU signals					
sio_dmu_hdr_vld	O	1	SIU->DMU		Envelops the header of any requests from SIU to DMU. Not asserted during the data transfer phase. DMU determines from the header if and how much data will follow.
sio_dmu_data[127:0]	O	128	SIU->DMU		Packet header/data for DMU
sio_dmu_parity[7:0]	O	4	SIU->DMU		Parity of payload cycles (128:0)

6.8.5 SIU-TCU Interface List

TABLE 6-25 SIU-TCU Interface List

Signal Name	I/O	Size	From/To	Timing	Description
TCU to SII					
tcu_sii_vld	I	1	TCU->SII		Valid signal to qualify the tcu_sii_data
tcu_sii_data	I	1	TCU->SII		Serial data bus for dma rd/wr request
SIO to TCU					
sio_tcu_vld	O	1	SIO->TCU		Valid signal to qualify the sio_tcu_data
sio_tcu_data	O	1	SIO->TCU		Serial bus for DMA return data/hdr
SII to DBG					
sii_dbg_l2t0_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t1_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t2_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t3_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t4_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t5_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t6_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8
sii_dbg_l2t7_req[1:0]	O	2	SII->DBG		00=no req, 01=rd,10=wr,11=wr8

CHAPTER 7

Non-Cacheable Unit (NCU)

This chapter contains the following sections:

7.1 Overview

The main function of NCU is to communicate between the CMP cores (64 threads total) and the various blocks in the IO subsystem. FIGURE 7-1 shows the connectivity of NCU with various IO subsystem blocks as well as the XBAR, which connects to CMP core on the other side. Traffic on XBAR side runs at CPU clock frequency whereas traffic on IO subsystem side is at IO clock frequency. In general, traffic goes to NCU does not require high performance and can tolerate high lateny.

FIGURE 7-1 NCU Connectivity

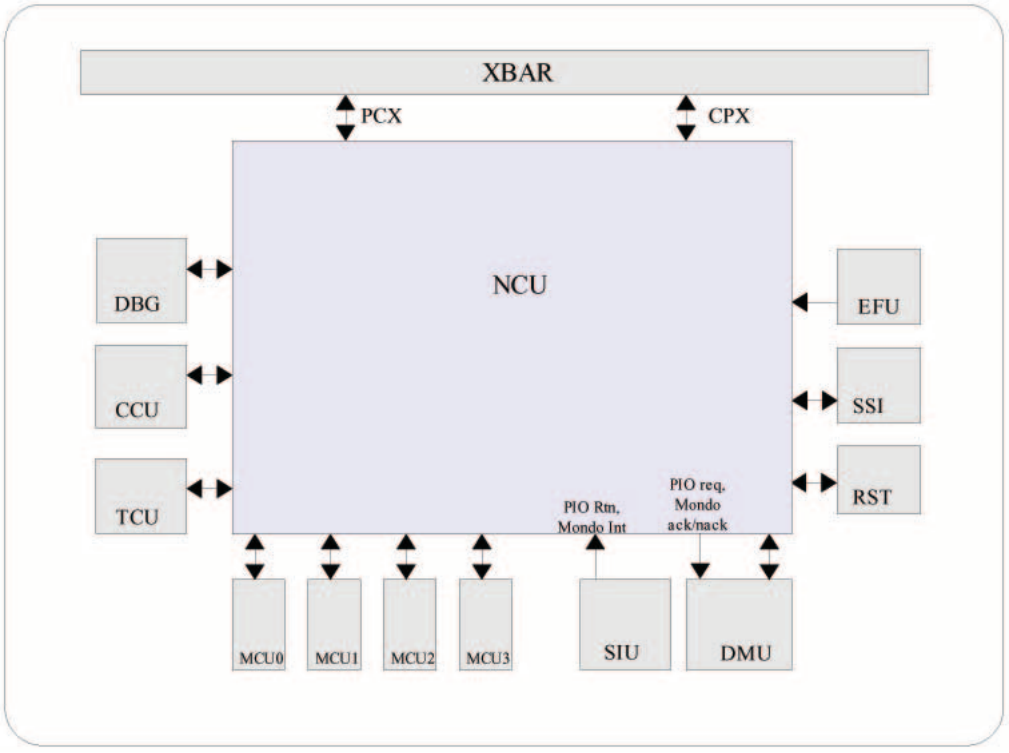


TABLE 7-1 shows a summary of types of traffic and bus size between each unit control block and NCU.

TABLE 7-1 NCU / UCB Communication Type and Bus Size

Unit Control Block (UCB)	Bus Size (downstream/ upstream (protocol))	External req/ack/intr	CSR	On Chip Interrupts	Boot Up
SII	-- / 32 (SII/NCU Table4)	Mondo intr., PIO rtns	---	---	---
DMU	Mondo resp.: 4 / -- PIO: 64 / -- (NCU/DMU 5) CSR: 32 / 32 (UCB Table2,3)	Mondo Interrupt. Resp. PIO rd_req / wr requests ---	CSR rw	---	---
MCUs	4 / 4 (UCB 2,3)	---	CSR rw	OCI	---
CCU	4 / 4 (UCB 2,3)	---	CSR rw	---	---

TABLE 7-1 NCU / UCB Communication Type and Bus Size *(Continued)*

TCU	8 / 8	(UCB 2,3)	---	CSR rw	---	---
SSI (integrated into NCU)	4 / 4	(UCB 2,3)	---	CSR rw	OCI	Instructions
RST	4 / 4	(UCB 2,3)	---	CSR rw	---	---
DBG	4 / 4	(UCB 2,3)	---	CSR rw	---	---

7.1.1 Changes from OpenSPARC T1 IOB

Changes from 2MCUs to 4MCUs.

CTU changes to CCU + TCU.

JBUS changes to SIU + DMU with different interface format

Adds DMU CSR support.

Adds DMU PIO token ID engine to limit numbers of outstanding PIO to DMU

Adds DMU PIO memory due to OpenSPARC T2 IOMMU changes

Adds support for Mondo Interrupt ID return for DMU

Adds ASI register to comply with SUN's CMP specification

Adds L2 partial bank mode support

Internal memories upsizing to accommodate 64 threads and memory pipelines adjustment

XBAR packet format changes

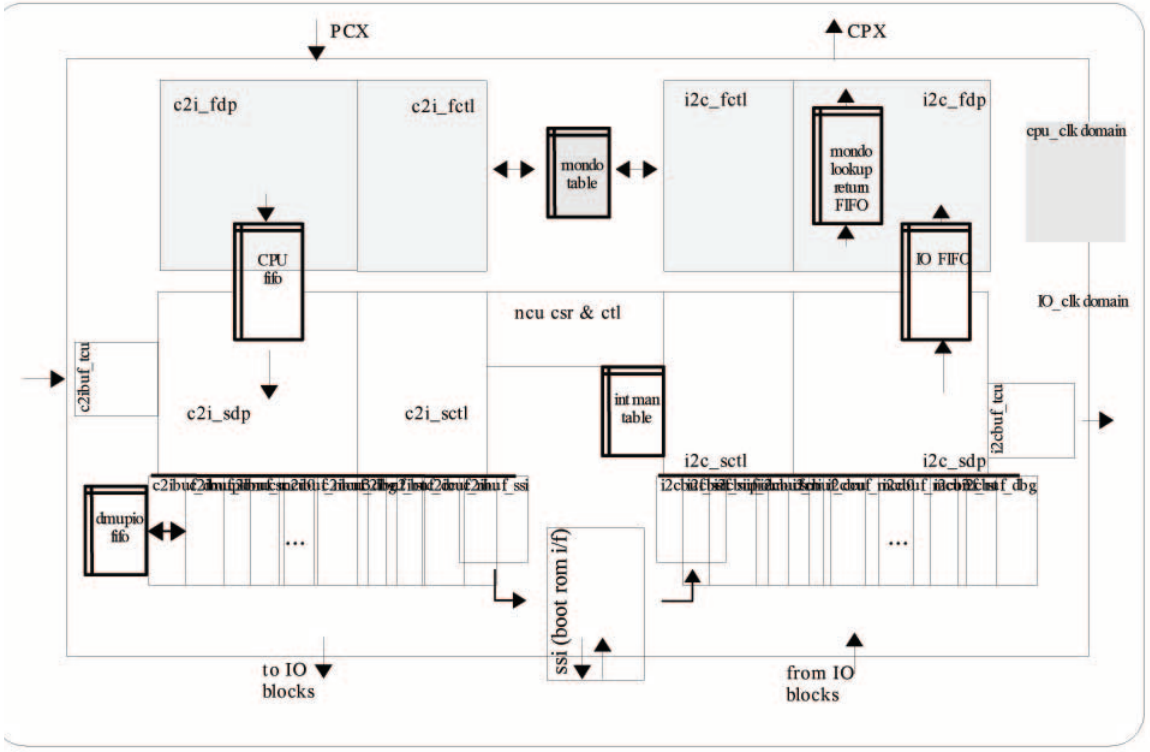
Modifies reset handling to comply with OpenSPARC T2's reset scheme

Integrates SSI (boot ROM i/f logics)

RAS logics

OpenSPARC T2 naming rule compliance.

FIGURE 7-2 NCU Internal Logical Block Diagram



NCU retains most of the internal block name from OpenSPARC T1 IOB since it does not violate the OpenSPARC T2 naming rule:

c2i – cpu to io

i2c – io to cpu

sdp – slow clock (io_clk domain) data path

sctl – slow clock control logic

fdp – fast clock (cpu_clk domain) data path

fctl – fast clock control logic's

7.2 Clock Domains

There are 2 clock domains in NCU : CPU clock domain and IO clock domain.

- CPU clock domain : XBAR side communications at CPU clock frequency (targeted for 1.4GHz)
- IO clock domain: IO subsystem side communications at IO clock frequency (targeted for 1.4GHz / 4 = 350MHz or 1.4GHz /3 = 467MHz)

7.3 Data Flow

Data Flow can be subdivided into the following categories:

Downstream -- packets / information going from XBAR to IO subsystem.

1. CPU non-cacheable external PIO store requests (8B max) /load requests (16B max).
2. CPU external instruction fetch (IFILL) requests.
3. On chip CSR read / write requests (8B only) from CPU to IO subsystem.
4. Upstream – packets / information going from IO subsystem to XBAR.
5. PIO read returns (16B max).
6. External Instruction fetch returns (4B only).
7. CSR read returns (8B only).
8. Mondo Interrupts (with 16B mondo payload).
9. On-chip interrupts (MCU, SSI).

Loopback – packet / information going from XBAR and being sent back to XBAR

1. CPU Mondo Interrupt Table lookup.
2. NCU's internal CSR / ASI register access.

Undeliverable – NCU adopted the OpenSPARC T1 IOB's packet delivery policy. All writes/STORE_REQs to NCU from core is non-posted which means core requires STORE_ACK to confirm the packets delivery. NCU generates STORE_ACK back to core automatically whenever a STORE_REQ is dequeued successfully from the main downstream FIFO or the DMUPIO fifo. Therefore, core still gets a STORE_ACK for

an undeliverable STORE_REQ, and the actual STORE_REQ packet is discarded. For undeliverable LOAD_REQ/read, NCU generates an CPX Load Return packet with uncorrectable error bit set.

7.3.1 Downstream Path Block Diagrams

FIGURE 7-3 and FIGURE 7-4 show the logical block diagram for downstream communication path, PCX interface is a 130 bit-wide data bus running at 1.5GHz., sourcing packets into NCU. The packet is then decoded by c2i_fdp and c2i_fctl blocks to extract the CPU Mondo interrupt table access from other traffic, which includes on-chip CSR read / write access, CPU non-cacheable external load / store request (PIO read / write), and CPU instruction fetch request. All requests other than Mondo interrupt table access are sent to the CPU command FIFO, which is a 32 deep domain crossing FIFO. The write control is managed by c2i_fctl block in CPU clock domain and the read control is managed by c2i_fctl block in IO clock domain. When a CPU packet is dequeued from the FIFO, c2i_sdp and c2i_sctl blocks will determine which of UCB output buffers to send the packet to. Each of the UCB output buffer contains a double buffer and a working buffer as shown in FIGURE 7-4.

FIGURE 7-3 Downstream Path Logic Block Diagram

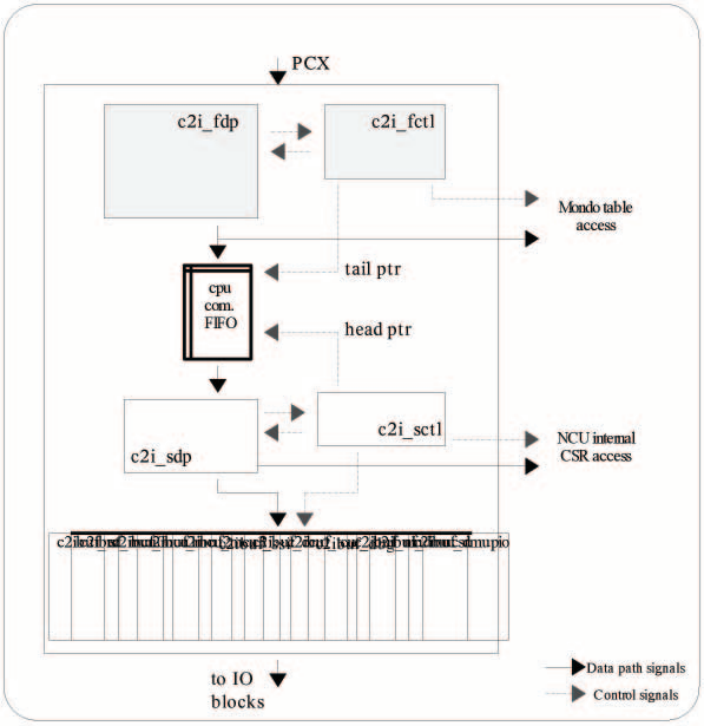
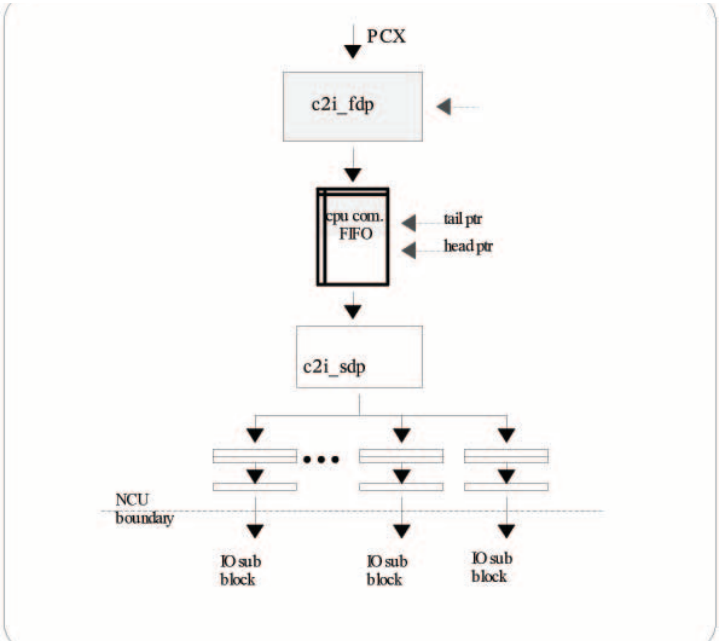


FIGURE 7-4 Downstream Data Path Block Diagram



Note that there is no downstream communications to SIU because all non cacheable external CPU load / store requests (PIO requests) are sent directly to DMU, and there is no CSR in SIU neither. However, SIU talks to NCU on the upstream data return path, and this will be covered in the upstream path.

7.3.2 Upstream Path Block Diagrams

FIGURE 7-5 and FIGURE 7-6 show the logical block diagram for the upstream communication path. NCU collects packets from each IO block and push them into the upstream main FIFO in IO clock domain. For Mondo Interrupt case (originated from DMU and sent via SIU to NCU), NCU checks the internal status table for the target CPU thread's availability and responses with an “ack” or “nack,” An “ack” means the Mondo Interrupt is accepted, and a “nack” means it is rejected. The upstream main FIFO is a 32 deep domain crossing FIFO, shared by all IO blocks. Data is written in IO clock domain and read out in CPU clock domain. The head pointer and tail pointer of the FIFO are controlled by the i2c_sctl block in IO clock domain and i2c_fctl block in CPU clock domain, respectively. C2i_fdp and c2i_fctl blocks also arbitrate and mux between the upstream main FIFO output and CPU Mondo lookup data output as shown in FIGURE 7-5 and FIGURE 7-6.

FIGURE 7-5 Upstream Path Logic Block Diagram

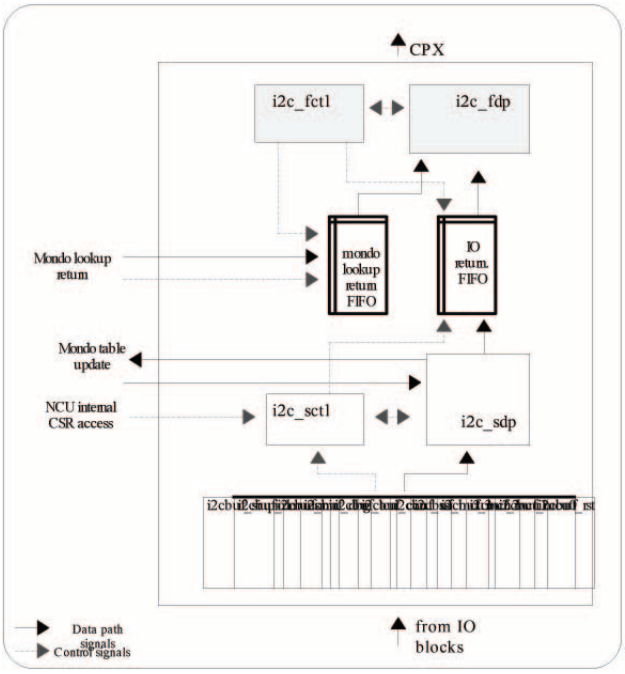
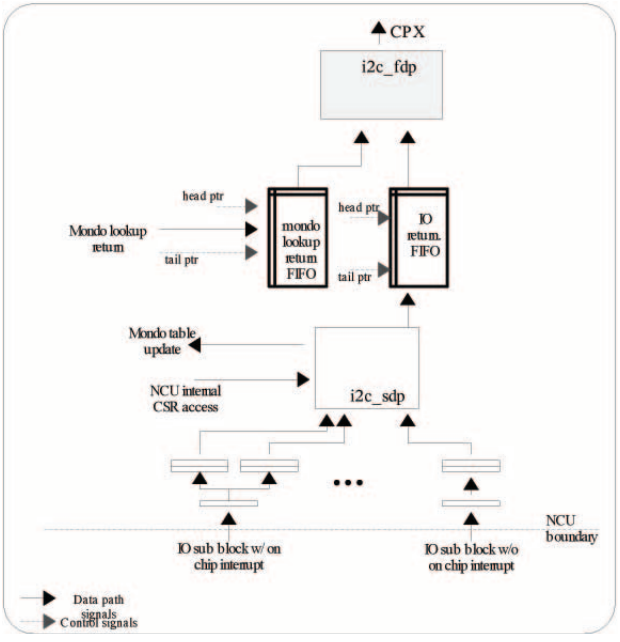


FIGURE 7-6 Upstream Data Path Block Diagram



7.4 Interface Signals, Protocols, and Timing Diagrams

TABLE 7-2 NCU / XBAR(CCX)IInterface Signals

NCU / XBAR(CCX) Interface Signals	Direction	Comment
ncu_pcx_stall_pq	Output	NCU back pressure control signal to CCX
pcx_ncu_data_rdy_px1	Input	PCX to NCU data valid (px1 version)
pcx_ncu_data_px2[129:0]	Input	PCX to NCU data bus (px2 version)
cpx_ncu_grant_cx[7:0]	Input	CPX grant indicates the corresponding packet has reached its CPU destination and there is room for more packet for the same corresponding CPU destination.
ncu_cpx_req_cq[7:0]	Output	NCU to CPX request Signals
ncu_cpx_data_ca[144:0]	Output	NCU to CPX data bus

TABLE 7-3 NCU / MCU0 Interface Signals

NCU / MCU0 Interface Signals	Direction	Comment
mcu0_ncu_stall	Input	MCU0 back pressure control signal to NCU
ncu_mcu0_vld	Output	NCU to MCU0 data valid
ncu_mcu0_data[3:0]	Output	NCU to MCU0 data bus
ncu_mcu0_stall	Output	NCU back pressure control signal to MCU0
mcu0_ncu_vld	Input	MCU0 to NCU data valid
mcu0_ncu_data[3:0]	Input	MCU0 to NCU data bus
mcu0_ncu_e0	input	Error strobe0 from MCU0
ncu_mcu0_e0i	output	Error injection0 to MCU0
mcu0_ncu_e1	input	Error strobe1 from MCU0

TABLE 7-3 NCU / MCU0 Interface Signals

ncu_mcu0_e1i	output	Error injection1 to MCU0
mcu0_ncu_e2	input	Error strobe2 from MCU0
ncu_mcu0_e2i	output	Error injection2 to MCU0

TABLE 7-4 NCU / MCU1 Interface Signals

NCU / MCU1 Interface Signals	Direction	Comment
mcu1_ncu_stall	Input	MCU1 back pressure control signal to NCU
ncu_mcu1_vld	Output	NCU to MCU1 data valid
ncu_mcu1_data[3:0]	Output	NCU to MCU1 data bus
ncu_mcu1_stall	Output	NCU back pressure control signal to MCU1
mcu1_ncu_vld	Input	MCU1 to NCU data valid
mcu1_ncu_data[3:0]	Input	MCU1 to NCU data bus
mcu1_ncu_e0	input	Error strobe0 from MCU1
ncu_mcu1_e0i	output	Error injection0 to MCU1
mcu1_ncu_e1	input	Error strobe1 from MCU1
ncu_mcu1_e1i	output	Error injection1 to MCU1
mcu1_ncu_e2	input	Error strobe2 from MCU1
ncu_mcu1_e2i	output	Error injection2 to MCU1

TABLE 7-5 NCU / MCU2 Interface Signals

NCU / MCU2 Interface Signals	Direction	Comment
mcu2_ncu_stall	Input	MCU2 back pressure control signal to NCU
ncu_mcu2_vld	Output	NCU to MCU2 data valid
ncu_mcu2_data[3:0]	Output	NCU to MCU2 data bus
ncu_mcu2_stall	Output	NCU back pressure control signal to MCU2

TABLE 7-5 NCU / MCU2 Interface Signals

mcu2_ncu_vld	Input	MCU2 to NCU data valid
mcu2_ncu_data[3:0]	Input	MCU2 to NCU data bus
mcu2_ncu_e0	input	Error strobe0 from MCU2
ncu_mcu2_e0i	output	Error injection0 to MCU2
mcu2_ncu_e1	input	Error strobe1 from MCU2
ncu_mcu2_e1i	output	Error injection1 to MCU2
mcu2_ncu_e2	input	Error strobe2 from MCU2
ncu_mcu2_e2i	output	Error injection2 to MCU2

TABLE 7-6 NCU / MCU3 Interface Signals

NCU / MCU3 Interface Signals	Direction	Comment
mcu3_ncu_stall	Input	MCU3 back pressure control signal to NCU
ncu_mcu3_vld	Output	NCU to MCU3 data valid
ncu_mcu3_data[3:0]	Output	NCU to MCU3 data bus
ncu_mcu3_stall	Output	NCU back pressure control signal to MCU3
mcu3_ncu_vld	Input	MCU3 to NCU data valid
mcu3_ncu_data[3:0]	Input	MCU3 to NCU data bus
mcu3_ncu_e0	input	Error strobe0 from MCU3
ncu_mcu3_e0i	output	Error injection0 to MCU3
mcu3_ncu_e1	input	Error strobe1 from MCU3
ncu_mcu3_e1i	output	Error injection1 to MCU3
mcu3_ncu_e2	input	Error strobe2 from MCU3
ncu_mcu3_e2i	output	Error injection2 to MCU3

TABLE 7-7 NCU / SSI Interface Signals

NCU / SSI Interface Signals	Direction	Comment
ncu_mio_ssi_sck	Output	Boot ROM interface clk (iol2clk/8)
ncu_mio_ssi_mosi	Output	Boot ROM interface data ssi output to ROM
mio_ncu_ssi_miso	input	Boot ROM interface data ROM to ssi
mio_ncu_ssi_ext_int_l	input	Low active external trigger interupt

TABLE 7-8 NCU / DBG1 Interface Signals

NCU / DBG1 Interface Signals	Direction	Comment
dbg1_ncu_stall	Input	DBG1 back pressure control signal to NCU
dbg1_ncu_vld	Input	DBG1 to NCU data valid
dbg1_ncu_data[3:0]	Input	DBG1 to NCU data bus
ncu_dbg1_vld	Output	NCU to DBG1 data valid
ncu_dbg1_data[3:0]	Output	NCU to DBG1 data bus
ncu_dbg1_stall	Output	NCU back pressure control signal to DBG1
ncu_dbg1_error_event	Output	NCU error happens, enabled with wmr_vec_mask

TABLE 7-9 NCU / CCU Interface Signals

NCU / CCU Interface Signals	Direction	Comment
ccu_ncu_stall	Input	CCU back pressure control signal to NCU
ncu_ccu_vld	Output	NCU to CCU data valid
ncu_ccu_data[3:0]	Output	NCU to CCU data bus
ncu_ccu_stall	Output	NCU back pressure control signal to CCU
ccu_ncu_vld	Input	CCU to NCU data valid
ccu_ncu_data[3:0]	Input	CCU to NCU data bus

TABLE 7-10 NCU / TCU Interface Signals

NCU / TCU Interface Signals	Direction	Comment
tcu_ncu_stall	Input	TCU back pressure control signal to NCU
tcu_ncu_vld	Input	TCU to NCU data valid
tcu_ncu_data[7:0]	Input	TCU to NCU data bus
ncu_tcu_stall	Output	NCU back pressure control signal to TCU
ncu_tcu_vld	output	NCU to TCU data valid.
ncu_tcu_data[7:0]	Output	NCU to TCU data bus.
ncu_tcu_soc_error	Output	1 pulse signal to TCU each time when an soc error packet is generated from NCU to the core
ncu_tcu_bank_avail[7:0]	output	Copy from bankavail[7:0].
tcu_ncu_mbist_start[1:0]	Input	Mbist start (1'b0 for normal function mode)
ncu_tcu_mbist_done[1:0]	Output	Mbist done
ncu_tcu_mbist_fail[1:0]	Output	Mbist fail
tcu_dbr_gateoff	Input	Turn off all the vld and stall when it is 1'b1.

TABLE 7-11 NCU / RST Interface Signals

NCU / RST Interface Signals	Direction	Comment
rst_ncu_stall	Input	RST back pressure control signal to NCU
ncu_rst_vld	Output	NCU to RST data valid
ncu_rst_data[3:0]	Output	NCU to RST data bus
ncu_rst_stall	Output	NCU back pressure control signal to RST
rst_ncu_vld	Input	RST to NCU data valid
rst_ncu_data[3:0]	Input	RST to NCU data bus

TABLE 7-11 NCU / RST Interface Signals

rst_ncu_unpark_thread	Input	After each "warm reset" is de-asserted and all BISX activities are completed, RST send in a 1-clock wide pulse to tell NCU the system is ready to wake up the master thread, which is the lowest available thread basing on core_enable_status ASI register.
rst_ncu_xir_	Input	External Initiated Interrupt (multi-clock wide pulse signal) This signal triggers interrupts to cpu_thr that based on XIR_steering register. It will be deasserted when ncu_rst_xir_done is high.
ncu_rst_xir_done	Output	NCU asserts this signal back to RST block to indicate all XIR interrupts have been generated (multi-clock wide pulse). It will be deasserted when rst_ncu_xir_ is back to high.

TABLE 7-12 NCU / DMU CSR Interface Signals

NCU / DMU CSR Interface Signals	Direction	Comment This section is currently under definition
dmu_ncu_stall	Input	DMU CSR bus back pressure control signal to NCU.
ncu_dmu_vld	Output	NCU to DMU CSR data valid.
ncu_dmu_data[31:0]	Output	NCU to DMU CSR data bus.
ncu_dmu_stall	Output	NCU CSR bus back pressure control signal to DMU.
dmu_ncu_vld	Input	DMU to NCU CSR data valid.
dmu_ncu_data[31:0]	Input	DMU to NCU CSR data bus.

TABLE 7-13 NCU / DMU PIO and Mondo Interface

NCU / DMU PIO and Mondo Interface	Direction	Comment
ncu_dmu_pio_hdr_vld	Output	Indicates ncu_dmupio_data is valid for PIO header transaction.
ncu_dmu_mmu_addr_vld	Output	Indicates ncu_dmupio_data is valid for 1 cycle for "mmu invalidate vector." The vector is coming a write operation into CSR register 0x80_0000_2030
ncu_dmu_pio_data[63:0]	Output	NCU to DMU data bus

TABLE 7-13 NCU / DMU PIO and Mondo Interface

dmu_ncu_wrack_par	input	Odd parity check for dmu_ncu_wrack_tag[3:0].
dmu_ncu_wrack_vld	Input	Indicates dmu_ncu_wrack_tag[3:0] is valid
dmu_ncu_wrack_tag[3:0]	Input	Credit ID back to NCU for PIO write completion.
ncu_dmu_mondo_ack	Output	Mondo Interrupt ack (ncu_dmu_mondo_id[5:0] is valid when this signal is asserted to indicate the mondo_id it is acking.)
ncu_dmu_mondo_nack	Output	Mondo Interrupt nack (ncu_dmu_mondo_id[5:0] is valid when this signal is asserted to indicate the mondo_id it is nacking.)
ncu_dmu_mondo_id[5:0]	Output	Mondo Interrupt ID, valid when ncu_dmu_mondo_ack or ncu_dmu_mondo_nack is asserted.
dmu_ncu_ctag_ue	Input	Ctag double bit ue from SIO DMA read return.
ncu_dmu_ctag_uei	Output	Ctag double bit ue injected by RASEJR.
dmu_ncu_ctag_ce	Input	Error strobe from dmu (ctag ue)
ncu_dmu_ctag_cei	Output	Error injection signal
dmu_ncu_d_pe	Input	Error strobe from dmu (data parity error)
ncu_dmu_d_pei	Output	Error injection signal
dmu_ncu_siicr_pe	Input	Error strobe from dmu (siicr parity error)
ncu_dmu_siicr_pei	Output	Error injection signal
dmu_ncu_ncucr_pe	Input	Error strobe from dmu (ncucr parity error)
ncu_dmu_ncucr_pei	Output	Error injection signal
dmu_ncu_ie	Input	Error strobe from dmu (internal error)
ncu_dmu_iei	Output	Error injection signal

TABLE 7-14 NCU / SII Interface Signals

NCU / SII Interface Signals	Direction	Comment
ncu_sii_gnt	Output	NCU to SII grant signal to indicate there is room to receive one more packet. Transaction should starts in the next cycle
sii_ncu_req	Input	SII to NCU packet available request
sii_ncu_data[31:0]	Input	SII to NCU data bus

TABLE 7-14 NCU / SII Interface Signals *(Continued)*

sii_ncu_dparity[1:0]	input	SII to NCU data parity (covers sii_ncu_data[31:0] bus for data cycle only). Odd parity: bit[0] covers even bits, and bit[1] covers odd bits.
sii_ncu_dmua_ue	input	Error strobe from sii (dmu pkt address ue)
ncu_sii_dmua_uei	output	Error injection signal
sii_ncu_dmuctag_ue	input	Error strobe from sii (dmu pkt ctag ue)
ncu_sii_dmuctag_uei	output	Error injection signal
sii_ncu_dmuctag_ce	input	Error strobe from sii (dmu pkt ctag ce)
ncu_sii_dmuctag_cei	output	Error injection signal
sii_ncu_dmud_pe	input	Error strobe from sii (dmu pkt data parity error)
ncu_sii_dmud_pei	output	Error injection signal
sii_ncu_niua_ue	input	Error strobe from sii (niu pkt address ue)
ncu_sii_niua_uei	output	Error injection signal
sii_ncu_niuctag_ue	input	Error strobe from sii (niu pkt ctag ue)
ncu_sii_niuctag_uei	output	Error injection signal
sii_ncu_niuctag_ce	input	Error strobe from sii (niu pkt ctag ue)
ncu_sii_niuctag_cei	output	Error injection signal
sii_ncu_niud_pe	input	Error strobe from sii (niu pkt data parity ue)
ncu_sii_niud_pei	output	Error injection signal
sii_ncu_syn_vld	input	Error syndrome vld for sii_ncu_syn_data[3:0]
sii_ncu_syn_data[3:0]	output	Error syndrome data bus for
ncu_sii_pm	Output	L2 bank partial mode. (Value is from BANK_ENABLE_STATUS register)
ncu_sii_ba01	Output	L2 bank0,1 available (Value is from BANK_ENABLE_STATUS register)
ncu_sii_ba23	Output	L2 bank2,3 available (Value is from BANK_ENABLE_STATUS register)

TABLE 7-14 NCU / SII Interface Signals *(Continued)*

ncu_sii_ba45	Output	L2 bank4,5 available (Value is from BANK_ENABLE_STATUS register)
ncu_sii_ba67	Output	L2 bank6,7 available (Value is from BANK_ENABLE_STATUS register)
ncu_sii_l2_idx_hash_en	Output	L2 index hash enable. (Value is from L2_IDX_HASH_EN_STATUS register)

TABLE 7-15 SIO/NCU Interface Signals

SIO/NCU Interface Signals	Direction	Comment
sio_ncu_ctag_ue	input	Error strobe from sio (ctag ue)
ncu_sio_ctag_uei	output	Error injection signal
sio_ncu_ctag_ce	input	Error strobe from sio (ctag ce)
ncu_sio_ctag_cei	output	Error injection signal
sio_ncu_d_pe	input	Error strobe from sio (data parity error)
ncu_sio_d_pei	output	Error injection signal

TABLE 7-16 EFUSE / NCU Interface Signals

EFUSE / NCU Interface Signals	Direction	Comment
efu_ncu_fuse_data	Input	Fuse unit serial data signal
efu_ncu_coreavl_xfer_en	Input	Indicates data bit is valid for core available register.
efu_ncu_bankavl_xfer_en	Input	Indicates data bit is valid for bank available register.
efu_ncu_fusestat_xfer_en	Input	Indicates data bit is valid for fusestatus reg.
efu_ncu_sernum0_xfer_en	Input	Indicates data bit is valid for sernum0 reg.
efu_ncu_sernum1_xfer_en	Input	Indicates data bit is valid for sernum1 reg.
efu_ncu_sernum2_xfer_en	Input	Indicates data bit is valid for sernum2 reg.

TABLE 7-17 CCU / NCU Interface Signals

CCU / NCU Interface Signals	Direction	Comment
ccu_cmp_io_sync_en	Input	Sync. pulse for cmp domain to io domain
ccu_io_cmp_sync_en	Input	Sync. pulse for io domain to cmp domain
tcu_pce_ov	Input	TEST control. 1'b0 for normal functional mode
tcu_ncu_clk_stop	Input	TEST control. 1'b0 for normal functional mode
tcu_ncu_io_stop	Input	TEST control. 1'b0 for normal functional mode
ccu_io_out	Input	CCU output goes to NCU io clkgen.
tcu_aclk	Input	SCAN clock
tcu_bclk	Input	SCAN clock

TABLE 7-18 Global Signals

Global Signals	Direction	Comment
scan_in	Input	SCAN IN (1'b0 or 1'b1 for normal function mode simulation)
scan_out	Output	SCAN OUT
tcu_ncu_mbist_scan_in	Input	Mbist scan in (1'b0 or 1'b1 for normal function mode simulation)
ncu_tcu_mbist_scan_out	Output	Mbist scan out
tcu_mbist_bisi_en	Input	Bist engine enable (1'b0 for normal function mode)
tcu_scan_en	Input	SCAN enable. 1'b0 for normal functional mode
tcu_se_scancollar_in	Input	TEST control. 1'b0 for normal functional mode
tcu_se_scancollar_out	Input	TEST control. 1'b0 for normal functional mode
tcu_array_wr_inhibit	Input	TEST control. 1'b0 for normal functional mode

TABLE 7-19 Signals to L2T

Signals to L2T	Direction	Comment
ncu_l2t_pm	Output	L2 bank partial mode. (Value is from BANK_ENABLE_STATUS register)
ncu_l2t_ba01	Output	L2 bank0,1 available (Value is from BANK_ENABLE_STATUS register)
Ncu_l2t_ba23	Output	L2 bank2,3 available (Value is from BANK_ENABLE_STATUS register)
ncu_l2t_ba45	Output	L2 bank4,5 available (Value is from BANK_ENABLE_STATUS register)
ncu_l2t_ba67	Output	L2 bank6,7 available (Value is from BANK_ENABLE_STATUS register)

TABLE 7-20 Signals to all SPC

Signals to all SPC	Direction	Comment
ncu_spc_pm	Output	L2 bank partial mode. (Value is from BANK_ENABLE_STATUS register)
ncu_spc_ba01	Output	L2 bank0,1 available (Value is from BANK_ENABLE_STATUS register)
ncu_spc_ba23	Output	L2 bank2,3 available (Value is from BANK_ENABLE_STATUS register)
ncu_spc_ba45	Output	L2 bank4,5 available (Value is from BANK_ENABLE_STATUS register)
ncu_spc_ba67	Output	L2 bank6,7 available (Value is from BANK_ENABLE_STATUS register)
ncu_spc_l2_idx_hash_en	Output	L2 index hash enable. (Value is from L2_IDX_HASH_EN_STATUS register)
cmp_tick_enable	output	ASI register cmp_tick_enable signal.
tcu_wmr_vec_mask	output	ASI register wmr_vec_mask signal.

TABLE 7-21 SPC 0

SPC 0	Direction	Comment
ncu_spc0_core_enable_status	Output	For gating off clock to SPC0
ncu_spc0_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC0 for each thread
spc0_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC0 status is active or parking.

TABLE 7-22 SPC1

SPC1	Direction	Comment
ncu_spc1_core_enable_status	Output	For gating off clock to SPC1
ncu_spc1_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC1 for each thread
spc0_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC1 status is active or parking.

TABLE 7-23 SPC2

SPC2	Direction	Comment
ncu_spc2_core_enable_status	Output	For gating off clock to SPC2
ncu_spc2_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC2 for each thread
spc2_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC2 status is active or parking.

TABLE 7-24 SPC3

SPC3	Direction	Comment

TABLE 7-24 SPC3 *(Continued)*

ncu_spc3_core_enable_status	Output	For gating off clock to SPC3
ncu_spc3_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC3 for each thread
spc3_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC3 status is active or parking.

TABLE 7-25 SPC4

SPC4	Direction	Comment
ncu_spc4_core_enable_status	Output	For gating off clock to SPC4
ncu_spc4_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC4 for each thread
spc4_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC4 status is active or parking.

TABLE 7-26 SPC5

SPC5	Direction	Comment
ncu_spc5_core_enable_status	Output	For gating off clock to SPC5
ncu_spc5_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC5 for each thread
spc5_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC5 status is active or parking.

TABLE 7-27 SPC6

SPC6	Direction	Comment
ncu_spc6_core_enable_status	Output	For gating off clock to SPC6
ncu_spc6_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC6 for each thread
spc6_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC6 status is active or parking.

TABLE 7-28 SPC7

SPC7	Direction	Comment
ncu_spc7_core_enable_status	Output	For gating off clock to SPC7
ncu_spc7_core_running[7:0]	Output	8-bit Signals to indicate parking or unparking request to SPC7 for each thread
spc7_ncu_core_running_status[7:0]	Input	8-bit Signals to indicate the current SPC7 status is active or parking.

7.4.1 XBAR Interface

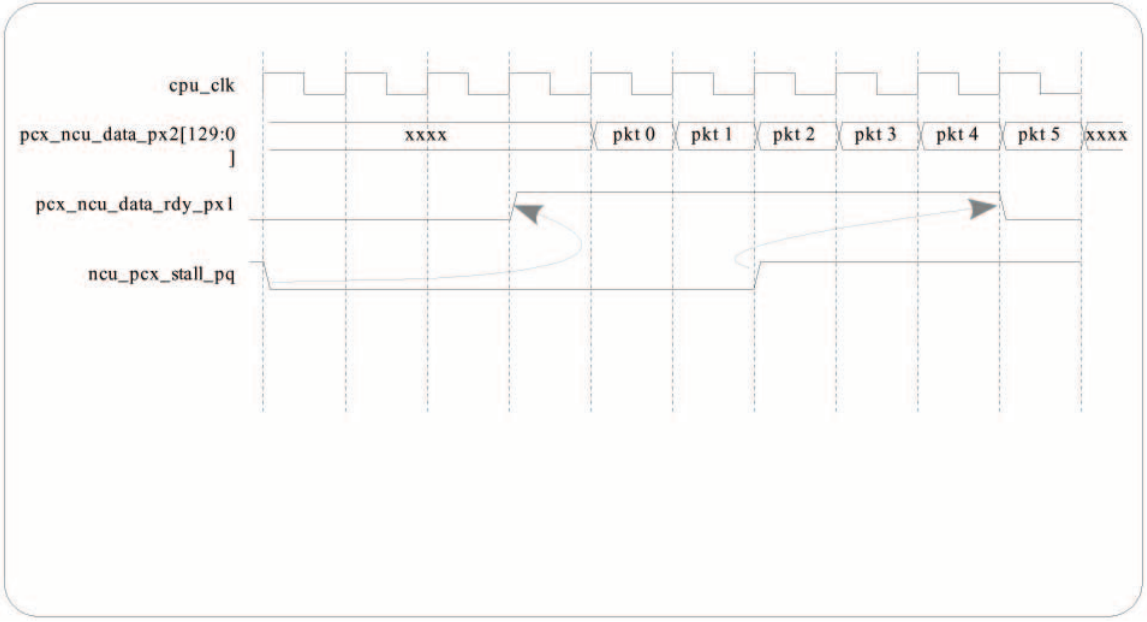
7.4.1.1 NCU / XBAR PCX Interface (Downstream)

The PCX interface is a 130 bit-wide bus with 2 bits flow control Signals. The signal "ncu_pcx_stall_pq" is to back pressure XBAR when the downstream CPU shared buffer becomes full. An asserted "pcx_ncu_data_rdy_px1" indicates the data bus "pcx_ncu_data_px2[129:0]" to NCU is valid in next cycle. FIGURE 7-7 shows the case that CPU shared buffer has available entry and is filled by PCX packets again. The de-assertion of "ncu_pcx_stall_pq" indicates there is room for at least 6 more PCX packets. Due to the nature of pipelining design, there may be three more packets in flight after the assertion of "ncu_pcx_stall_pq" signal. Therefore, when "ncu_pcx_stall_pq" signal is asserted, NCU is guaranteed to be able to accept as least three more packets from XBAR PCX interface as shown in FIGURE 7-7. For details of PCX or CPX packet format, please reference to "crossbar packet definition" on OpenSPARC T2's web page.

Note that NCU is using px1 version of "data ready" signal and px2 version for "data" bus.

Note – For PCX / CPX packet format, please see "Crossbar packet definition" on OpenSPARC T2 web page.

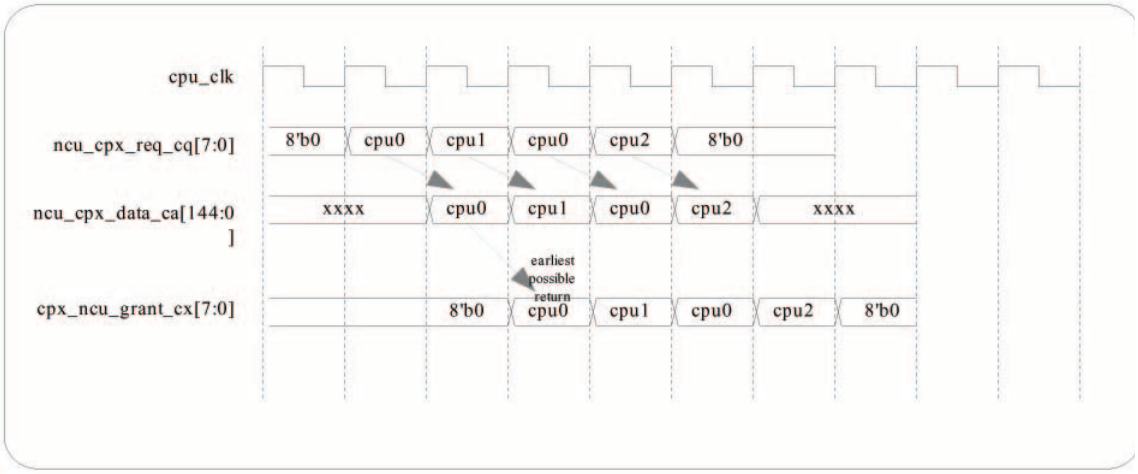
FIGURE 7-7 Downstream PCX Interface Timing



7.4.1.2 NCU / XBAR CPX Interface (Upstream)

The CPX interface is a 146 bit-wide data bus plus 2 sets of 8 bit-wide flow control. NCU keeps track of the number outstanding requests without grant for each of the 8 CPU. When the number of outstanding requests without grant reaches 2 for a particular CPU, NCU will stop sending the 3rd request to the same CPU until the 1st grant has returned. This is mainly due to CCX has 2 levels of buffering for each CPU destination, and sending a 3rd outstanding packet to the same destination will result in packet being lost. The Timing diagram for the CPX bus is shown in FIGURE 7-8.

FIGURE 7-8 Upstream PCX Interface Timing



7.4.2 NCU / MCU Interface

There are four MCUs on OpenSPARC T2, and they are all connected to NCU in the same manner. The downstream and upstream paths are both 4 bit-wide data bus with 2 control Signals. The interface protocol is a 128 bit packet being spread into 32 cycles of transactions. NCU only sends type “READ_REQ,” and “WRITE_REQ.” with 8B request size to MCU for CSR access. The packet types that MCU sends upstream to NCU are:

“READ_ACK,” with 8B payload in response to a successful “READ_REQ,” (128-bit UCB packet)

“READ_NACK,” without payload in response to an unsuccessful “READ_REQ,” (64-bit UCB packet without payload)

“INT,” for on chip interrupt, resulting from some error conditions in MCU (64-bit UCB Int. packet with dev_id = 1)

FIGURE 7-9 and FIGURE 7-10 show the downstream and upstream timing diagram for NCU / MCU interface.

FIGURE 7-9 NCU to MCU/SSI/RNG/CCU/RST Downstream Timing Diagram (back-to-back case)

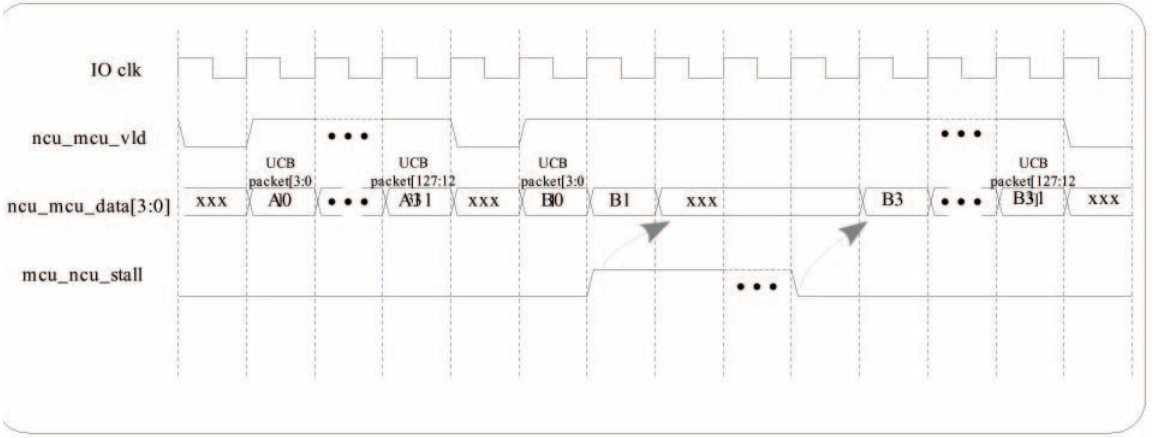
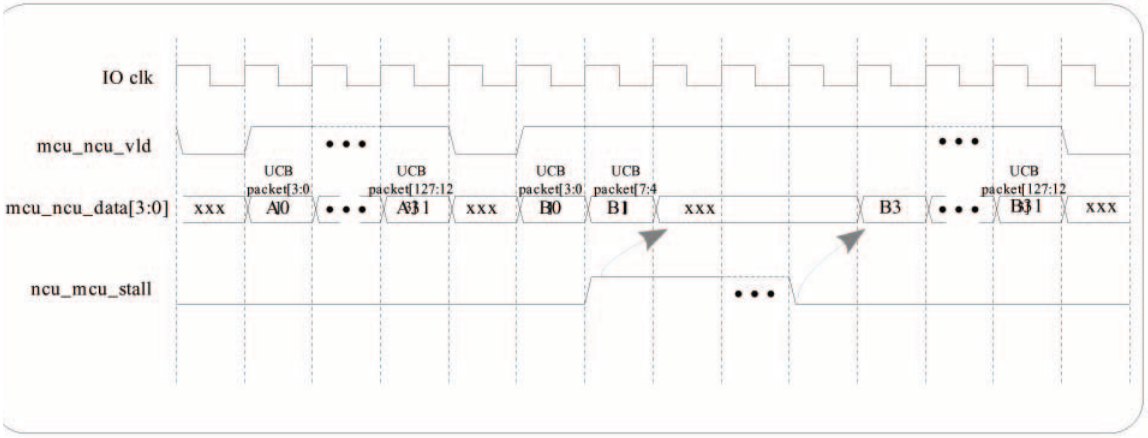


FIGURE 7-10 MCU/SSI/RNG/CCU/RST to MCU Upstream Timing Diagram (back-to-back read ack return of IFIKK return with 8B payload))



7.4.3 Boot ROM Interface (NCU/SSI))

NCU has integrated the SSI interface logics which originated from OpenSPARC T1. With modification, the ncu_mio_ssi_sck frequency is now programmable. It could be iol2clk/8 (default) or iol2clk/4. There are 4 i/o pins directly connecting to the external. Please reference to appendix regarding the boot ROM interface. The original SSI UCB interface has become NCU's internal Signals and is no longer visible from outside of NCU cluster. If a request from CPU is an IFILL request, but the 40-bit PA is not addressed to SSI (0xFF_Fxxxx_xxxx), NCU/SSI classifies this as an un-deliverable packet, and will reply with IFILL return to CPU with uncorrectable error set. When CPU initiates an IFILL request to NCU, CPU expects only 4 bytes IFILL load return. For IFILL request, CPU should only request 4Byte. The F4B field in CPX packet should always be set to “1”. However, for SSI's CSR request, CPU should always do 8-byte access similar to all other CSR access. SSI also supports non-IFILL and non CSR type access, which read or write to external. For this type of requests, SSI supports 1,2,4,8 bytes access.

ncu_mio_ssi_sck could be programmed as iol2clk/8 or iol2clk/4, depends on the CSR register NCU_SCKSEL. This register is warm_reset protected. The new value programmed into NCU_SCKSEL register, can't effects current ncu_mio_ssi_sck until next warm reset.

After warm reset, NCU holds up to 5 ms before sending 1st request. This is the time FPGA needs to lock sck clock. However, during test and debug mode TCU can drive tcu_sck_bypass signal to "1", and this will cause NCU/SSI to skip the 5ms wait. System developer should make sure the external boot ROM interface logic is stabilized and ready before SSI sends out the first request.

7.4.4 NCU / CCU Interface

The CCU interface is same as the NCU/MCU interface in UCB packet format. The request size is always 8B and the request types that NCU sends downstream to CCU are "READ_REQ," and "WRITE_REQ." CCU returns the following to NCU:

- "READ_ACK, "with 64 bit payload in response to a successful CSR "READ_REQ;"
- "READ_NACK," without payload in response to an unsuccessful CSR "READ_REQ."

The interface timing diagram for NCU/CCU is same as NCU/MCU, which can be found in FIGURE 7-9 and FIGURE 7-10.

7.4.5 NCU / RST Interface

The RST interface is same as the NCU/MCU interface in UCB packet format. The request size is always 8B and the request types that NCU sends downstream to RST are "READ_REQ," and "WRITE_REQ." RST returns the following to NCU:

- "READ_ACK,"with 64 bit payload in response to a successful CSR "READ_REQ;"
- "READ_NACK," without payload in response to an unsuccessful CSR "READ_REQ."

The interface timing diagram for NCU/RST is same as NCU/MCU, which can be found in FIGURE 7-9 and FIGURE 7-10.

7.4.6 NCU / DMUCSR Interface

The DMUCSR Interface is same as the NCU interface in UCB packet format. The request size is always 8B, and the request types that NCU sends downstream to DMUCSR interface are "READ_REQ," and "WRITE_REQ." DMUCSR interface returns the following to NCU:

- "READ_ACK,"with 64 bit payload in response to a successful CSR "READ_REQ;"
- "READ_NACK," without payload in response to an unsuccessful CSR "READ_REQ."

The interface timing diagram for NCU/DMUCSR is same as NCU/NIU, which can be found in FIGURE 7-11 and FIGURE 7-12.

7.4.7 NCU / DBG Interface

NCU/DBG interface is similar to NCU/MCU interface. The downstream and upstream paths are both 4 bit-wide data bus with 2 control Signals. The interface protocol is a 128 bit packet being spread into 32 cycles of transactions. NCU only sends type "READ_REQ," and "WRITE_REQ." with 8B request size to DBG for CSR access. The packet types that DBG sends upstream to NCU are:

- "READ_ACK," with 8B payload in response to a successful "READ_REQ," (128-bit UCB packet)
- "READ_NACK," without payload in response to an unsuccessful "READ_REQ," (64-bit UCB packet without payload)

Please refers to NCU/MCU downstream and upstream timing diagrams for NCU / DBG interface.

7.4.8 NCU / TCU Interface

The TCU interface is similar to NCU/MCU or NCU interfaces using UCB packet format except it is an 8-bit data bus plus 2 control Signals each way. For write_req, read_req, and read_ack type, the packet size is 128-bit and requires 16 cycles to complete the transaction. For read_nack type, the packet size is 64-bit, and requires 8 cycles to complete the transaction. TCU is intended to connect to the external service processor (JTAG / TAP controller,) and, therefore, is capable to initiate request type UCB packet into NCU. TCU can be in both master or slave mode.

NCU in Niagra2 does not support JTAG/TAP access across the Crossbar to L2s nor CPUs. Therefore, TCU is limited to access the following via NCU: NCU's CSR, MCUs' CSR, NIU's CSR/PIO, SSI's CSR, DMU's CSR+PIO, RST's CSR, CCU's CSR.

The request type UCB packet from TCU to NCU should contain the following fields: Buffer ID (always 2'b01), a valid 40-bit PA field, a valid Packet Type and a Request Size field .On the return path, when an UCB packet returned to NCU with the Buffer ID field marked as TAP packet, NCU routes the packet back to TCU accordingly. All write requests from TCU are "posted," which means no "ack" is generated back to TCU after a write request, and the packet will be dropped silently if address is illegal. This implies TCU can generated multiple consecutive write requests (possibly back-to-back) in a short period of time because it does not require "ack" for a write request. However, once TCU generates a read requests to NCU, there should not be any more requests until a "READ_ACK" or "READ_NACK" UCB packet has returned back to TCU. NCU does not support Interrupt type packet nor IFILL type packet on NCU/TCU interface.

TCU / NCU interface is designed as a low performance, infrequent access interface. Unlike other interface, NCU provides only minimum buffering for TCU accessing. Excessive traffic from TCU can possibly slow down performance of NCU due to lack of buffering issue.

TABLE 7-29 UCB Packet Types supported on TCU / NCU interface

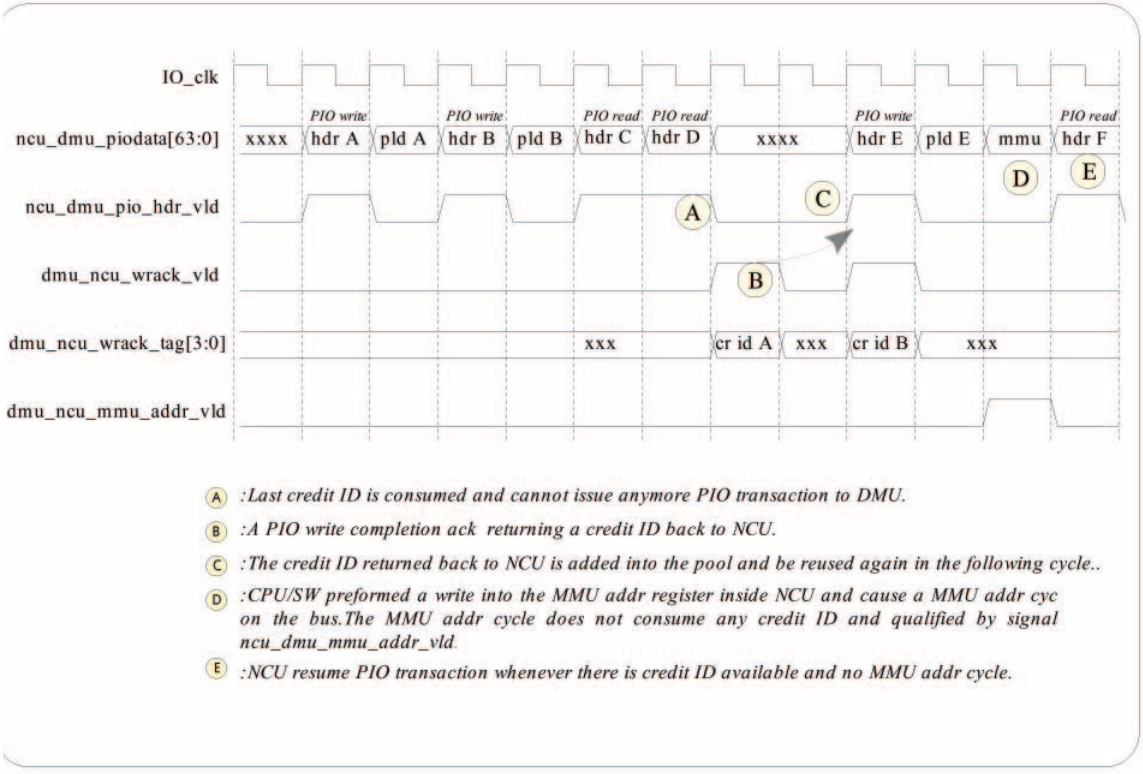
WRITE_REQ	TCU->NCU	128-bit UCB packet (8B header + 8B payload)
READ_REQ	TCU->NCU	128-bit UCB packet (8B header + 8B meaningless payload)
READ_ACK	NCU->TCU	128-bit UCB packet (8B header + 8B payload). Note: PA and Size fields are invalid in a return packet.
READ_NACK	NCU->TCU	64-bit UCB packet (8B header only). Note: PA and Size fields are invalid in a return packet.

The interface timing diagram for NCU/TCU is same as NCU/MCU, which can be found in FIGURE 7-9and FIGURE 7-10 with data bus set to [7:0] and number of cycles set to 16.

7.4.9 NCU / DMUPIO Interface

NCU sends PIO read/write request (non cacheable LOAD_REQ / STORE_REQ) directly through the NCU / DMUPIO interface. NCU keeps a total of 16 “credit ID”s. Each PIO request sends to DMU will consume a “credit ID,” which will be returned from the signal dmu_ncu_wrack_tag[3:0] after a PIO write is completed. The PIO read returned packet from SIU also has a returning “credit ID” embedded in its header. These returned “credit ID”s are put back to the pool and will be reused again. Therefore, there can be a maximum of 16 outstanding PIO read and PIO write requests. Note that DMU has a limit of processing up to 16 PIO requests FIGURE 7-11 show the timing diagram the DMUPIO interface.. Signal dmu_ncu_mmu_addr_vld will be asserted when CPU perform a write to NCU's MMU_ID_ADDR register, and the value of the register is put on to the ncu_dmu_piodata[63:0] bus.

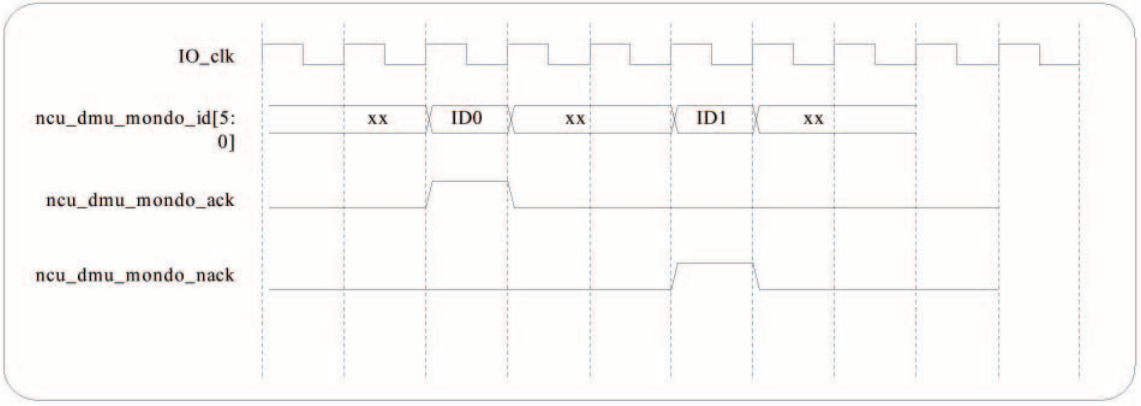
FIGURE 7-11 NCU / DMUPIO Interface Timing Diagram



7.4.10 NCU / DMU Mondo Response Interface

After receiving Mondo Interrupt packet from SIU, NCU directly response to DMU with the Mondo ID which is qualified by an “ack” or a “nack” at the same cycle. The 6-bit ID bus is valid when “ncu_dmu_mondo_ack” or “ncu_dmu_mondo_nack” signal is asserted. FIGURE 7-12 shows the timing diagram for NCU / DMU Mondo Response interface.

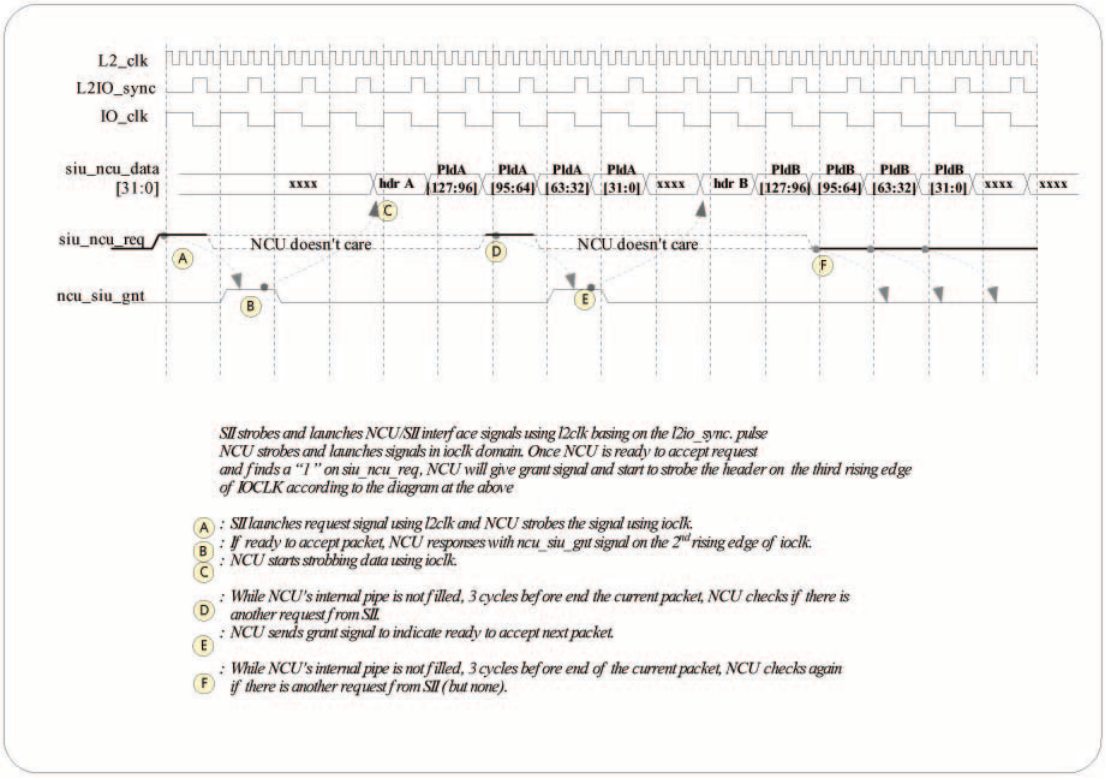
FIGURE 7-12 NCU / DMU Mondo Response Interface Timing Diagram (from NCU to DMU.)



7.4.11 NCU / SII Interface

SII only has upstream path to NCU. Packets received from SII are either PIO read returns or Mondo Interrupts. Wen signal "ncu_siu_gnt" is asserted, in response to an asserted "siu_ncu_req", a new packet transaction should start in the next cycle. Once a packet transaction is in progress, NCU ignores the signal "siu_ncu_req" until two cycle before ending of the current packet. Details of the interface timing is shown in FIGURE 7-13. packet from SII to NCU always have 5 cycles (one header cycle with 4 payload cycles.)

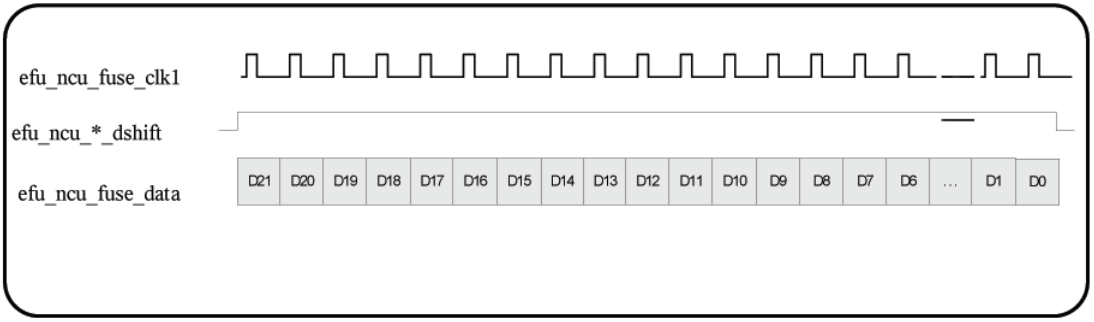
FIGURE 7-13 NCU / SIU Interface Timing Diagram (from SIU to NCU)



7.4.12 EFUSE Interface

NCU only receive Signals from EFUSE. This interface has a serial data signal shared by different register. EFUSE will guarantee there are 22 consecutive bits/data with MSB first per "*dshift" assertion as shown if the following diagram. The only exception is efu_ncu_fusestat_dshift signal, which is 64-bit status information inputting into NCU's EFUSE Status CSR register.

FIGURE 7-14 EFU / NCU Interface Timing Diagram.



7.4.13 Packet Format

7.4.13.1 UCB (Unit Control Block) Data Packet Format

TABLE 7-30 UCB Data Packet Format

bit	Name	definition
[127:64]	Payload Data	8B Payload Data. If packet is from PCX, this field is from PCX's data field. Payload is valid only for “WRITE_REQ,” “READ_ACK,” and “IFILL_ACK” types packets For “READ_REQ,” and “IFILL_REQ” packet, this 8B payload is meaningless. For “READ_NACK,” or “IFILL_NACK” types, there is no payload.
[63:55]	ByteMask	(use only for TCU to DMUPIO packets) This field is being ignored in general. Only exception is when TCU initiates a DMUPIO request. In such case, this field is being treated similar to PCX packet's size field.

TABLE 7-30 UCB Data Packet Format

[54:15]	40 bit PA	This field is valid only for request type packet, and should be ignored for return type packet. CSR Address: Upper 8 bit indicates the block that it should go to. Individual block should only look at the lower 32 bits If packet is from PCX, this field is from PCX's address field.
[14:12]	Request Size	(This field is being ignored in TCU to DMUPIO packets) This field is valid only for request type packets and should be ignored for return type packets. If packet is from PCX, this field is from PCX's size field. Supported size for request type: 3'b000 : 1 Byte (valid only for SSI non-ifill type) 3'b001 : 2 Byte (valid only for SSI non-ifill type) 3'b010 : 4 Byte (valid only for SSI, can be ifill or non-ifill type) 3'b011 : 8 Byte (for all other access) note: UCB protocol only support up to 8Byte payload max. DMUPIO is not in UCB protocol, and, therefore ,is not limited by this restriction.
[11:10]	Buffer ID	NCU sends 2'b00 if a request is originated from Core and sends 2'b01 if request is originated from TCU. All UCB clients returning packets to NCU must return the same value in this field as in the original request packet.

TABLE 7-30 UCB Data Packet Format

[9:7]	CPU ID [2:0]	This field indicates the source CPUID this packet is from, or the target CPUID this packet should be send back to.
[6:4]	Thread ID [2:0]	This field indicates the target thread this packet is from or targeting to.
[3:0]	Packet Type	4'b0000: READ NACK (generates CPX NCU Load Return with U.E. if packet is to CPX) 4'b0001: READ ACK (generates CPX NCU Load Return if packet is to CPX) 4'b0011: IFILL ACK (generates CPX NCU Ifill Return if packet is to CPX) 4'b0111: IFILL NACK (generates CPX NCU Ifill Return with err if packet is to CPX) 4'b0100: READ REQ (from PCX LOAD if packet is from PCX) 4;b0101: WRITE REQ (from PCX STORE if packet is from PCX) 4'b0110: IFILL REQ (from PCX Inst.FILL if packet is from PCX)

7.4.13.2 UCB (Unit Control Block) Interrupt Packet Format

TABLE 7-31 UCB Interrupt Packet Format

bit	Name	definition
[63:57]	reserved	Reserved (may not be 0)
[56:51]	Interrupt vector	interrupt vector (valid only when Packet Type=INT_VEC, MCU, SSI should always use Packet Type=INT, which cause s NCU to ignore this field)
[50:19]	Reserved	Reserved (may not be 0)
[18:10]	Device ID	This field identify the entry of the int_man mem. lookup table.
[9:7]	CPU ID	This field indicates the target CPU this interrupt packet should be sent to when Packet Type=INT (should not be use by,MCU, or SSI)
[6:4]	Thread ID	This field indicates the target thread this interrupt packet should be sent to when Packet Type=INT
[3:0]	Packet Type	4'b1000: INT (interrupt) 4'b1100: INT_VEC (interrupt w/ vector field,cpu_id,and thread_id valid)

7.4.13.3 SII to NCU Header Format

Each header is followed by 4 cycles of payload data.

TABLE 7-32 SIU to NCU Header Format

Header Cycle "siu_ncu_data[31:0]"	name	definition
[31]	TimeOut	Packet timed out (this will cause a CPX NCU Load Return packet with 'err' field set to uncorrectable error)
[30]	DmuAE	Unmapped Error (this will cause a CPX NCU Load Return packet with 'err' field set to uncorrectable error)
[29]	DmuUe	Uncorrected error from DMU (this will normally cause a CPX NCU Load Return packet with 'err' field set to uncorrectable error)
[28]	Ebit	Packet error bit (indicates packet has error and already reported by SII, and NCU will terminate this packet silently with any other action)
[27:22]	Reserved	Reserved (ignore by NCU)
[21:16]	dmc_tag_ecc[5:0]	dmc_ctag ecc check bits
[15:0]	dmc_tag[15:0]	**dmc_tag[15] = 0 --> Mondo Interrupt packet** For Mondo Interrupt, NCU returns {dmc_tag[14:11],dmc_tag[2:1]} back to DMU with Mondo_ack or Mondo_nack Signals asserted. Targeted cpu_thread ID = payload[75:70] mondo_data0=payload[127:64] mondo_data1=payload[63:0] . dmc_tag[15] = 1 --> PIO read return packet dmc_tag[14:12] : reserved (must be 0) dmc_tag[11:8] : NCU credit ID return, dmc_tag[7:6] : buf_id[1:0], 2'b00=normal, 2'b01=jtag access dmc_tag[5:0] : cpu_thread[5:0]

7.4.13.4 NCU to DMUPIO Header Format

TABLE 7-33 NCU to DMUPIO Header Format.

Header Cycle "ncu_dmupio_data[63:0]"	name	definition
[63:61]	Reserved	Reserved (may not be 0)
[60]	PIO read	1'b1 for PIO read 1'b0 for PIO write

TABLE 7-33 NCU to DMUPIO Header Format.

[59:56]	NCU Credit ID	4-bit Credit from that will eventually return back to NCU for reuse. This is to guarantee that there can only be 16 outstanding PIO transactions as DMU cannot take more than 16.
[55:48]	Byte Count / Byte Mask	This field is directly come from 'size' field of a PCX packet For PIO read : (cannot count on upper 5bit to be zero for read case from PCX packet) 8'bxxxx_x000 : 1 Byte 8'bxxxx_x001 : 2 Bytes 8'bxxxx_x010 : 4 Bytes 8'bxxxx_x011 : 8 Bytes 8'bxxxx_x100 : 16Bytes For PIO write : 8bit byte mask indicates which of the 8B of store data should be updated
[47:40]	NCU PIO ID	{buf_id[1:0],CPU_thrID[5:0]}
[39:38]	reserved	Must be 0
[37:36]	Command Mapping	Please see PCIE base/mask CSR registers for mapping details. 2'b11 : Mem64 space 2'b10 : Mem32 space 2'b01 : IO space (if PA[28]=1'b1) 2'b00 : Config space (if PA[28]=1'b0)
[35:0]	Bus Address	PA address [35:0]. This address is PCX packet address, masked with mask registers.

7.4.13.5 DMUPIO Read Request Address and Data Format

When CPU sends a non-cacheable external LOAD_REQ request (PIO read request) to NCU via the PCX interface, the packet contains a 40 bit address, request type, request size, etc.(please reference to the 'Crossbar Packet Definition' document on OpenSPARC T2 web page) Followings are rules for read access to the external PCI space.

1. The 40-bit address is a byte address pointing to the 1st byte CPU is interested in.
2. The most CPU asks for are 16 bytes which is indicated by the PCX packet's size field.

3. DMU/NCU returns 16B to CPU via XBAR (with 64 msb is replicated to 64 lsb for non 16B returns)
4. DMU/NCU do not align the return data.
5. For 2 Byte LOAD_REQ, the fetch should not cross the 2 Byte boundary

   (i.e. ByteAddress should be 0,2,4,6,8..., case 3 below)
6. For 4 Byte LOAD_REQ, the fetch should not cross the 4 Byte boundary

   (i.e. ByteAddress should be 0,4,8,..., case 4 below)
7. For 8 Byte LOAD_REQ, the fetch should not cross the 8 Byte boundary

   (i.e. ByteAddress should be 0,8,..., case 5 below)
8. For 16 Byte LOAD_REQ, the fetch should not cross the 16 Byte boundary

   (i.e. ByteAddress should be 0,x10,x20,x30,..., case 6 below)

Example: let's focus on the lower 16 bits of the 36-bit address, and assume the upper 24 bits point to the correct PIO region in the following cases.

TABLE 7-34 PIO Read Address and Data Format

Case (in plain English)	note	Lower 16 bits of PA field in PCX packet	Lower 2 bit of size field in PCX packet	16 Byte return from SIU some time later
Case 1: LOAD_REQ Byte Address 1 for 1 Byte	(Byte Address can be any value for 1 Byte)	16'h0001	3'b000 (means 1 Byte)	{2{64'hxxAB_xxxx_xxxx_xxxx}} (x's means unknown, cpu 'don't care,' and could be anything depends on what DMU is reading)
Case 2: LOAD_REQ Byte Address 3 for 1 Byte	(Byte Address can be any value for 1 Byte)	16'h0003	3'b000	{2{64'hxxxx_xxAB_xxxx_xxxx}}
Case 3: LOAD_REQ Byte Address 4 for 2 Bytes	(Byte Address 1,3,5,7 are illegal)	16'h0004	3'b001 (means 2 Bytes)	{2{64'hxxxx_xxxx_ABCD_xxxx}}

TABLE 7-34 PIO Read Address and Data Format

Case 4: LOAD_REQ Byte Address 4 for 4 Bytes	(Byte Address 1~3 or 5~7 are illegal)	16'h0004	3'b010 (means 4 Bytes)	{2{64'hxxxx_xxxx_ABCD_EF01}}
Case 5: LOAD_REQ Byte Address 8 for 8 Bytes	(Byte Address 1~7 or 9~f are illegal)	16'h0008	3'b011 (means 8 Bytes)	{2{64'hABCD_EF01_2345_6789}}
Case 6: LOAD_REQ Byte Address 0x10 for 16 Bytes	(Byte Address 1~f are illegal)	16'h0010	3'b100 (means 16 Bytes)	{64'hABCD_EF01_2345_6789_ 0123_4567_89AB_CDEF} Note: this does not imply any data replication, just all bits are valid and no replication for a 16B load return.

7.4.13.6 DMUPIO Write Request Address and Data Format

when CPU sends a non-cacheable external STORE_REQ (PIO write request) to NCU via the PCX interface, the packet contains a 40 bit byte address, request type, request size, etc.(please reference to the 'Crossbar Packet Definition' document on OpenSPARC T2 web page) Followings are rules for write access to the external PCI space.

1. The 40-bit address from PCX is a byte address pointing to the 1st byte CPU is interested in.
2. The most CPU generated store is 8 Bytes which is indicated by the PCX packet's byte mask field.
3. NCU support “partial store” feature by sending the 8bit 'byte mask field', which is position mask, directly to DMU. The 8bit byte mask field can be at any combinations. DMU in OpenSPARC T2 also supports “partial store” feature.
4. NCU does not align any payload data. The 8B payload data is sent to DMU unmodified. The masked 36-bit byte address is sent to DMU with the lower 3 bits PA[2:0] being turn off (always 0) for write only.

Example: let's focus on the lower 16 bits of the 40-bit address, and assume the upper 24 bits point to the correct PCI PIO region in the following cases.

TABLE 7-35 PIO Write Address and Data Format

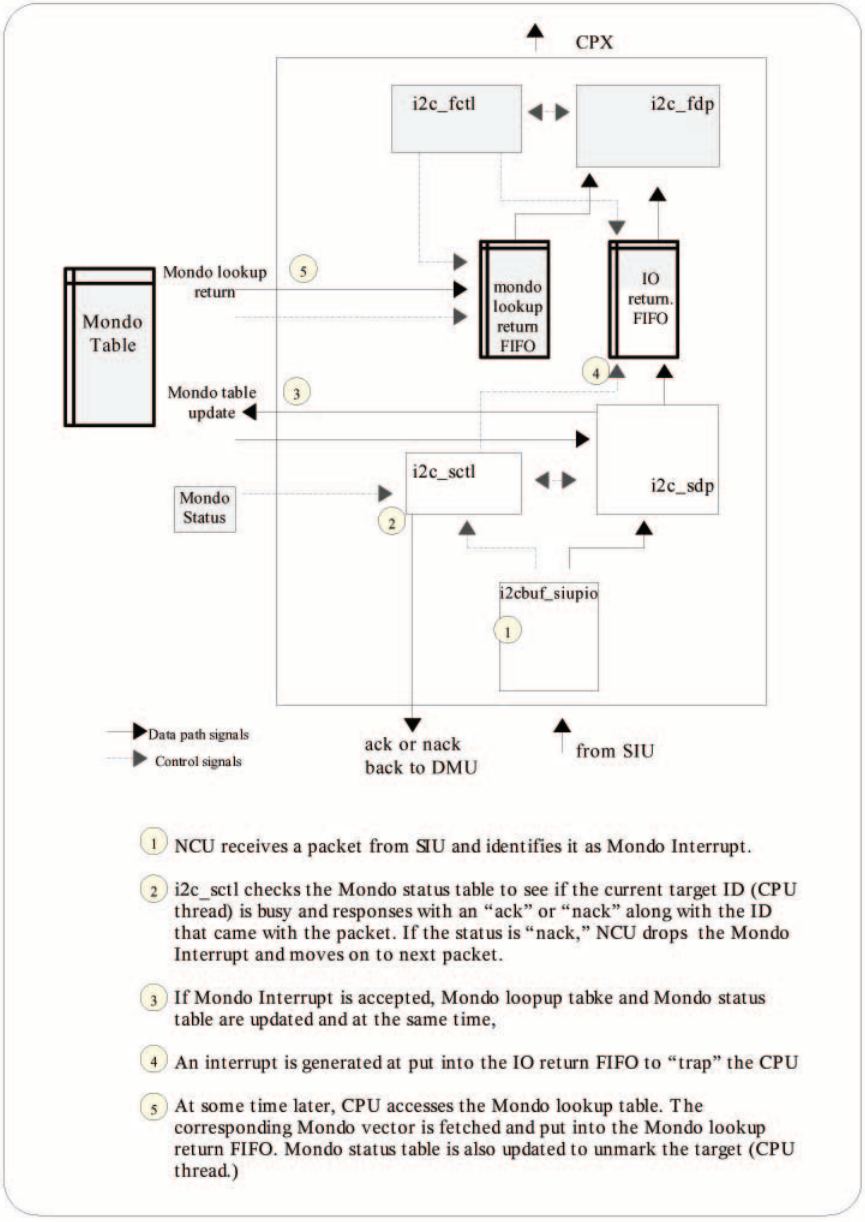
Case (in plain English)	note	Lower 16 bits of address field from PCX packet	8 bit size field from PCX packet	8 Byte payload from PCX and to DMU
Ex. 1: STORE_REQ Byte Address 1 for 1 Byte	(Byte Address from CPU can be any value pointing to 1st critical Byte)	16'h0001 (sending to DMU as 16'h0000)	8'b0100_0000	64'hxxAB_xxxx_xxxx_xxx (x's means unknown, cpu 'don't care,' and could be anything depends on what DMU is reading)
Ex. 2: STORE_REQ Byte Address 5 for 1 Byte	(Byte Address from CPU can be any value pointing to 1st critical Byte)	16'h0005 (sending to DMU as 16'h0000)	8'b0000_0100	64'hxxxx_xxxx_xxAB_xxxx
Ex. 3: STORE_REQ Byte Address 3 for 2 Bytes (contiguous)	(Byte Address from CPU can be any value pointing to 1st critical Byte)	16'h0003 (sending to DMU as 16'h0000)	8'b0001_1000	64'hxxxx_xxAB_CDxx_xxxx
Ex. 4: STORE_REQ Byte Address 3 for 2 Bytes (non-contiguous)	(Byte Address from CPU can be any value pointing to 1st critical Byte)	16'h0003 (sending to DMU as 16'h0000)	8'b0001_0010	64'hxxxx_xxAB_xxxx_CDxx
Ex. 5: STORE_REQ Byte Address 1 for 4 Bytes (non-contiguous)	(Byte Address from CPU can be any value pointing to 1st critical Byte)	16'h0001 (sending to DMU as 16'h0000)	2'b0101_0011	64'hxxAB_xxCD_xxxx_EF01

7.5 Interrupts

7.5.1 Mondo Interrupt Path (External Interrupts)

All interrupts come from DMU / SIU are treated as Mondo Interrupts. It is originated from DMU but send to NCU via SIU. Mondo Interrupts in NCU, actually, including external devices interrupts and MSI Interrupts. However, NCU does not distinguish between them. FIGURE 7-15 shows detail for the Mondo Interrupt path and control. Since NCU serves Mondo Interrupts in the order of receiving, DMU is guarantee to have Mondo “ack” or “nack” back in the order of sending

FIGURE 7-15 Mondo Interrupt Path



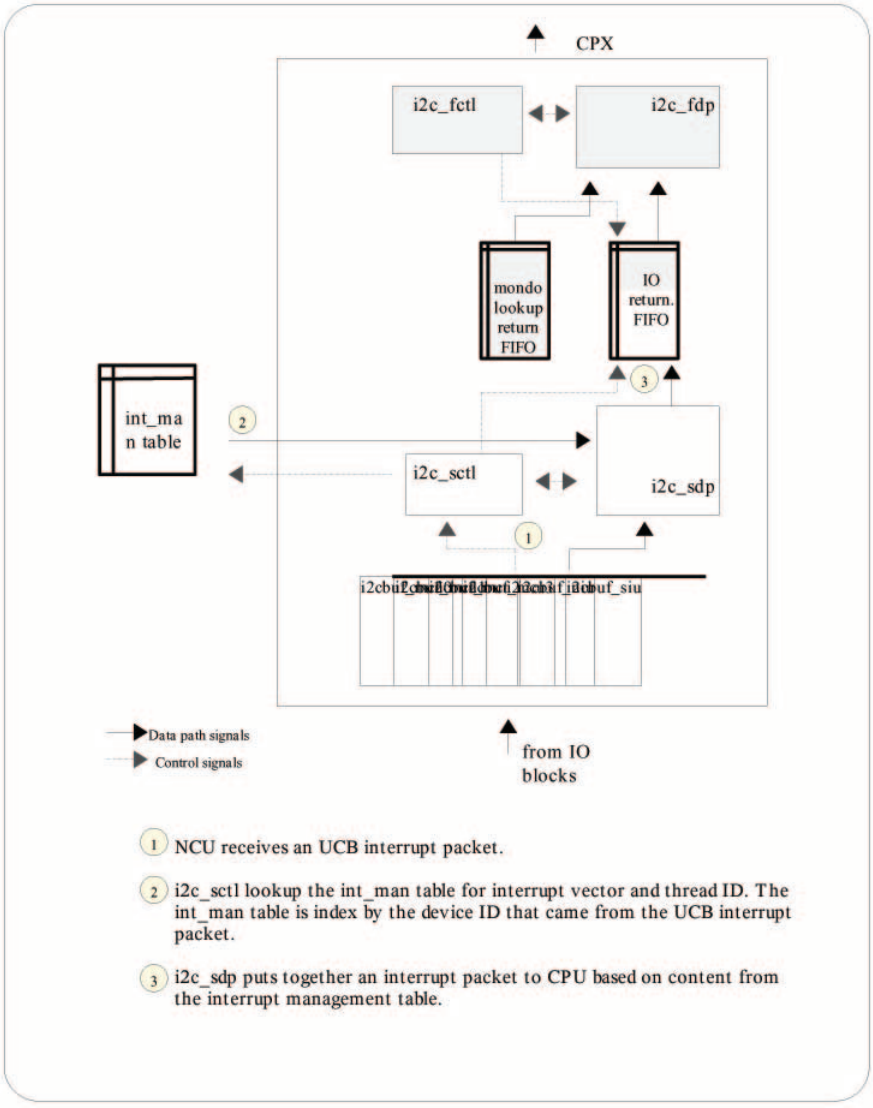
7.5.2 Non Mondo Interrupt (On Chip Interrupt)

The on chip interrupt or Non Mondo Interrupt is identified by its device ID. Each device ID associates with an interrupt source and also index to an entry in Interrupt Management Table (int_man table). This non-mondo interrupt is a “fire-and-forget” type interrupt because once NCU fires the interrupt to the processor, no further information is retained inside NCU. FIGURE 7-15 shows details of the non-mondo type interrupt path, and a list of device ID can be found in TABLE 7-36.

TABLE 7-36 Device ID Assignments

Device ID	definition
0	Reserved
1	MCU ECC errors, counter rollover, SSI errors
2	SSI Interrupt from EXT_INT_L pin
3 ~ 63	Reserved

FIGURE 7-16 Non Mondo Interrupt Path



7.6 NCU Global Physical Address (PA) Assignments

7.6.1 Global Physical Address Assignments

NCU also serves as the Physical Address parser. Each packet dequeued from the downstream main FIFO carries a 40-bit address, also known as physical address (PA). NCU determines the destination of the packet by examining the 8 MSB (bit[39:32]) of the physical address. The address range of each IO subsystem block can be found in TABLE 7-37.

In OpenSPARC T2, CCX (Crossbar) automatically filters out all L2 related packets (L2 non-cacheable CSR and all cacheable packet,) and send them directly to L2. Therefore, packets come from PCX interface to NCU are guaranteed to have bit[39] set to "1'b1."

TABLE 7-37 Global Physical Address Assignments

MSB Address Range[39:32]	Assignment
0x80	NCU
0x82	Reserved
0x83	CCU
0x84	MCUs [13:12] = 2'b00 for MCU0 [13:12] = 2'b01 for MCU1 [13:12] = 2'b10 for MCU2 [13:12] = 2'b11 for MCU3
0x85	TCU
0x86	DBG
0x87	Reserved
0x88	DMUCSR
0x89	RST
0x90	ASI CPU shared registers
0x91 ~ 0x9F	Reserved
0xA0 ~ 0xBF	L2 CSR (handles by CCX directly and does not come to NCU)
0xD0 ~ 0xFE	Reserved
0xFF	SSI (boot ROM)

7.6.2 NCU Local CSR Assignments

7.6.2.1 NCU Management

Each device sends its device ID to NCU along with the UCB interrupt packet. The device ID is used to index into the Interrupt Management table.

TABLE 7-38 Interrupt Management – INT_MAN (0x80_0000_0000) (count 128 step 8)

Bit	Name	Initial Value	R/W	Description
[63:14]	Reserved	0	RO	Reserved
[13:8]	CPU	X	RW	CPUID to manage the device
[7:6]	Reserved	0	RO	Reserved
[5:0]	Vector	X	RW	Interrupt Vector

MONDO_INT_VEC performs the identical function for Mondo Interrupts that INT_MAN performs for other IO interrupts, except that the CPU_ID (thread ID) is specified in the Mondo interrupt transaction.

TABLE 7-39 Mondo Interrupt Vector Register – MONDO_INT_VEC (0x80_0000_0a00)

Bit	Name	Initial Value	R/W	Description
[63:6]	Reserved	0	RO	Reserved
[5:0]	Vector	0	RW	Interrupt Vector for Mondo interrupts (encodes bit set in ASI_SWVR_INTR_RECEVIE)

TABLE 7-40 Processor Serial Number – SER_NUM (0x80_0000_1000)

Bit	Name	Initial Value	R/W	Description
[63:44]	sernum2	0	RO	Chip's serial number programmed by efuse
[43:22]	sernum1	0	RO	Chip's serial number programmed by efuse
[21:0]	sernum0	0	RO	Chip's serial number programmed by efuse

This register is warm reset protected.

TABLE 7-41 EFUSE Status – EFU_STAT (0x80_0000_1008)

Bit	Name	Initial Value	R/W	Description
[63:0]	efu_status	0xFFFFFFFFFFFFFF	RO	Efuse status programmed by efuse block

This register is warm reset protected.

TABLE 7-42 Core Available – CORE_AVAIL (0x80_0000_1010)

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_avail	0xFFFFFFFFFFFFFF	RO	Core available programmed by efuse This register is same as core_available in ASI

This register is warm reset protected.

The following Bank Available, Bank Enable and Bank Enable Status works the same fashion as the ASI's Core Available, Core Enable and Core Enable status. Bank Available is only programmable by EFUSE after POR event, and will not change. This default value is propagated in to Bank Enable register which is programmable at any time after POR event. Finally, the Bank Enable Status register is the one that being used by different clusters / clock disabling. The Bank Enable Status is only updated from Bank enable register at the de-assertion of WMR event.

TABLE 7-43 Bank Available – BANK_AVAIL (0x80_0000_1018)

Bit	Name	Initial Value	R/W	Description
[63:8]	Reserved	**0**	**RO**	Reserved
[7:0]	Bank_avail	**0xFF**	**RO**	Bank available programmed by EFUSE This register indicates the availability of each L2 bank.

This register is warm reset protected.

TABLE 7-44 Bank Enable – BANK_ENABLE (0x80_0000_1020)

Bit	Name	Initial Value	R/W	Description
[63:8]	Reserved	0	RO	Reserved
[7:0]	Bank_enable	0xFF	RW	Received initial from Bank Available after POR event. This programed value is reflected onto Bank Enable Status at the de-assertion of WMR event. (note: hardware forces a non-available bank indicated by Bank Available register to 0, so that SW cannot enable a non-available bank)

This register is warm reset protected.

There are certain rules for L2 partial bank mode, please refer to L2, SIU, CCX specifications for details. The Bank Enable Status changes only after the de-assertion of WMR event. NCU provides a signal for each bank pair. For example, BA01 means bank 0 and bank 1. BA01 is “0” if bank 0 or bank 1 is disable or unavailable.

Since there are encoding involves between Bank Enable and Bank Enable Status. A preview version of partial bank Signals is provided whenever there is a change in Bank Enable register. However, the final/usable copy that provided to different clusters will not be changed until after WMR event.

The PM (partial mode) signal is 1 if any of the bank is not enable. Each of the BA* signal is a result of 2 bank enable being “ANDED” together. However, there are some illegal combinations of BA* Signals, and NCU is the BA* rule enforcer. Please refer to SIU/L2/CCX document for illegal case details. The following is the illegal case mapping by NCU.

TABLE 7-45 Illegal Case Mapping

Illegal PM,BA67,BA45,BA23,BA01 combinations		*Resulting BA67,BA45,BA23,BA01*
1,0,0,0,0	-->	1,0,0,0,0 (Bad Chip, no mapping)
1,0,1,1,1	-->	1,0,0,1,1
1,1,0,1,1	-->	1,0,0,1,1
1,1,1,0,1	-->	1,1,1,0,0
1,1,1,1,0	-->	1,1,1,0,0

TABLE 7-46 Bank Enable Status – BANK_ENABLE_STATUS (0x80_0000_1028)

Bit	Name	Initial Value	R/W	Description
[63:13]	Reserved	0	RO	Reserved
[12]	PM_preview	0	RO	L2 partial mode preview value
[11]	BA67_preview	1	RO	BA67 preview value
[10]	BA45_preview	1	RO	BA45 preview value
[9]	BA23_preview	1	RO	BA23 preview value
[8]	BA01_preview	1	RO	BA01 preview value
[7:5]	Reserved	0	RO	Reserved
[4]	PM	0	RO	L2 partial mode (final copy to different clusters)
[3]	BA67	1	RO	Availability of bank 6 and bank 7 (final copy to different clusters)
[2]	BA45	1	RO	Availability of bank 4 and bank 5 (final copy to different clusters)
[1]	BA23	1	RO	Availability of bank 2 and bank 3 (final copy to different clusters)
[0]	BA01	1	RO	Availability of bank 0 and bank 1 (final copy to different clusters)

TABLE 7-47 L2 Index Hash Enable – L2_IDX_HASH_EN (0x80_0000_1030)

Bit	Name	Initial Value	R/W	Description
[63:1]	Reserved	0	RO	Reserved
[0]	L2_Idx_Hash_en	0	RW	L2 indexing enable. New value will not propagate to L2_Idx_hash_en_status until the next wrm_reset.

This register is warm reset protected.

TABLE 7-48 L2 Index Hash Enable Status – L2_IDX_HASH_EN_STATUS (0x80_0000_1038)

Bit	Name	Initial Value	R/W	Description
[63:1]	Reserved	0	RO	Reserved
[0]	L2_Idx_hash_en_status	0	RO	Final / usable copy of l2_index_hash_en to SII and SPC

TABLE 7-49 NCU/SSI SCK clock select – NCU_SCKSEL (0x80_0000_3040)

Bit	Name	Initial Value	R/W	Description
[63:2]	Reserved	0	RO	Reserved
[1:0]	ncu_scksel	0	RW	when “01”, ssi_sck = iol2clk / 4 all other cases, ssi_sck = iol2clk / 8

This register is warm reset protected.

7.6.2.2 RAS Related Registers

Logics sets ESR bit on the error indication (thus recording the error) if the corresponding bit in the ELE is set. Errors will continue to be recorded until a logged error also has its respective EIE bit set. This causes the NCU to dispatch an “SocError” message using a CPX Error Indication Packet. A “snapshot” of the ESR be taken/stored in the PER register and the ESR cleared.

All CPUTHR ID fields are protected by ECC (SecDed) in NCU's memories. As a general policy, if an uncorrectable error happens at CPUTHR ID, NCU terminates the corrupted packet silent without replying to any of the CPUTHR ID since the CPITHR ID is unknown. TABLE 7-50 shows expected NCU behavior when NCU detected an error.

TABLE 7-50 NCU Response to Error

Error type	Cause of source	Transaction (return packet)	Syndrome reg
NcuDmuCredit	DMUPIO store from PCX interface	When NCU received wack with parity error, NCU drop the wack_tag.	No
NcuCtagCe [23]	1 DMUPIO read return for sii interface 2. MONDO interrupt from sii interface	Complete. send load return cpx packet without error Complete. send INT cpx packet without error	No
NcuCtagUe [22]	1. DMUPIO read return from sii interface 2. MONDO interrupt from sii interface	Terminated. Do not send load return CPX packet Continue. send INT CPX packet with error but not ack back mondo id	Format 2 data (ctag is corrupted)
NcuDataParity [14]	1. DMUPIO read return from SII interface 2. MONDO interrupt from sii interface	Continue. Send load return CPX packet with error terminate, do not send INT CPX packet and not ack back mondo id	Format 2 data
NcuDmuUe[21]	1. DMUPIO store/load (read) from PCX interface	Terminate. Not forward packet to DMUPIO. If cputhr id is corrupted: no response is generated If cputhr id is not corrupted: return without error bit set for store, return with error bit set for load.	Format 1 RCTP=4'hf RCTP data match with PCX packet

TABLE 7-50 NCU Response to Error

NcuCpxUe [20]	1. PIO/CSR store from PCX interface. 2. Load return/IO interrupt form Ios	Continue transfer packet to target. Don't send store ack return CPX packet. (EJR write can't be affected) \ Terminate. Don't send load return/Interrupt CPX packet.	No
NcuPcxUe [19]	PIO/CSR load/store form PCX interface	Terminate. Don't pass done packet to target. Don't send store ack return CPX packet	Format 1 RCTP=4'h1011 RCTP data match with PCX packet (data is corrupted data)
NcuPcxData[18]	PIO/CSR Load from PCX interface PIO/CSR store from PCX interface	Continue (read) Terminated. (write) Ack back to CPU without error.	Format 1 RCTP = 4'h0110 RCTP data match with PCX packet
NcuIntTable [17]	Load INT table from PCX interface IO interrupts form IO (NIU/MCU/SSI) interface	Continue, send load ack return CPX packet with error terminate. Don't' send INT CPX packet	Format 1 RCTP = 4'hf RCTP data match with PCX packet format 1 RCTP = 4'h6 RCTP data match with interrupt table (corrupted)
NcuMondoFifo[16]	load MONDO table from PCX interface	terminate. Don't sent ack return CPX packet	
NcuMondoTable[15]	Load the MOND table	Continue, send load ack return CPX packet with error	

TABLE 7-51 Error Status Register - ESR (0x80_0000_3000)

Bit	Name	Initial Value	R/W	Description
[63]	valid	0	RW	Valid : indicates that any error or multiple error has been recorded.
[62:41]	Reserved	0	RO	Reserved
[42]	NcuDmuCredit	0	RW	Credit token to NCU for DMU pio write credits

TABLE 7-51 Error Status Register - ESR (0x80_0000_3000)

[41]	Mcu3Ecc	0	RW	MCU3 ECC Correctable (exceeded data CE threshold)
[40]	Mcu3Fbr	0	RW	MCU3 Fbdimm Recoverable
[39]	SpareBit[4]	0	RW	This bit is always set to 0 and does not caputre anything regardless of EJR settings (see note below table)
[38]	Mcu2Ecc	0	RW	MCU2 ECC Correctable (exceeded data CE threshold)
[37]	Mcu2Fbr	0	RW	MCU2 Fbdimm Recoverable
[36]	SpareBit[3]	0	RW	This bit is always set to 0 and does not caputre anything regardless of EJR settings (see note below table)
[35]	Mcu1Ecc	0	RW	MCU1 ECC Correctable (exceeded data CE threshold)
[34]	Mcu1Fbr	0	RW	MCU1 Fbdimm Recoverable
[33]	SpareBit[2]	0	RW	This bit is always set to 0 and does not caputre anything regardless of EJR settings (see note below table)
[32]	Mcu0Ecc	0	RW	MCU0 ECC Correctable (exceeded data CE threshold)
[31]	Mcu0Fbr	0	RW	MCU0 Fbdimm Recoverable
[30]	SpareBit[1]	0	RW	This bit is always set to 0 and does not caputre anything regardless of EJR settings (see note below table)
[29]	NiuDataParity	0	RW	Data Parity error in the DMA read return from the SIO
[28]	NiuCtagUe	0	RW	Ctag double bit Uncorrected error from the SIODMA read return
[27]	NiuCtagCe	0	RW	Ctag single bit Corrected Error from the SIO DMA read return
[26]	SioCtagCe	0	RW	Ctag single bit Corrected Error after the OLD Fifo.
[25]	SioCtagUe	0	RW	Ctag double bit Uncorrected Error from the OLD Fifo. Recommended Fatal Error
[24]	SpareBit[0]	0	RW	(Does not capture anything)
[23]	NcuCtagCe	0	RW	Ctag single bit Corrected error on Interrupt write or PIO read return.
[22]	NcuCtagUe	0	RW	Cag double bit error on Interrupt write or PIO read return. Recommended Fatal Error. (NCUSYN)

TABLE 7-51 Error Status Register - ESR (0x80_0000_3000)

[21]	NcuDmuUe	0	RW	For IOMMU conflicts. (NCUSYN)
[20]	NcuCpxUe	0	RW	Error in Output Fifo to CPX.
[19]	NcuPcxUe	0	RW	CPU PIO/CSR commands, may be Fatal. (NCUSYN)
[18]	NcuPcxData	0	RW	Error in CPU PCX Fifo. (NCUSYN)
[17]	NcuIntTable	0	RW	Error in NCU read of Interrupt table. (NCUSYN)
[16]	NcuMondoFifo	0	RW	Parity/ECC error in read of Mondo Fifo
[15]	NcuMondoTable	0	RW	Parity/ECC error in CPU read Mondo table
[14]	NcuDataParity	0	RW	Parity for Interrupt write or PIO read return from the SIO. (NCUSYN)
[13]	DmuDataParity	0	RW	Data Parity error in the DMA read return from the SIO
[12]	DmuSiiCredit	0	RW	Parity error in the DMA write acknowledge Credit from the SII. Recommended Fatal Error
[11]	DmuCtagUe	0	RW	Ctag duble bit Uncorrected error from the SIO DMA read return. Recommended Fatal Error
[10]	DmuCtagCe	0	RW	Ctag single bit Corrected error from the SIO DMA read return
[9]	DmuNcuCredit	0	RW	Parity error in the PIO read/Mondo acknowledge Credit from the NCU Recommended Fatal Error
[8]	DmuInternal	0	RW	Rocommended Fata Error
[7]	SiiDmuAparity	0	RW	Parity error for Address field for DMA transactions from DMU Fifo. (SIISYN)
[6]	SiiNiuDParity	0	RW	Data parity error for DMA writes from DMU Fifo. (SIISYN)
[5]	SiiDmuDParity	0	RW	Data parity error for DMA writes from DMU Fifo. (SIISYN)
[4]	SiiNiuAParity	0	RW	Parity error fro Address field for DMA transactions from NIU Fifo. (SIISYN)
[3]	SiiDmuCtagCe	0	RW	Ctag single bit Corrected error, in transaction from NIU Fifo
[2]	SiiNiuCtagCe	0	RW	Ctag single bit Corrected error in transaction from NIU Fifo
[1]	SiiDmuCtagUe	0	RW	Ctag double bit Uncorrected ECC or Command Parity Error in transaction from DMU Fifo.(SIISYN)
[0]	SiiNiuCtagUe	0	RW	Ctag double bit Uncorrected ECC or Command Parity Error in transaction from NIU Fifo.(SIISYN)

This register is warm reset protected.

Note – Bit[30], [33], [36], and [39] in the ESR register does not capture anything from hardware point of view. Even though the corresponding EJE is set, these bit still capture 0. However, software can set these bit to 1 and cause SOC interrupt or fatal error for testing purposes.

ELE provides the capability to select individual error events to be logged in the ESR. If a 'Log Enable' bit is set, and the corresponding error type signal is asserted, then the respective bit position in the ESR's “Recorded Error Type” field is set.

TABLE 7-52 Error Log Enable - ELE (0x80_0000_3008)

Bit	Name	Initial Value	R/W	Description
[63:43]	Reserved	0	RO	Reserved
[42:0]	Error Log Enable	0x7FFFFFFFFF	RW	1-to-1 corresponding to ESR register

This register selects individually logged errors to dispatch an SocError message. Each bit enables interrupting (dispatching SOCError) for the respective bit position in the ESR. Interrupts may be sent if no other SOCError is still pending as indicated by the PER valid bit=1. Thus, if no pending SOCError (i.e. PER valid=0) and the respective “Interrupt Enable” bit is set, the SOCError indication will be dispatched for logged errors at this bit location.

TABLE 7-53 Error Interrupt Enable - EIE (0x80_0000_3010)

Bit	Name	Initial Value	R/W	Description
[63:43]	Reserved	0	RO	Reserved
[42:0]	Error Interrupt Enable	0	RW	1-to-1 corresponding to ESR register

This register provides the capability to select individual error checking nodes to have their parity/ECC bits flipped. When the respective bit is set, the parity/ECC will be flipped, causing on error, for this particular parity/ECC checking location.

TABLE 7-54 Error Injection Register - EJR (0x80_0000_3018)

Bit	Name	Initial Value	R/W	Description
[63:43]	Reserved	0	RO	Reserved
[42:0]	Error Injection Enable	0	RW	1-to-1 corresponding to ESR register with exception of bit[30],[33],[36], and [39], which the corresponding ESR bit always capture 0 regardless of EJR is set.

Each error type may be program to cause a Fatal Error This register enables an error to cause the signal “ncu_rst_fatal_error” to be asserted to the Reset Unit. If the respective “Fatal Error Enable” bit is set, and the corresponding error type is asserted, a fatal error will be dispatched to the Reset Unit. This functionality is not dependent on the settings of the ESR, PER, ELE or EIE.

TABLE 7-55 Fatal Error Enable - FEE (0x80_0000_3020)

Bit	Name	Initial Value	R/W	Description
[63:43]	Reserved	0	RO	Reserved
[42:0]	Fatal Error Enable	0	RW	1-to-1 corresponding to ESR register

This register is a snapshot copy of the entire 64-bit of the ESR. This “snapshot” is taken when NCU initiates an SOC Error packet dispatch. This is caused when an error type occurs that has both the respective “log” enable and respective “interrupt” enable bit positions set. After an SOC Error message, the thread's trap handler may read this register to determine the error “cause”. When this register's Valid bit is set further SOCError message dispatches are disabled.

TABLE 7-56 Pending Error Register - PER (0x80_0000_3028)

Bit	Name	Initial Value	R/W	Description
[63:0]	(same as ESR)	(same as ESR)	(same as ESR)	(same as ESR)

This register is warm reset protected.

The SII Error Syndrome Register stores the syndrome (header) information from an SII caused error event. This register is located in NCU. Data is sent to NCU from SII on a special 4-bit serial bus (please refer to Appendix B). When the logging is disable for this error type, NCU will simply ignore the data syndrome data transfer from SII.

In this case, it will retain the prior data already stored in the SIISYN register. If bit[63],"Valid"-bit, is already set, NCU will ignore further SIISYN coming from SII until software clears this bit.

TABLE 7-57 SII Error Syndrome - SIISYN (0x80_0000_3030)

Bit	Name	Initial Value	R/W	Description
[63]	valid	0	RW	valid
[62:59]	Reserved	0	RO	Reserved
[58:56]	Etag	0	RW	Indicates which type of error is associated with this syndrome. This is the lower 3-bit of the error type index in ESR (SII error types only limited to bit7-bit0). For example, the "Etag" of a "SiiNiuAParity" error = 4.
[55:40]	Ctag	0	RW	16-bit CTAG or ID field from SII or DMU/SII header (refer to SII MAS)
[39:0]	PA	0	RW	40 physical address (refer to SII MAS)

This register is warm reset protected.

TABLE 7-58 NCU Error Syndrome - NCUSYN (0x80_0000_3038)
If bit[62] is 0: format 1

Bit	Name	Initial Value	R/W	Description
[63]	Valid	0	RW	Valid
[62]	Format=0	0	RW	Format 0
[61:58]	RCTP	0	RW	Rqtyp,Cpu,Thr,PA valid
[57:56]	Reserved	0	RO	Reserved
[55:51]	etag	0	RW	Which bit in ncuesr causes loading of syndrome
[50:46]	Rqtyp	0	RW	Packet request type
[45:43]	Cpu_id	0	RW	CPU ID
[42:40]	Thr_id	0	RW	Thread ID
[39:0]	PA	0	RW	40bit PA

This register is warm reset protected.

TABLE 7-59 NCU Error Syndrome - NCUSYN (0x80_0000_3038)
If bit[62] is 1

Bit	Name	Initial Value	R/W	Description
[63]	Valid	0	RW	Valid
[62]	Format=1	0	RW	Format 1
[61:58]	Reserved	0	RW	Rqtyp,Cpu,Thr,PA valid
[57:56]	Reserved	0	RO	Reserved
[55:51]	etag	0	RW	Which bit in ncuesr causes loading of syndrome
[50:46]	Reserved	0	RW	Packet request type
[45:43]	Reserved	0	RW	CPU ID
[42:40]	Reserved	0	RW	Thread ID
[39:0]	CTAG	0	RW	{24'b0,ctag[5:0]}

This register is warm reset protected.

TABLE 7-60 DBG1 Error Event Trigger Enable - NCU_CREG_DBGTRIG_EN (0x80_0000_4000)

Bit	Name	Initial Value	R/W	Description
[63:1]	Reserved	0	R/O	reserved
[0]	dbgtrigen	0	R/W	Enable dbg1 error event trigger.

This register is warm reset protected.

7.6.2.3 Mondo Table Access

The following register are used to manage the Mondo Interrupts.

When NCU receives a Mondo interrupt, it sets the Busy bit and ack DMU. When a Busy bit is set, it means an interrupt is waiting to be serviced or is being serviced. Software needs to reset the Busy bit after it completes servicing the interrupt. If the Busy bit is already set when an interrupt arrives at NCU, a NACK will be sent back to DMU. The Busy bit is set after a reset and software has to clear it to begin receiving interrupts.

There are two Mondo Interrupt Mondo Tables. The tables are read-only by software and the entries are updated by DMU Mondo interrupts, provided that corresponding Busy bit is not currently set. NCU will ack the interrupt if it is not busy, otherwise the NCU will NACK it.

TABLE 7-61 Mondo Interrupt Data0 – MONDO_INT_DATA0 (0x80_0004_0000) (Count 64 Step 8)

Bit	Name	Initial Value	R/W	Description
[63:0]	Data0	X	RO	First 64 bits of Mondo interrupt data

TABLE 7-62 Mondo Interrupt Data1 – MONDO_INT_DATA1 (0x80_0004_0200) (Count 64 Step 8)

Bit	Name	Initial Value	R/W	Description
[63:0]	Data1	X	RO	Second 64 bits of Mondo interrupt data

When a thread reads the following alias register, it is reading its own entry in the Mondo Data0 table (i.e. The PA will from PCX bus will be ignored, and the cputhr[5:0] will be used for accessing the table entry.) This is designed for a CPU thread accessing its own entry without doing address calculation or knowing its own cpu thread I.D. If access if from JTAG the cputhr[5:0] in UCB packet will be used for table indexing.

TABLE 7-63 Alias Mondo Interrupt Data0 – MONDO_INT_ADATA0 (0x80_0004_0400)

Bit	Name	Initial Value	R/W	Description
[63:0]	Data0	X	RO	First 64 bits of Mondo interrupt data

When a thread reads the following alias register, it is reading its own entry in the Mondo Data1 table (i.e. The PA will from PCX bus will be ignored, and the cputhr[5:0] will be used for accessing the table entry.) This is designed for a CPU thread accessing its own entry without doing address calculation or knowing its own cpu thread I.D. If access if from JTAG the cputhr[5:0] in UCB packet will be used for table indexing.

TABLE 7-64 Alias Mondo Interrupt Data1 – MONDO_INT_ADATA1 (0x80_0004_0600)

Bit	Name	Initial Value	R/W	Description
[63:0]	Data1	X	RO	Second 64 bits of Mondo interrupt data

TABLE 7-65 Mondo Interrupt Busy – MONDO_INT_BUSY(0x80_0004_0800) (Count 64 Step 8)

Bit	Name	Initial Value	R/W	Description
[63:7]	Reserved	0	RO	Reserved
[6]	Busy	1	RW	Hardware set Busy to "1" when an interrupt is received. Hardware nacks an incoming Mondo interrupt if Busy bit is already set.
[5:0]	Reserved	0	RO	Reserved

When a thread reads the following alias register, it is reading its own entry in the Mondo Busy table (i.e. The PA will from PCX bus will be ignored, and the cputhr[5:0] will be used for accessing the table entry.) This is designed for a CPU thread accessing its own entry without doing address calculation or knowing its own cpu thread I.D. If access if from JTAG the cputhr[5:0] in UCB packet will be used for table indexing.

TABLE 7-66 Alias Mondo Interrupt Busy – MONDO_INT_ABUSY(0x80_0004_0a00)

Bit	Name	Initial Value	R/W	Description
[63:7]	Reserved	0	RO	Reserved
[6]	Busy	1	RW	Hardware set Busy to "1" when an interrupt is received. Hardware nacks an incoming Mondo interrupt if Busy bit is already set.
[5:0]	Reserved	0	RO	Reserved

7.6.3 ASI Registers

The ASI registers could be accessible by both JTAG and core. The algorithm for mapping from ASI address to IO address is as follows:

PA[39:32] = 0x90

PA[31:29] = core_id[2:0] (physical core id)

PA[28:26] = tid[2:0] (thread id)

PA[25:18] = asi[7:0]

PA[17:3] = VA[17:3]

PA[2:0] = 000

If it's a register that is shared by all virtual cores, then the core_id, PA[31:29] and thread_id, PA[28:26] are ignored. NCU always decode only PA [25:0] if PA[39:32]= 0x90.

7.6.3.1 Core Available Register – ASI_CORE_AVAILABLE (0x90_0104_0000)

(ASI:41 VA:00)

This register is programmed by EFUSE controller after POR is deasserted. NCU will detect the de-assertion of efu_ncu_coreavail_dshift signal which triggers update to Core Enable, Core Enable Status and XIR Steering registers. The granularity of the fuses is at each physical core level, and there are 8 core in OpenSPARC T2. Therefore, physically there are only 8 bits for this register. Hardware automatically expands each bit (representing a core) to 8 bits and becomes a total of 64 bit representing 64 threads.

- JTAG accessible (RO)

TABLE 7-67 Core Available Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_available	0xFFFFFFFFFFFFFFFF(by POR)	RO	A one means the thread is available

This register is warm reset protected.

7.6.3.2 Core Enable Status Register – ASI_CORE_ENABLE STATUS (0x90_0104_0010)

(ASI:41 VA:10)

The Core Enable Status Register is updated from Core Enable register at the deassertion of “warm reset”, or from Core Available register at de-assertion of efu_ncu_coreavail_dshift signal (after POR deasserted). JTAG could program the Core Enable register after POR and before the “warm reset,” so that Core Enable Status register takes the value of Core enable at the next “warm reset” deassertion.

Hardware implements only 8-bit for this register. When SW reads, NCU automatically expands each bit to 8-bit wide and becomes 64 bits total to represent 64 threads. In OpenSPARC T2, CPU uses the value of this register to gate off the clock to the appropriate physical core.

- JTAG accessible (RO)

- A thread that is not available in the Core Available register must have its corresponding status bit set to 0 by hardware.

TABLE 7-68 Core Enable Status Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_enable_status	0xFFFFFFFFFFFFFFF	RO	A one means the thread is currently enabled

7.6.3.3 Core Enable Register – ASI_CORE_ENABLE (0x90_0104_0020)

ASI:41 VA:20

This register is first update after POR (actually at the assertion of efu_ncu_coreavail_dshift) based on Core Available register. When SW uses this register to enable/disable a core or thread, the effect of programming this register will take place only after the following “warm reset.”

Hardware implements only 8 bits, representing 8 cores for this register. When reading, NCU expands each bit into 8 bits and becomes a total of 64 bits, representing 64 threads. When writing, NCU ANDed 8 corresponding bits to a physical core to reduce the 64 bits Signals down to 8 bits which representing 8 cores.

- JTAG accessible: RW
- Bits corresponding to the same core is ANDed together by NCU before writing into the register. So, if one thread is being disabled, all threads within the same physical core are also being disabled.
- Hardware forces all threads in an unavailable core's (based on Core Available register) to be disabled.
- Hardware enforces “no all-core-disabled” rule to protect the situation that all cores are disabled by SW or by JTAG. If JTAG writes all 0s to this register, NCU will set the lowest available core (based on Core Available register) to 1. If CPU writes all 0 to this register, NCU will keep the bit corresponding to CPU that initiates the command to 1. A disabled / unavailable thread (basing on core available and core enable status registers) should never access this register. Unpredictable hardware behavior will be resulted in such case.

TABLE 7-69 Core Enable Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_enable	0xFFFFFFFFFFFFFFF (by POR)	R/W	A one means the thread will be enable following the next “warm reset”

This register is warm reset protected.

7.6.3.4 XIR Steering Register – ASI_XIR_STEERING (0x90_0104_0030)

(ASI:41 VA30)

SW can program which thread gets XIR when XIR pin is asserted. SW can program this register such that all threads, a subset of threads, a thread, or none of the threads will get XIR.

XIR Steering register first receives a default value based on Core available register after POR (actually at deassertion of efu_ncu_coreavail_dshift). At each deassertion of "warm reset," XIR Steering register gets new default value basing on Core Enable register which could be programmed by SW or JTAG.

- JTAG accessible RW
- If a core is not enable, all corresponding bits in XIR Steering register are force to 0 by hardware.

TABLE 7-70 XIR Steering Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Xir_steering	0xFFFFFFFFFFFFFF	R/W	A one means the thread will receive a "reset" interrupt when XIR external pin is asserted

7.6.3.5 Core Running RW Register – ASI_CORE_RUNNING_RW(0x90_0104_0050)

(ASI:41 VA:50)

SW uses this register to park or unpark a thread. Each bit position corresponds to a thread. If the bit is set to 1, the thread is running. When set to 0, the thread is parked. A parked thread stops execute new instructions and will not initiate transaction except in response to a coherancy transaction initiated by other threads. It could take arbitrarily long from the time this register is programmed to the thread is actually parked or unparked.

Upon "warm reset," this register is set to all 0. When NCU receives the rst_ncu_wake_thread signal from RST cluster, NCU will set the lowest available thread bit to 1 based on Core Enable Status register. This thread becomes the master thread. Privileged software, running on the master thread, will subsequently write to this register to unpark other threads. It is up to software to perform the initializations that are required by thread upon unparking. There are 3 ways to program this register:

1. Writing directly to Core Running RW register
2. Alternatively, SW can write a 1 to the corresponding thread bit in Core Running W1S register. This results in setting the corresponding bit in Core Running RW register to 1.
3. Alternatively, SW can write a 1 to the corresponding thread bit in Core Running W1C register. This results in clearing the corresponding bit the Core Running RW register to 0.

- JTAG accessible RW
- Hardware forces all unavailable or disabled threads to be parked (base on Core Enable Status register) Writing 1 into the disabled thread bits will have no effect.

Only JTAG is able to park all threads during debug by writing all '0' to core_running register. Other than JTAG, hardware enforce “no all-thread-parked” rule. When core write to core_running register to park all threads, the hardware will keep the requesting thread unparked. A disabled / unavailable thread (basing on core available and core enable status registers) or a parked thread should never access this register. Unpredictable hardware behavior will be resulted in such case.

TABLE 7-71 Core Running RW Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_running RW	0x1 (by POR and WMR)	R/W	A one means the thread is being unparked. A zero means the thread is current park or disabled. The status is reported in Core_running_status register

7.6.3.6 Core Running Status Register – ASI_CORE_RUNNING_STATUS (0x90_0104_0058)

(ASI:41 VA:58)

Each SPC thread will send spc_core_running_status to indicate its status. The SPC thread determines the status of each thread by the following criteria. The SPC thread receives a request to park or unpark the based upon a '1' to '0' or '0' to '1' transition on the ncu_spc_core_running signal from NCU. An indeterminate time later, once all activity for that thread has been processed (the store buffer is empty, any pending cache and TLB misses have bee processed, and all instructions have completed execution), the SPC will drive the spc_cmp_core_running_status signal to a '0' (to signal the thread is parked) or to a '1' (to signal the thread is running). Upon “warm reset,” Core Running Status should be all 0s.

TABLE 7-72 Core Running Status Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_running_status	0x1 (by POR and WMR)	RO	A one means the thread is currently running. A zero means the thread is currently parked or disabled.

7.6.3.7 Core Running W1S Register – ASI_CORE_RUNNING_W1S (0x90_0104_0060)

(ASI:41 VA:60)

TABLE 7-73 Core Running W1S Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_running_W1S	N/A	WO	Write one to a bit will cause the corresponding bit in core_running_rw register to be set to a one. Write zero or

7.6.3.8 Core Running W1C Register – ASI_CORE_RUNNING_W1C (0x90_0104_0068)

(ASI:41 VA:68)

TABLE 7-74 Core Running W1C Register

Bit	Name	Initial Value	R/W	Description
[63:0]	Core_running_W1C	N/A	WO	Write one in a bit will cause the corresponding bit int core_running_rw register to be cleared to a zero. Write zero means no change on the bit.

7.6.3.9 Interrupt Vector Dispatch Register – INT_VEC_DISP (0x90_01CC_0000)

(ASI:73 VA:00)

A thread may write to the following register to trigger an interrupt to another thread. NCU will generate an interrupt packet and send to a targeted CPU Thread specified in CPU_TH[5:0]. TCU may also write into this register to generate an interrupt to a specific CPU Thread.

TABLE 7-75 Interrupt Vector Dispatch Register

Bit	Name	Initial Value	R/W	Description
[63:14]	Reserved	0	RO	Reserved (NCU ignores write to these bits)
[13:8]	Thread	0	WO	CPU_TH[5:0]
[7:6]	Reserved	0	RO	Reserved (NCU ignores write to these bits)
[5:0]	Vector	0	WO	Interrupt Vector (encodes bit set in ASI_SWVR_INTR_RECEVIE)

7.6.3.10 RAS Error Steering Register – RAS_ERR_STEERING (0x90_0104_1000)

(ASI:41 VA:1000)

This register stores the virtual core ID (VCID), which is used by NCU to determine the error thread target of socerror messages. This 6-bit cpuid + threadid will be included in cpx packet. Refers to RAS spec, section 7.3.

TABLE 7-76 RAS Error Steering Register

Bit	Name	Initial Value	R/W	Description
[63:6]	reserved	0	RO	Reserved.
[5:0]	VCID	0	RW	cpuID+threadID for target error thread location.

This register is warm reset protected.

7.6.3.11 ASI CMP Tick Enable Register – ASI_CMP_TICK_ENABLE(0x90_0140_0038)

(ASI:41 VA:38)

This register is used to synchronize the TICK register of all physical cores. Refers to Op*enSPARC T2 Programmer's Reference Manual* spec 14.1.5.

TABLE 7-77 ASI CMP Tick Enable Register

Bit	Name	Initial Value	R/W	Description
[63:1]	reserved	0	RO	Reserved.
[0]	tick_enable	0	RW	Set to '1' to enable incrementing of TICK registers in all physical cores.

Its value is preserved across warm reset.

7.6.3.12 ASI Warm Reset Vector Mask Register – ASI_WMR_VEC_MASK(0x90_0114_0018)

(ASI:45 VA:18)

When this register is set to '1' by software, POR, WMR or DBR will be able to be directed to RAM, at location (0x000000020). Refers to DBG spec section 3.2 and appendix 10.3.4.

TABLE 7-78 ASI Warm Reset Vector Mask Register

Bit	Name	Initial Value	R/W	Description
[63:1]	reserved	0	RO	Reserved.
[0]	Wmr_vec_mask	0	RW	Send to TCU for wmr protect.

Its value will be preserved during warm reset.

Please note that POR and WMR are events, not Signals

- Please reference to SUN's CMP spec. for more details on each signal
- Please reference to RST MAS spec. for detailed POR and WMR events

7.7 Appendix A

SSI Software Interface

Addresses within the SSI address range (0xFF_F000_0000 to 0xFF_FFFF_FFFF) are issued to the off-chip SSI interface bus. The only transactions that are supported directly to the SSI interface are:

1, 2, 4, 8 Byte aligned Reads

1, 2, 4, 8 Byte aligned Writes

Since the Boot ROM is predominantly used for instructions, which is explicitly always big-endian, all accesses to the SSI interface bus are treated as big-endian.

1. SSI Register Interface The SSI registers all deal with error handling, so are described in Chapter 12 of the *OpenSPARC T2 Programmer's Reference Manual*, Error Handling.

2. SSI Error Handling TABLE 7-79 describes the SSI's handling of errors. The error indication on read returns is delivered regardless of the ERREN bit, where it is up to the processor to ignore the error or receive it. Logging the error and sending an error interrupt are controlled by the ERREN bit. Note that returning zeros on an I-fetch timeout will tend to cause an illegal instruction trap.

TABLE 7-79 SSI Error Handling

Error	TType	Severity	Logs	Returns	ERREN
SSI Parity Error	Read	Uncorrectable	Just the bit	Data with error indication	Asynch Intr
SSI Parity Error	Write	Uncorrectable	Just the bit	N/A	Asynch Intr
SSI Timeout	Read	Uncorrectable	Just the bit	All Zeros with error indication	Asynch Intr
SSI Timeout	Write	Uncorrectable	Just the bit	N/A	Asynch Intr

3. SSI Interrupts

   SSI generates interrupts for two reasons: either the EXT_INT_L pin was asserted, or an error was detected.

   The external interrupt pin is intended to be used by the FPGA, and has NO ordering protection, meaning when EXT_INT_L is asserted, an interrupt is issued to the IOB, without checking any transactions in flight. The interrupt is delivered to the IOB using the SSI device ID, i.e. (device ID == 2).

   EXT_INT_L is treated as an asynchronous input, meaning the JBI must synchronize it to its internal clock before using it. Also, EXT_INT_L is treated as an edge-triggered interrupt, meaning that JBI will detect a rising edge on the

synchronized signal, and issue an interrupt to the IOB on those rising edges. If the actual use is level-sensitive, software is responsible for querying the FPGA device (or whatever is driving EXT_INT_L), to see if the interrupt is still asserted, at the end of the interrupt handler.

To guarantee being seen, EXT_INT_L must be asserted for at least 4.5 JBUS cycles.

Error interrupts, when enabled, are delivered to the IOB using the error device ID, (device ID == 1).

4. SSI Interface The Serial System Interface (SSI) is defined for to allow microprocessors to access peripherals in a low pin count fashion. The OpenSPARC T2 chip will not directly interface to peripherals but instead will provide a interface that can be easily converted to peripheral protocols by an external Programmable Logic Device (PLD). Isolating the OpenSPARC T2 chip from these peripherals allows the devices to use higher voltage signalling and provides a mechanism for protocol conversion.

   For the purposes of this discussion, some assumptions of the environment will be made. The JBUS will be assumed to run at 200 MHz nominally, although the actual frequency could be somewhat less than 200. In addition the OpenSPARC T2 chip is assumed to interface to either a CPLD or a more complex FPGA. In the former case, the CPLD may just interface to a Flash PROM. In the latter case, the FPGA may include peripherals of its own (e.g. RS232 UART or system management microprocessor) and have a dedicated parallel (8-bit or wider) interface to Flash ROMs and potentially SRAMs. All of these peripherals would be memory mapped into the 256 Megabyte SSI addressable location area (FF_F000_0000 FF_FFFF_FFFF). All devices accessable off the SSI interface will be only targets OpenSPARC T2 will always be the master of the bus.

5. Functional Interface

   The SSI interface includes three pins: SSI_SCK (clock), SSI_MOSI (master out/slave in), and SSI_MISO (master in/slave out). SSI_CLK and SSI_MOSI are outputs of OpenSPARC T2, and SSI_MISO is an input. The SSI_SCK is a free running clock, toggling whenever the on chip JBUS clock is toggling. It is assumed to be nominally 50 Mhz, but is always a divide by 4 or 8 of the JBUS clock.

6. SSI Request

   An SSI request is transmitted on the SSI_MOSI line. It can be either a read command or a write commad. The format of all these requests is 1 start bit, 3 bit command (CMD[2:0]), a 28 bit address, 0-64 bits of data, and a parity bit. The high order (most significant) bit within the command, address and data are always transmitted first, with the low order bit transferred last. Zeros are transmitted as a low voltage value and ones are transmitted as a high value. A start bit is a high value.

   CMD[2] is 0 for write, 1 for read

CMD[1:0] encodes the transaction size as follows:

2'b00 - 1 byte

2'b01 - 2 byte

2'b10 - 4 byte

2'b11 - 8 byte

For every SSI request, a SSI response is expected. A succeeding request can not be sent until the preceding request has had a response. (No command pipelining is supported.)

When OpenSPARC T2 has no request to transfer or is waiting for a response, the SSI_MOSI line is held in the low voltage state.

The parity bit is set such that the number of 1s in the start bit, the command, the address, any data bits, and the parity bit is an even number.

7. SSI Response

   An SSI response is received on the SSI_MISO line. It can be either a read response which must contain data or a write response which must contain no data. The format of a read response is 1 start bit, 8-64 data bits, and 1 parity bit. The format of a write response is 1 start bit and 1 parity bit. The high order (most significant) bit within the data are always transmitted first, with the low order bit transferred last. Zeros are transmitted as a low voltage value and ones are transmitted as a high value. A start bit is a high value.

   The parity bit is set such that the number of 1s in the start bit, any data bits, and the parity bit is an even number. This means a write response is two 1's in consecutive cycles.

   When the target has no response to transfer or is processing a request, the SSI_MISO line is held in the low voltage state.

   Electrical Interface

   The SSI_SCK, SSI_MOSI, SSI_MISO, and EXT_INT_L Signals will be HSTL Signals at 1.5V. Care must be taken on the input so that overshoot doesn't exceed the 1.5V VDD for long enough to induce gate oxide breakdown for the CO27.C process. (See the signal ERS for voltage levels and currents.)

   When driving OpenSPARC T2 will drive SSI_MOSI for 3 JBUS cyles prior to a SSI_SCK rising edge and hold SSI_MOSI for 1 JBUS cycle after the SSI_SCK rising edge. When receiving, OpenSPARC T2 will wait 3 JBUS cycles after a rising SSI_SCK edge to sample the input line.

7.8 Appendix B

The following is the Sii/NCU interface data format which results in the SIISYN syndrome register.

siisyn_data[63:0] comes from SII to NCU, 4-bit at a time (see following timing diagram) ,

starting 1st transfer in bit[3:0], then bit[7:4], and so on

```
Siisyn_data[39:0] = PA,
siisyn_data[55:40] = ctag,
siisyn_data[61] = niud_pe,
Siisyn_data[60] = niua_pe,
siisyn_data[59] = niuctag_ue,
siisyn_data[58] = dmud_pe,
Siisyn_data[57] = dmua_pe,
siisyn_data[56] = dmuctag_ue,
```

NCU will encode siisyn_data[61:56] to 3-bit Etag, siisyn[58:56] as in TABLE 7-80

TABLE 7-80 SII/NCU Interface Data Format

siisyn_data[61]	“000001”	“000001”	“000001”	“000001”	“000001”	“000001”
Etag[2:0]	“001”	“111”	“101”	“000”	“100”	“110”

FIGURE 7-17 SII to NCU Eror Strobe

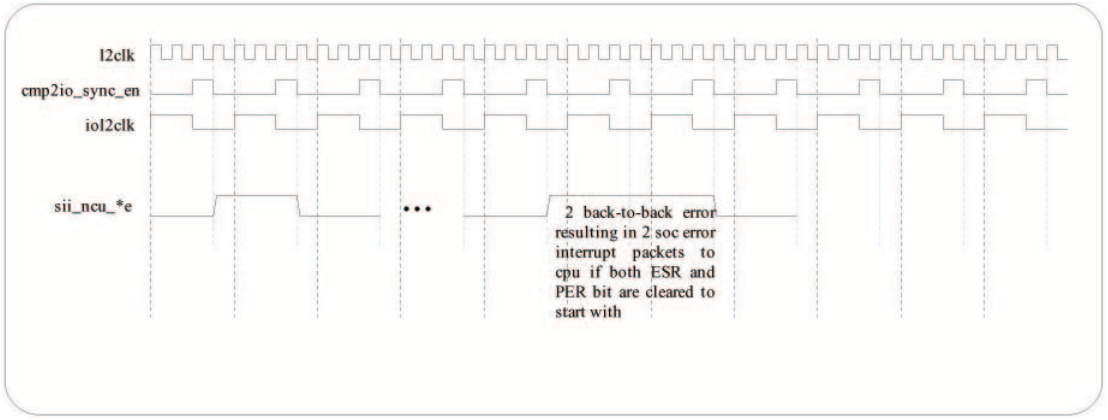


FIGURE 7-18 Sii to NCU Error Syndrome

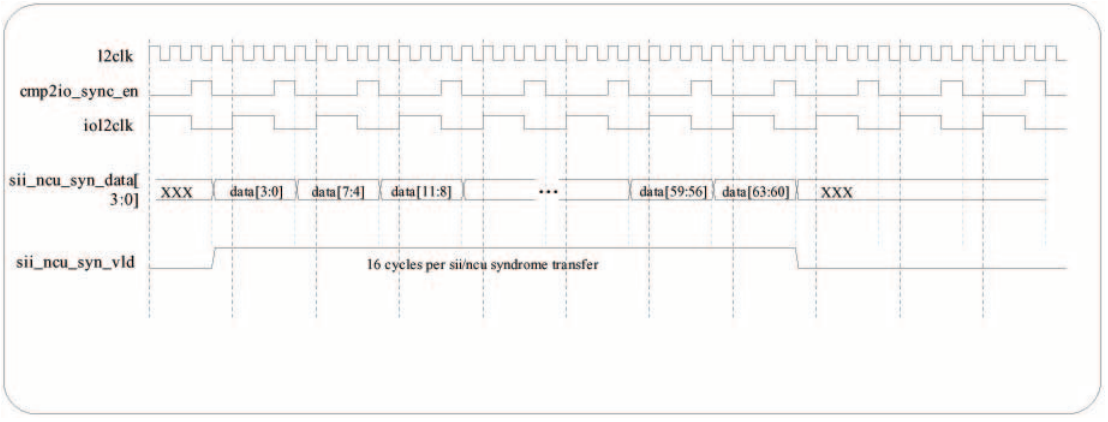
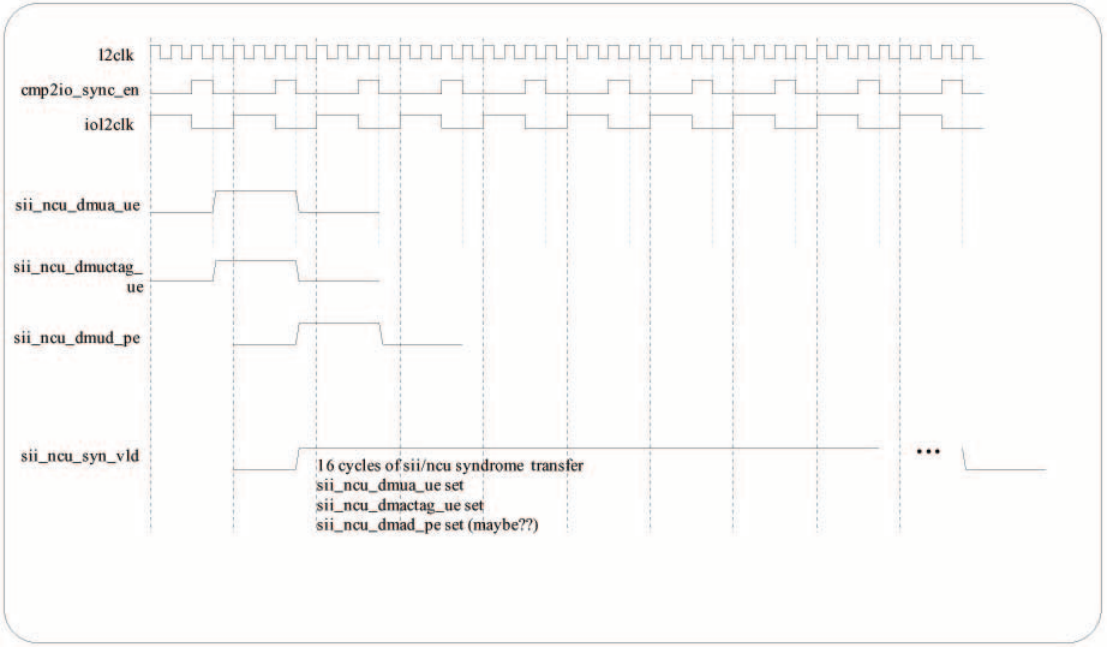


FIGURE 7-19 SII to NCU Error Strobe and Syndrome Transfer Example


l2clk
cmp2io_sync_en
iol2clk
sii_ncu_dmua_ue
sii_ncu_dmuctag_ue
sii_ncu_dmud_pe
sii_ncu_syn_vld
16 cycles of sii/ncu syndrome transfer
sii_ncu_dmua_ue set
sii_ncu_dmactag_ue set
sii_ncu_dmad_pe set (maybe??)

CHAPTER 8

Data Management Unit (DMU)

This chapter contains the following sections:

8.1 Overview

The OpenSPARC T2 PCI-Express subsystem leverages the Data Management Core (DMC) from VSP Fire ASIC for PCI-Express Packet processing. With the additional glue logic (DSN block) between Fire DMC IP, N2 SIU (system interface unit) and N2 NCU (non-cacheable unit), the DSN block plus Fire DMC constitutes the Data Management Unit (DMU) in the OpenSPARC T2 PCI-Express Subsystem.

This specification document the high level DMU function.

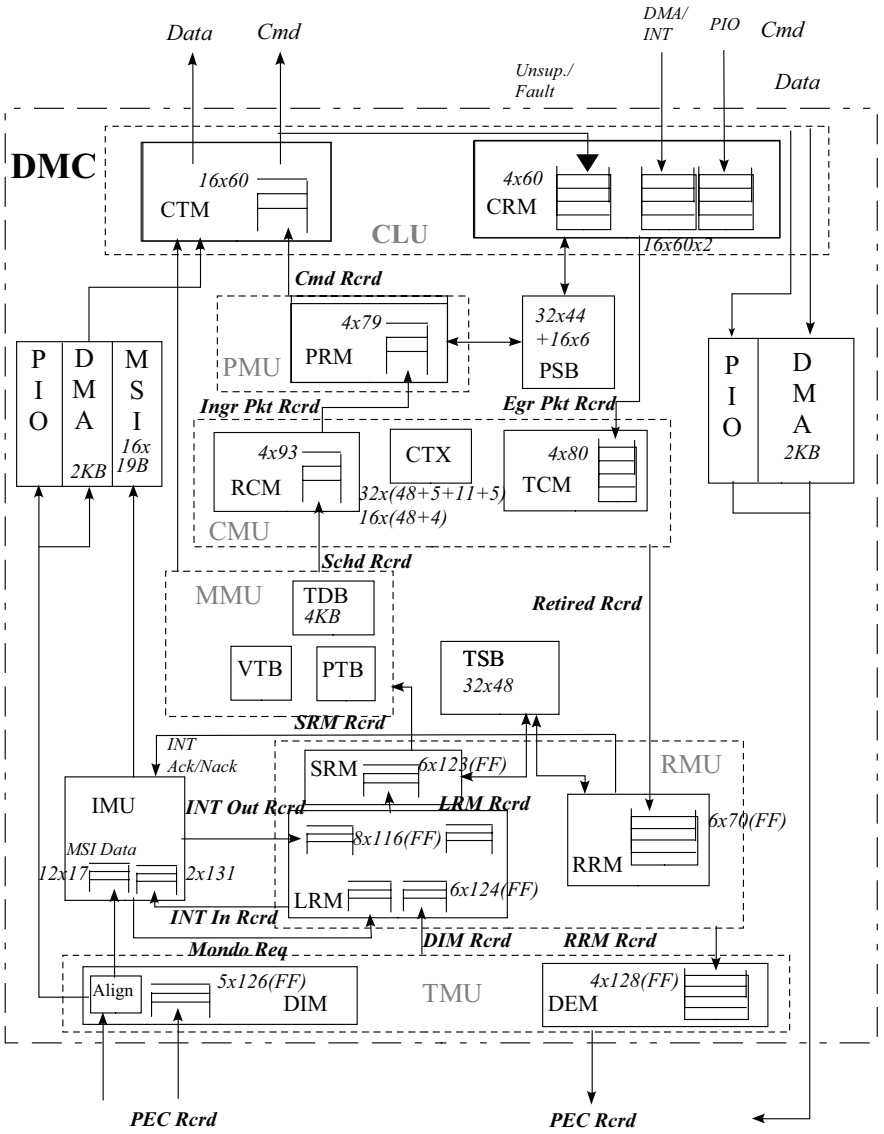
The DMU is responsible for managing and directing all command and data flows from/to PCI-Express Unit (PEU), System Interface Unit (SIU), and Non-Cacheable Unit (NCU). The DMU has 3 primary external interfaces, one to the SIU, one to the NCU and one to the PEU.

The DMU manages Transaction Layer Packet (TLP) to/from the PEU and maintains the same ordering as from the PEU and then to the SIU. For maintaining ordering between PEU and SIU, the DMU requires the policy that has PIO reads pulling DMA writes to completion. When the PEU issues complete TLP transactions to the DMU, the DMU segments the TLP packet into multiple cacheline oriented SIU commands and issues them to the SIU. The DMU also queues the response cachelines from SIU, reassembly the multiple cachelines into one TLP packet with maximal payload size. Furthermore, the DMU accepts / queues the PIO transactions requests from NCU, and coordinates with the appropriate destination, to which the address and data will be sent.

The DMU encapsulates the functions necessary to resolve a virtual PCI-Express packet address into a L2 cacheline physical address which can be presented on the SIU interface. The DMU also encapsulates the functions necessary to interpret PCI-Express Message Signaled Interrupts, Emulated INTX Interrupts and provides the functions to post interrupt events to queues managed by software in main memory and generates the Solaris Interrupt Mondo to notify software. The DMU decodes INTACK and INTNACK from interrupt targets and conveys the information to the Interrupt Function so it can move on to service the next interrupt if any (for INTACK) or replay the current interrupt (for INTNACK).

8.1.1 DMC Block Diagram

FIGURE 8-1 DMC Block Diagram



8.1.2 Abbreviation

DMU - Data Management Unit

SIU - System Interface Unit

NCU - Non-Cacheable Unit

PEU – PCI-Express Unit

CLU – Cache Line Unit

CTM – Cacheline Transmit Manager

CRM – Cacheline Receive Manager

PMU – Packet Manager Unit

CMU – Context Manager Unit

IOMMU – IO Memory Management Unit

IMU – Interrupt and Message Unit

RMU – Record Manager Unit

TMU – Transaction Manager Unit

DIU – Data-In Unit

DOU – Data-Out Unit

TSB – Transaction ScoreBoard

PSB – Packet ScoreBoard

VTB – Virtual Tag Buffer

TDB – Translation Data Buffer

MSI – Message Signal Interrupt

8.1.3 General DMC IP Ingress Pipeline Operations

1. TMU dequeues PEU TLP Record from input Queue.
2. TMU parses PEU TLP Record - extract record contents, Data.
3. TMU moves write data to DATA Pool if necessary.
4. RMU builds/Installs Transaction entry on Transaction scoreboard (TRN SCBD)

5. RMU build Schedule Record and enqueue Schedule Record to IOMMU
6. IOMMU Manager dequeues Schedule Record, builds VAR record (if necessary), enqueues VAR record on VAR Queue, dequeues VAR record, does VA-> PA translation and returns results in PhyAD Q, Merges PhyAD from PhyAD Q into Schedule Record, enqueues Schedule Record to Context Manager
7. Context Receive Manager dequeues Schedule Record, and installs context in current Context (CNTXT) Lists
8. Context Manager fetches next Context from CNTXT list, builds Packet Record, enqueues Packet Record to Packet Receive Manager
9. Packet Receive Manager dequeues Packet Record, Breaks up Packet Record into cacheline oriented record, builds a Cacheline Command Record, enqueues Cacheline Command Record to Cacheline Transmit Manager, builds/updates Packet Scoreboard entry.
10. Cacheline Transmit Manager dequeues Cacheline Command Record, enqueues cacheline Command Record onto DSN interface, pulls data from DATA pool and enqueues data on outgoing data queue to DSN.

8.1.4 General Egress Pipeline Operations

DMA Rd Data Responses:

1. CRM (Cacheline Receive Manager) dequeues a DATA return from the DMA read request, builds Packet Record, enqueues Packet Record to TCM (Transmit Context Mgr.) Queue, updates PKT scoreboard
2. TCM (Transmit Context Manager) dequeues Packet Record, matches the context to a current Context (CNTXT) list entry, processes the context, builds a Retire Record, enqueues Retire Record to Retire Record Manager, marks the Context as done if all Packet Records have been returned, retires Context
3. RRM (Retire Record Manager) dequeues Retire Record, updates Transaction Scoreboard, builds and issues TLP Record to Transaction Manager.
4. Transaction Manager dequeues TLP Record, builds a PEU Record enqueues PEU Record to PEU Egress Interface Layer HDR FIFO with address in DMC DATA Pool
5. PEU Egress Interface Layer moves data from DMU Data Pool to VC Data Buffer

Commands (PIO):

1. CRM dequeues a PIO Record from the DSN, builds Packet Record, enqueues Packet Record to Transmit Context Mgr Queue, updates PKT scoreboard.

2. Transmit Context Manager dequeues Packet Record, bypasses Context, builds and enqueues Retire Record to Retire Record Manager.
3. Retire Record Manager dequeues Retire Record, builds and issues TLP Record to Transaction Manager.
4. Transaction Manager dequeues TLP Record, builds a PEU Record and enqueues the record into the Egress Interface Layer HDR FIFO.
5. Egress Interface Layer moves data from DMC PIO Pool to VC Data Buffer

8.2 Functional Description of DMC Sub-blocks

The DMC contains several groups of functions, including Cache Line Unit (CLU), Packet Manager Unit (PMU), Context Manager Unit (CMU), IO Memory Management Unit (IOMMU), Record Manager Unit (RMU), Interrupt and Message Unit (IMU), Transaction Manager Unit (TMU), Transaction Scoreboard (TSB), Packet Scoreboard (PSB), Data Buffers (DIU and DOU). The following sections describe the architecture, functionality and change requirement of each groups.

8.3 Transaction Manager Unit (TMU)

8.3.1 TMU Function Description:

The TMU interfaces with the Interface Layer Unit (ILU) of the PEU to manage the TLPs ingress and egress flows. It consists of two sub-blocks, Data Ingress Manager (DIM) and Data Egress Manager (DEM).

8.3.1.1 Data Ingress Manager (DIM)

In the ingress direction, ILU pushes header record to a record FIFO residing in DIM. These records include a pointer to a packet payload in the IDB of PEU. The DIM manages the DIB buffer space allocation on a cacheline basis. It aligns a packed 16-byte wide data pulled from IDB to a non-packed cacheline oriented data format, calculates byte masks, and pushes the data and byte mask to DIU.

Meanwhile, DIM sends release records to ILU when it pulls data out of IDB. The DIM identifies a Message Signaled Interrupt (MSI) type from a DMA Mwr. On a MSI operation, it checks the data parity and pushed the result along with the payload to IMU MSI data FIFO, and a reformed header record is pushed to LRM record FIFO.

The records out of DIM is in strict order as the records into DIM. For non-MSI records with payload, the record will be pushed into LRM record FIFO before the associated payload is transferred from IDB in PEU to DIU in DMC. DIM passes DIU DMA write buffer and PIO read completion buffer write pointers to CLU. CLU knows if the payload associated with certain records is ready by comparing the write pointers to its own data buffer read pointers.

For MSI, DIM pushes an MSI associated payload directly into IMU MSI data FIFO and passes the MSI record to IMU via LRM, and the payload can not arrive IMU after the associated record.

8.3.1.2 Data Egress Manager (DEM)

In the egress direction, the header records are pushed by the RMU to a record FIFO residing in DEM. DEM computes the full 64-bit address from a 40-bit encoded address for PIO memory 64-bit address access. It pushes the reformatted header records down to the EIL record FIFO if the FIFO is not full.

8.3.1.3 MSI-X Support:

To support MSI-X, the datapath width between DIM and IMU MSI data fifo need to be increased from 16 bits to 32 bits.

8.4 Interrupt Message Unit (IMU)

8.4.1 IMU Function Description:

8.4.1.1 Definition of Terms

- Event Queue - A ring buffer in memory defined by a physical base address which is cacheline aligned, it's size in cachelines, a head pointer, and a tail pointer.

- Event Queue Interrupt - A type of Mondo interrupt to notify software that a given event queue has entries in it which need to be processed. This type of Mondo interrupt can only be mapped to Solaris interrupt numbers 24-59.
- Event Queue Number - A number between 0 and 35 which is used to uniquely identify which given event queue an event queue write is destined for.
- Event Queue Write - A 64 byte write to memory (virtual or physical) which is caused by the reception of a MSI/Message from PEU. The IMU actually only writes 16 bytes into the DMC DIU, the remaining bytes are zero filled when CLU dispatches the write packets to DSN.
- Inband Interrupt - A sub set of I/O bus interrupts which are received by OpenSPARC T2 PEU via the normal flow of traffic on the PCI-Express. MSI s and INT x emulated messages fall into the category.
- Internal Interrupt - A type Mondo interrupt which is generated internally by N2 IO subsystem. They are caused by errors which occur within the chip. Each unit has the ability to generate 1 internal interrupt which are then mapped to Solaris Interrupt numbers 62 (DMU) and 63(PEU).

8.4.1.2 IMU Functional Descriptions

The IMU handles MSIs (Message Signal Interrupts), PCIE messages, INTx emulation interrupts, and internal interrupts (error and event). In response to receiving one of the above transactions requests, the IMU must generate a response which will be either a null record, an event queue write record or a Solaris interrupt Mondo record. It also generates properly formatted 16 bytes of data required with each transaction.

IMU uses event queues to queue up MSI's and valid PCIE messages received which require software notification. An event queue is tied to a specific processor and generates only one outstanding Solaris Mondo interrupt for one or more than one write to the event queue.

MSIs are mandatory in PCI-Express. They are queued in one or more event queues located in system memory. Each event queue generates only one outstanding Solaris Mondo interrupt. When the MSI record is dequeued off the In Interrupt Record Queue and associated DATA is dequeued off of the MSI data queue, the MSI data is decoded and EQ state is looked up. If EQ is available, a new header record is formed and enqueued into the Out Interrupt Record Queue. At the same time, the data is sent to DIU along with the corresponding the parity and byte enables generated by IMU. The process of PCI-Express Message is similar to MSI except no associated data. When MSI's and Messages pass through the command/header pipeline, an EQ write is performed if no error conditions occur. The tail pointer for the EQ is incremented automatically to prepare for the next EQ write. When a difference is seen between the head and tail pointers, a Mondo will be generated for the event

queue and sent to the Mondo Request Queue in LRM. The Event Queue Mondo record re-enters the IMU via the In Interrupt Record Queue at a later point. This Event Queue Mondo is then enqueued onto the Out Interrupt Record Queue.

INTx emulation interrupts trigger the state machine transition to generate a Mondo record. Then the Mondo record is enqueued to the Mondo Request Queue in the LRM. After being tagged by LRM, the Mondo record flows back to IMU to be processed via In Interrupt Record Queue, and then is enqueued onto the Out Interrupt Record Queue as other type of records.

Internal interrupts for error and status notification will generate one or more Solaris interrupt Mondo vectors. They will not use the event queues since some of the errors would be detected when trying to write to an event queue.

8.4.1.3 IMU Mondo State Machine

IMU uses a level sensitive interrupt mechanism and is governed by a certain set of rules which may be found below. Also please refer to the state machine diagram.

- A host bus interrupt can be in one of three states: IDLE, RECEIVED, or PENDING.

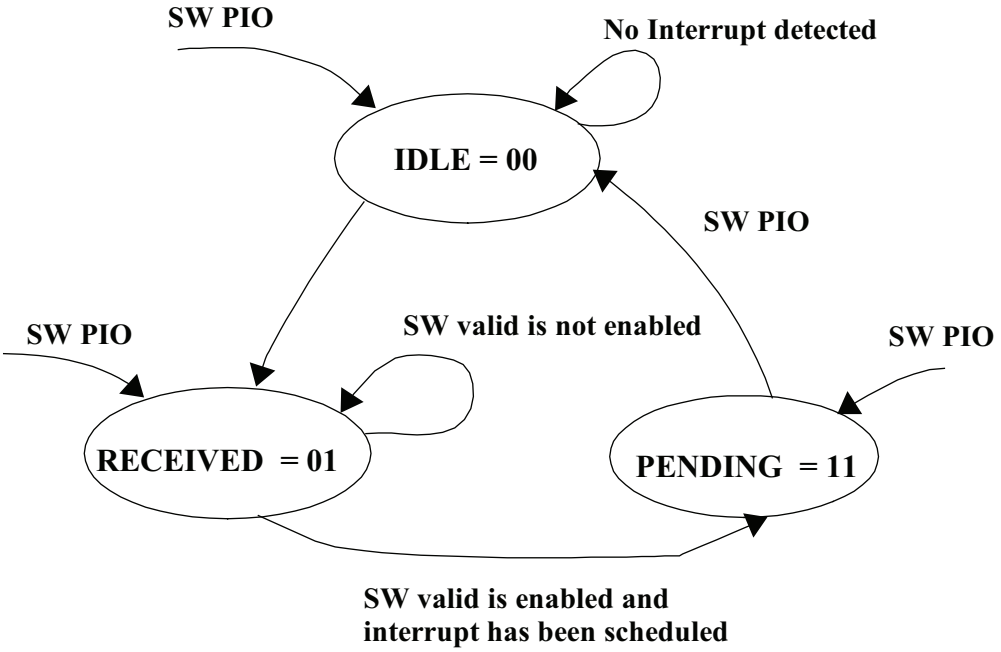
  a. IDLE - represents the state where no interrupts have been reported.

  b. RECEIVED - indicates that an interrupt has been detected by the hardware and should be delivered to the processor if/when the valid bit is set in its mapping register.

  c. PENDING - represents the state when the interrupt has been queued to be or has been sent to the processor to be handled.

- A detection of an interrupt by hardware when the current interrupt state is IDLE causes a state transition from IDLE to RECEIVED.
- Any subsequent detection of the same interrupt by hardware is dropped until software resets the state machine back to IDLE.
- If the valid bit for a given interrupt in the RECEIVED state is enabled and the hardware has scheduled that interrupt for transmission to the processor, a state transition occurs from RECIEVED to PENDING.
- At no point can hardware make any other transitions in the state machine besides the previously two afore mentioned transitions.
- The state for each interrupts can be set to any desired state by software.
- If SW sets the state machine into a given state, all of the HW arcs for that state are still valid and any events and or state transitions which should occur in that state will occur.

FIGURE 8-2 IMU Mondo State Machine



8.4.1.4 PCI-Express/PCI-X/PCI MSI Capability Structure

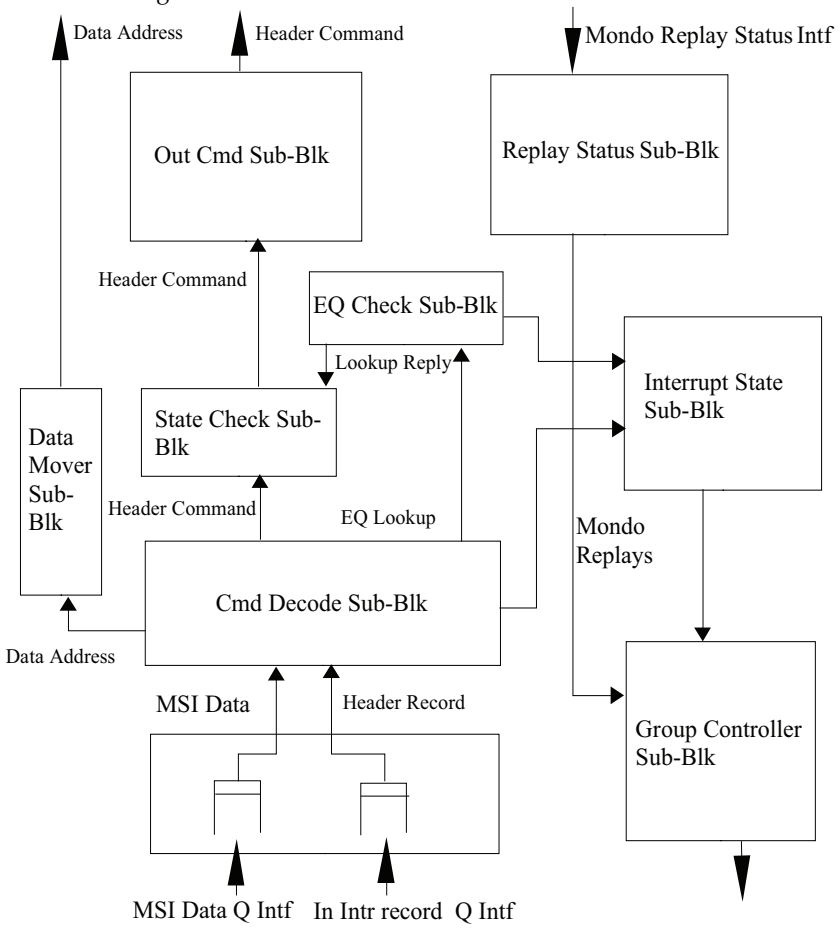
The capabilities mechanism in PCI-Express/PCI-X/PCI end device is used to identify and configure a MSI capable device. The message capability structure is illustrated below. Each device function that supports MSI (in a multi-function device) must implement its own MSI capability structure.

To request service, an MSI function writes the contents of the Message Data register to the address specified by the contents of the Message Address register (and, optionally, the Message Upper Address register for a 64-bit message address).

Compatability Structure for 32-bit Message Address		
31 ---- 16	15 ----- 8	7 ----- 0
Message Control	Nxt Ptr	Cap ID
	Message Address	
		Message Data

Compatability Structure for 64-bit Message Address		
31 ---- 16	15 ----- 8	7 ----- 0
Message Control	Nxt Ptr	Cap ID
Message Address		
Message Upper Address		
Message Data		

FIGURE 8-3 IMU Block Diagram



8.4.1.5 IMU Mondo INO Mapping Table

TABLE 8-1 IMU Mondo INO Mapping

INO	Function
INO's 0-19	Reserved
INO 20-23	4 interrupts for PCI Express INTx Emulation -20 INTA -21 INTB -22 INTC -23 INTD

TABLE 8-1 IMU Mondo INO Mapping *(Continued)*

INO 24 – 59	36 Event Queue Interrupts
INO 60 – 61	Reserved
INO 62	DMU Internal Interrupt
INO 63	PEU Internal Interrupt

8.4.1.6 IMU CSRs Change List

Interrupt Mapping Registers (0x601000 – 0x601150) 42 consecutive registers, one for each Mondo)

TABLE 8-2 Interrupt Mapping Registers

Field	Bits	Reset Name	Reset Value	Type	Description
MDO_MODE	[63]	rst_l	0x0	RW	This bit is used to select which of the 2 mondo formats the mondo will use. A value of 1 = Data Baring Mondo. A value of 0 = Non Data Baring Mondo (Normal Mondo). The value of this bit, will be used as bit 63 of the first data word in the Mondo vector. In general, EQ mondo s should have this bit set to 1 and non EQ Mondos should set this bit to 0.
RESERVED	[62:32]	rst_l	0x0	RW	Reserved field
V	[31]	rst_l	0x0	RW	Valid bit: When set to 0, interrupt will not be dispatched to Core. Has no other impact on interrupt state.
Thread_ID	[30:25]	rst_l	0x0	RW	Thread ID of the core that this interrupt will be sent to.
RESERVED	[24:10]				Reserved field
INT_CNTRL_ NUM	[9:6]	rst_l	0x0	RW	Interrupt Controller Number. This is used to select which Interrupt controller will issue the interrupt. This is a 1-hot value only 1 bit may be selected at a time. Valid Values are as follows: 0000 - No controller selected 0001 - Interrupt Controller 0 0010 - Interrupt Controller 1 0100 - Interrupt Controller 2 1000 - Interrupt Controller 3 If other values are programmed this is a programming error the results are undefined
RESERVED	[5:0]				Reserved field

TABLE 8-3 Interrupt Clear Registers (0x601400 – 0x601440) 42 consecutive registers, one for each Mondo

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:2]				Reserved field
INT_STATE	[1:0]	rst_l	0x0	RW	Writing of the register, the value of the lower two bits are used to control the state bits for the interrupt state machine associated with this interrupt. The following values my be written: 00 - Set the state machine to IDLE state. 01 - Set the state machine to RECEIVED state. 10 - Reserved, If this value is used it is a programming error. The results are undefined. 11 - Set the state machine to PENDING state. When reading from this register, the actual state of the associated interrupt state machine are read. The legal values are the same as listed above.

TABLE 8-4 Interrupt Retry Timer Register (0x601A00)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:25]				Reserved field
Limit	[1:0]	rst_l	0x0	RW	Limit the retry interval in clock cycles (N2 IO Clock Frequency)

TABLE 8-5 Interrupt State Status Register I (0x601A10)

Field	Bits	Reset Name	Reset Value	Type	Description
STATE	[63:0]	rst_l	0x0	R	State Values for Mondos 0 through 31 Each state is 2 bits in the register with the MSB being the 2nd bit of Mondo 31 and the LSB being the 1st bit of Mondo 0.

TABLE 8-6 Interrupt State Status Register II (0x601A18)

Field	Bits	Reset Name	Reset Value	Type	Description
STATE	[63:0]	rst_l	0x0	R	State Values for Mondos 32 through 63 Each state is 2 bits in the register with the MSB being the 2nd bit of Mondo 63 and the LSB being the 1st bit of Mondo 32.

TABLE 8-7 INTX Status Register (0x0060B000)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:4]				Reserved field
INT_A	[3]	rst_l	0x0	R	INT A Status 0= No INTX 1= INTX This register will be set when an assert INT A message is received and will be cleared when a deassert INT A message is received or when cleared via the INT A Clear Register by software
INT_B	[3]	rst_l	0x0	R	INT B Status 0= No INTX 1= INTX This register will be set when an assert INT B message is received and will be cleared when a deassert INT B message is received or when cleared via the INT B Clear Register by software
INT_C	[3]	rst_l	0x0	R	INT C Status 0= No INTX 1= INTX This register will be set when an assert INT C message is received and will be cleared when a deassert INT C message is received or when cleared via the INT C Clear Register by software
INT_D	[3]	rst_l	0x0	R	INT D Status 0= No INTX 1= INTX This register will be set when an assert INT D message is received and will be cleared when a deassert INT D message is received or when cleared via the INT D Clear Register by software

TABLE 8-8 INT A Clear Register (0x0060B008)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:1]				Reserved field
CLR	[0]	rst_l	0x0	RW1C	Write 0 = has no effect, Write 1 will clear the INT A bit of the INTX Status Register. When reading, the value of the INT A bit from the INTX Status Register will be returned

TABLE 8-9 INT B Clear Register (0x0060B010)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:1]				Reserved field
CLR	[0]	rst_l	0x0	RW1C	Write 0 = has no effect, Write 1 will clear the INT B bit of the INTX Status Register. When reading, the value of the INT B bit from the INTX Status Register will be returned

TABLE 8-10 INT C Clear Register (0x0060B018)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:1]				Reserved field
CLR	[0]	rst_l	0x0	RW1C	Write 0 = has no effect, Write 1 will clear the INT C bit of the INTX Status Register. When reading, the value of the INT C bit from the INTX Status Register will be returned

TABLE 8-11 INT D Clear Register (0x6010B018)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:1]				Reserved field
CLR	[0]	rst_l	0x0	RW1C	Write 0 = has no effect, Write 1 will clear the INT D bit of the INTX Status Register. When reading, the value of the INT D bit from the INTX Status Register will be returned

TABLE 8-12 Event Queue Base Address Register (0x00610000)

Field	Bits	Reset Name	Reset Value	Type	Description
ADDRESS	63:19	rst_l	0x0	RW	EQ Base Address, 512K Aligned This address has to a properly formatted physical or virtual address. Meaning if this address is suppose to be bypass it needs the upper 14 address bits [63:50] set to all 1 s, address bits [49:39] set to zero. The lower bits of the address [38:18] are used as the cacheable physical address on L2$. For a virtual address bit 63 need to be zero, bits [62:32] are don't care, bits [31:18] used to access the IOMMU.
RESERVED	18:0				Reserved field

TABLE 8-13 Event Queue Control Set Registers (0x00611000 – 0x00611118) - 36 consecutive registers, one for each EQ

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:58]				Reserved field
ENOVERR	[57]	rst_l	0x0	L	0-no action;1-Set OVERR bit, A read of this register is not allowed. This bit should only be set if the EQ is currently in the ACTIVE state. Setting this bit when the EQ is IDLE will cause undetermined results.
RESERVED	[56:45]				Reserved field
EN	[44]	rst_l	0x0	L	0-no action;1-Enable EQ, EQ will be running when STATE = ACTIVE, A read of this register is not allowed. This bit should only be written when the EQ is currently IDLE. If the EQ is not in the IDLE state this operation will have no effect on the state of the EQ.
RESERVED	[43:0]				Reserved field

TABLE 8-14 Event Queue Control Clr Registers (0x00611200 – 0x00611318) 36 consecutive registers, one for each EQ

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:58]				Reserved field
COVERR	[57]	rst_l	0x0	L	0-no action; 1-Clear OVERR bit, A read of this register is not allowed.
RESERVED	[56:48]				Reserved field
E2I	[47]	rst_l	0x0	L	0-no action;1-Go from ERROR to IDLE, A read of this register is not allowed. This bit should only be written when the EQ is currently in the error. If the EQ is not in the ERROR state this operation will have no effect on the state of the EQ.
RESERVED	[46:45]				Reserved field
DIS	[44]	rst_l	0x0	L	0-no action; 1-Disable EQ. A read of this register is not allowed. This bit should only be set if the EQ is currently in the ACTIVE state. If the EQ is not in the ACTIVE state this operation will have no effect on the state of the EQ.
RESERVED	[43:0]				Reserved field

TABLE 8-15 Event Queue State Register (0x00611400 – 0x00611518) - 36 consecutive registers, one for each EQ

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:3]				Reserved field
STATE	[2:0]	rst_l	0x0	L	Event Queue State 001-IDLE, 010-ACTIVE, 100-ERROR

TABLE 8-16 Event Queue Tail Register – (0x00611600 – 0x00611718) - 36 consecutive registers, one for each EQ

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:58]				Reserved field
OVERR	[57]	rst_l	0x0	R	1-EQ Overflow occurred.
RESERVED	[56:7]				Reserved field
TAIL	[6:0]	rst_l	0x0	RW	Value of the current HW tail pointer. In normal operation it is read by SW and written by HW.

TABLE 8-17 Event Queue Head Registers – (0x00611800 – 0x611918) - 36 consecutive registers, one for each EQ

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:7]				Reserved field
TAIL	[2:0]	rst_l	0x0	RW	EQ Head Pointer. Initialize by s/w, written by s/w during operation.

TABLE 8-18 MSI Mapping Registers - (0x00620000 – 0x006207f8) - 256 consecutive registers, one for each MSI

Field	Bits	Reset Name	Reset Value	Type	Description
V	[63]	rst_l	0x0	RW	0 - Not Valid, A received MSI of this number will be treated as an error. 1 - Valid, A received MSI of this number will be routed to the EQ specified in the eqnum field
EQWR_N	[62]	rst_l	0x0	R	0 - OK to write to, a received MSI of the number will be will be sent to the EQ specified. 1 - MSI already in EQ, received MSI of the number will be will be treated as a duplicate. S/W must clear this bit BEFORE calling the clients interrupt handler.
RESERVED	[61:6]				Reserved field
EQNUM	[5:0]	rst_l	0x0	RW	Event Queue Number

TABLE 8-19 MSI Clear Registers – (0x00628000 – 0x006287f8) - 256 consecutive registers, one for each MSI

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63]				Reserved field
EQWR_N	[62]	rst_l	0x0	RW1C	Write 0 = has no effect. Write 1 = will clear the EQWR_N bit of the MSI Mapping Register. When reading, the value of the EQWR_N bit from the MSI Mapping Register will be returned
RESERVED	[61:0]				Reserved field

TABLE 8-20 Interrupt Mondo Data 0 Register – (0x0062c000)

Field	Bits	Reset Name	Reset Value	Type	Description
DATA	[63:6]	rst_l	0x0	RW	Data 0 word, bits 63:6 of mondo used for a data baring mondos with the mode bit set to 1.
RESERVED	[5:0]				Reserved field

TABLE 8-21 Interrupt Mondo Data 1 Register – (0x0062c008)

Field	Bits	Reset Name	Reset Value	Type	Description
DATA	[63:0]	rst_l	0x0	RW	Data 1 word of mondo used for data baring mondos with the mode bit set to 1.

TABLE 8-22 ERR COR Mapping Register (0x00630000)

Field	Bits	Reset Name	Reset Value	Type	Description

TABLE 8-22 ERR COR Mapping Register (0x00630000) *(Continued)*

V	[63]	rst_l	0x0	RW	0 - Not Valid, A received message of this type will be treated as an error. 1 - Valid, A received message of this type will be routed to the EQ specified in the eqnum field
RESERVED	[62:6]				Reserved field
EQNUM	[5:0]	rst_l	0x0	RW	Event Queue Number

TABLE 8-23 ERR NONFATAL Mapping Register (0x00630008)

Field	Bits	Reset Name	Reset Value	Type	Description
V	[63]	rst_l	0x0	RW	0 - Not Valid, A received message of this type will be treated as an error. 1 - Valid, A received message of this type will be routed to the EQ specified in the eqnum field
RESERVED	[62:6]				Reserved field
EQNUM	[5:0]	rst_l	0x0	RW	Event Queue Number

TABLE 8-24 ERR FATAL Mapping Register (0x00630010)

Field	Bits	Reset Name	Reset Value	Type	Description
V	[63]	rst_l	0x0	RW	0 - Not Valid, A received message of this type will be treated as an error. 1 - Valid, A received message of this type will be routed to the EQ specified in the eqnum field
RESERVED	[62:6]				Reserved field
EQNUM	[5:0]	rst_l	0x0	RW	Event Queue Number

TABLE 8-25 PM PME Mapping Register (0x00630018)

Field	Bits	Reset Name	Reset Value	Type	Description
V	[63]	rst_l	0x0	RW	0 - Not Valid, A received message of this type will be treated as an error. 1 - Valid, A received message of this type will be routed to the EQ specified in the eqnum field
RESERVED	[62:6]				Reserved field
EQNUM	[5:0]	rst_l	0x0	RW	Event Queue Number

TABLE 8-26 PME To ACK Mapping Register (0x00630020)

Field	Bits	Reset Name	Reset Value	Type	Description
V	[63]	rst_l	0x0	RW	0 - Not Valid, A received message of this type will be treated as an error. 1 - Valid, A received message of this type will be routed to the EQ specified in the eqnum field
RESERVED	[62:6]				Reserved field
EQNUM	[5:0]	rst_l	0x0	RW	Event Queue Number

TABLE 8-27 IMU Error Log Enable Register (0x00631000)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[62:15]				Reserved field
SPARE_LOG_EN	[14:10]	por_l	0x1	RW	Spare Error, Error Log Enable Bits
EQ_OVER_LOG_EN	[9]	por_l	0x1	RW	EQ Overflow Error, Error Log Enable Bit
EQ_NOT_EN_LOG_EN	[8]	por_l	0x1	RW	EQ Not Enabled, Error Log Enable Bit
MSI_MAL_ERR_LOG_EN	[7]	por_l	0x1	RW	Malformed MSI, Error Log Enable Bit
MSI_PAR_ERR_LOG_EN	[6]	por_l	0x1	RW	MSI Data Parity Error, Error Log Enable Bit

TABLE 8-27 IMU Error Log Enable Register (0x00631000) *(Continued)*

PMEACK_MES_NOT_EN_LOG_EN	[5]	por_l	0x1	RW	PME to ACK Message Not Enabled, Error Log Enable Bit
PMPME_MES_NOT_EN_LOG_EN	[4]	por_l	0x1	RW	PM PME Message Not Enabled, Error Log Enable Bit
FATAL_MES_NOT_EN_LOG_EN	[3]	por_l	0x1	RW	Fatal Message Not Enabled, Error, Log Enable Bit
NONFATAL_MES_NOT_EN_LOG_EN	[2]	por_l	0x1	RW	Non Fatal Message Not Enabled, Error Log Enable Bit
COR_MES_NOT_EN_LOG_EN	[1]	por_l	0x1	RW	Correctable Message Not Enabled, Error Log Enable Bit
MSI_NOT_EN_LOG_EN	[0]	por_l	0x1	RW	MSI Not Enabled, Error Log Enable Bit

TABLE 8-28 IMU Interrupt Enable Register (0x00631008)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:47]				Reserved field
SPARE_S_INT_EN	[46:42]	rst_l	0x0	RW	Spare Error, Secondary Interrupt Enable Bits
EQ_OVER_S_INT_EN	[41]	rst_l	0x0	RW	EQ Overflow Error, Secondary Interrupt Enable Bit
EQ_NOT_EN_S_INT_EN	[40]	rst_l	0x0	RW	EQ Not Enabled, Secondary Interrupt Enable Bit
MSI_MAL_ERR_S_INT_EN	[39]	rst_l	0x0	RW	Malformed MSI, Secondary Interrupt Enable Bit
MSI_PAR_ERR_S_INT_EN	[38]	rst_l	0x0	RW	MSI Data Parity Error, Secondary Interrupt Enable Bit
PMEACK_MES_NOT_EN_S_INT_EN	[37]	rst_l	0x0	RW	PME to ACK Message Not Enabled, Secondary Interrupt Enable Bit
PMPME_MES_NOT_EN_S_INT_EN	[36]	rst_l	0x0	RW	PME Message Not Enabled, Secondary Interrupt Enable Bit
FATAL_MES_NOT_EN_S_INT_EN	[35]	rst_l	0x0	RW	Fatal Message Not Enabled, Secondary Interrupt Enable Bit
NONFATAL_MES_NOT_EN_S_INT_EN	[34]	rst_l	0x0	RW	Fatal Message Not Enabled, Secondary Interrupt Enable Bit

TABLE 8-28 IMU Interrupt Enable Register (0x00631008) *(Continued)*

COR_MES_NOT_EN_S_INT_EN	[33]	rst_l	0x0	RW	Correctable Message Not Enabled, Secondary Interrupt Enable Bit
MSI_NOT_EN_S_INT_EN	[32]	rst_l	0x0	RW	MSI Not Enabled, Secondary Interrupt Enable Bit
RESERVED	[31:15]				Reserved field
SPARE_P_INT_EN	[14:10]				Spare Error, primary Interrupt Enable Bits
EQ_OVER_P_INT_EN	[9]				EQ Overflow Error, primary Interrupt Enable Bit
EQ_NOT_EN_P_INT_EN	[8]	rst_l	0x0	RW	EQ Not Enabled, Primary Interrupt Enable Bit
MSI_MAL_ERR_P_INT_EN	[7]	rst_l	0x0	RW	Malformed MSI, Primary Interrupt Enable Bit
MSI_PAR_ERR_P_INT_EN	[6]	rst_l	0x0	RW	MSI Data Parity Error, Primary Interrupt Enable Bit
PMEACK_MES_NOT_EN_P_INT_EN	[5]	rst_l	0x0	RW	PME to ACK Message Not Enabled, Primary Interrupt Enable Bit
PMPME_MES_NOT_EN_P_INT_EN	[4]	rst_l	0x0	RW	PME Message Not Enabled, Primary Interrupt Enable Bit
FATAL_MES_NOT_EN_P_INT_EN	[3]	rst_l	0x0	RW	Fatal Message Not Enabled, Primary Interrupt Enable Bit
NONFATAL_MES_NOT_EN_P_INT_EN	[2]	rst_l	0x0	RW	Fatal Message Not Enabled, Primary Interrupt Enable Bit
COR_MES_NOT_EN_P_INT_EN	[1]	rst_l	0x0	RW	Correctable Message Not Enabled, Primary Interrupt Enable Bit
MSI_NOT_EN_P_INT_EN	[0]	rst_l	0x0	RW	MSI Not Enabled, Primary Interrupt Enable Bit

TABLE 8-29 IMU Interrupt Status Register – (0x00631010)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:47]				Reserved field
SPARE_S	[46:42]	rst_l	0x0	RW	Spare Error, Secondary Error Status Bit 1 = Error Received
EQ_OVER_S	[41]	rst_l	0x0	RW	EQ Overflow Secondary Error Status Bit 1 = Error Received

TABLE 8-29 IMU Interrupt Status Register – (0x00631010) *(Continued)*

EQ_NOT_EN_S	[40]	rst_l	0x0	RW	EQ Not Enabled Secondary Error Status Bit 1 = Error Received
MSI_MAL_ERR_S	[39]	rst_l	0x0	RW	Malformed MSI Secondary Error Status Bit 1 = Error Received
MSI_PAR_ERR_S	[38]	rst_l	0x0	RW	MSI Data Parity Secondary Error Status Bit 1 = Error Received
PMEACK_MES_NOT_EN_S	[37]	rst_l	0x0	RW	PME to ACK Message Not Enabled Secondary Error Status Bit 1 = Error Received
PMPME_MES_NOT_EN_S	[36]	rst_l	0x0	RW	PM PME Message Not Enabled Secondary Error Status Bit 1 = Error Received
FATAL_MES_NOT_EN_S	[35]	rst_l	0x0	RW	Fatal Message Not Enabled Secondary Error Status Bit 1 = Error Received
NONFATAL_MES_NOT_EN_S	[34]	rst_l	0x0	RW	Non Fatal Message Not Enabled Secondary Error Status Bit 1 = Error Received
COR_MES_NOT_EN_S	[33]	rst_l	0x0	RW	Correctable Message Not Enabled Secondary Error Status Bit 1 = Error Received
MSI_NOT_EN_S	[32]	rst_l	0x0	RW	MSI Not Enabled Secondary Error Status Bit 1 = Error Received
RESERVED	[31:15]				Reserved field
SPARE_P	[14:10]	rst_l	0x0	RW	Spare Error, Primary Error Status Bit 1 = Error Received
EQ_OVER_P	[9]	rst_l	0x0	RW	EQ Overflow Primary Error Status Bit 1 = Error Received
EQ_NOT_EN_P	[8]	rst_l	0x0	RW	EQ Not Enabled Primary Error Status Bit 1 = Error Received
MSI_MAL_ERR_P	[7]	rst_l	0x0	RW	Malformed MSI Primary Error Status Bit 1 = Error Received
MSI_PAR_ERR_P	[6]	rst_l	0x0	RW	MSI Data Parity Primary Error Status Bit 1 = Error Received
PMEACK_MES_NOT_EN_P	[5]	rst_l	0x0	RW	PME to ACK Message Not Enabled Primary Error Status Bit 1 = Error Received
PMPME_MES_NOT_EN_P	[4]	rst_l	0x0	RW	PM PME Message Not Enabled Primary Error Status Bit 1 = Error Received

TABLE 8-29 IMU Interrupt Status Register – (0x00631010) *(Continued)*

FATAL_MES_NOT_EN_P	[3]	rst_l	0x0	RW	Fatal Message Not Enabled Primary Error Status Bit 1 = Error Received
NONFATAL_MES_NOT_EN_P	[2]	rst_l	0x0	RW	Non Fatal Message Not Enabled Primary Error Status Bit 1 = Error Received
COR_MES_NOT_EN_P	[1]	rst_l	0x0	RW	Correctable Message Not Enabled Primary Error Status Bit 1 = Error Received
MSI_NOT_EN_P	[0]	rst_l	0x0	RW	MSI Not Enabled Primary Error Status Bit 1 = Error Received

TABLE 8-30 IMU Error Status Clear Register (0x00631018)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:47]				Reserved field
SPARE_S	[46:42]	por_l	0x0	RW1C	Spare Error, Secondary Error Status Bit 1 = Error Received
EQ_OVER_S	[41]	por_l	0x0	RW1C	EQ Overflow Secondary Error Status Bit 1 = Error Received
EQ_NOT_EN_S	[40]	por_l	0x0	RW1C	EQ Not Enabled Secondary Error Status Bit 1 = Error Received
MSI_MAL_ERR_S	[39]	por_l	0x0	RW1C	Malformed MSI Secondary Error Status Bit 1 = Error Received
MSI_PAR_ERR_S	[38]	por_l	0x0	RW1C	MSI Data Parity Secondary Error Status Bit 1 = Error Received
PMEACK_MES_NOT_EN_S	[37]	por_l	0x0	RW1C	PME to ACK Message Not Enabled Secondary Error Status Bit 1 = Error Received
PMPME_MES_NOT_EN_S	[36]	por_l	0x0	RW1C	PM PME Message Not Enabled Secondary Error Status Bit 1 = Error Received
FATAL_MES_NOT_EN_S	[35]	por_l	0x0	RW1C	Fatal Message Not Enabled Secondary Error Status Bit 1 = Error Received
NONFATAL_MES_NOT_EN_S	[34]	por_l	0x0	RW1C	Non Fatal Message Not Enabled Secondary Error Status Bit 1 = Error Received

TABLE 8-30 IMU Error Status Clear Register (0x00631018) *(Continued)*

COR_MES_NOT_EN_S	[33]	por_l	0x0	RW1C	Correctable Message Not Enabled Secondary Error Status Bit 1 = Error Received
MSI_NOT_EN_S	[32]	por_l	0x0	RW1C	MSI Not Enabled Secondary Error Status Bit 1 = Error Received
RESERVED	[31:15]				Reserved field
SPARE_P	[14:10]	por_l	0x0	RW1C	Spare Error, Primary Error Status Bit 1 = Error Received
EQ_OVER_P	[9]	por_l	0x0	RW1C	EQ Overflow Primary Error Status Bit 1 = Error Received
EQ_NOT_EN_P	[8]	por_l	0x0	RW1C	EQ Not Enabled Primary Error Status Bit 1 = Error Received
MSI_MAL_ERR_P	[7]	por_l	0x0	RW1C	Malformed MSI Primary Error Status Bit 1 = Error Received
MSI_PAR_ERR_P	[6]	por_l	0x0	RW1C	MSI Data Parity Primary Error Status Bit 1 = Error Received
PMEACK_MES_NOT_EN_P	[5]	por_l	0x0	RW1C	PME to ACK Message Not Enabled Primary Error Status Bit 1 = Error Received
PMPME_MES_NOT_EN_P	[4]	por_l	0x0	RW1C	PM PME Message Not Enabled Primary Error Status Bit 1 = Error Received
FATAL_MES_NOT_EN_P	[3]	por_l	0x0	RW1C	Fatal Message Not Enabled Primary Error Status Bit 1 = Error Received
NONFATAL_MES_NOT_EN_P	[2]	por_l	0x0	RW1C	Non Fatal Message Not Enabled Primary Error Status Bit 1 = Error Received
COR_MES_NOT_EN_P	[1]	por_l	0x0	RW1C	Correctable Message Not Enabled Primary Error Status Bit 1 = Error Received
MSI_NOT_EN_P	[0]	por_l	0x0	RW1C	MSI Not Enabled Primary Error Status Bit 1 = Error Received

TABLE 8-31 IMU Error Status Set Register (0x00631020)

Field	Bits	Reset Name	Reset Value	Type	Description
RESERVED	[63:47]				Reserved field
SPARE_S	[46:42]	por_l	0x0	RW1S	Spare Error, Secondary Error Status Bit 1 = Error Received
EQ_OVER_S	[41]	por_l	0x0	RW1S	EQ Overflow Secondary Error Status Bit 1 = Error Received
EQ_NOT_EN_S	[40]	por_l	0x0	RW1S	EQ Not Enabled Secondary Error Status Bit 1 = Error Received
MSI_MAL_ERR_S	[39]	por_l	0x0	RW1S	Malformed MSI Secondary Error Status Bit 1 = Error Received
MSI_PAR_ERR_S	[38]	por_l	0x0	RW1S	MSI Data Parity Secondary Error Status Bit 1 = Error Received
PMEACK_MES_NOT_EN_S	[37]	por_l	0x0	RW1S	PME to ACK Message Not Enabled Secondary Error Status Bit 1 = Error Received
PMPME_MES_NOT_EN_S	[36]	por_l	0x0	RW1S	PM PME Message Not Enabled Secondary Error Status Bit 1 = Error Received
FATAL_MES_NOT_EN_S	[35]	por_l	0x0	RW1S	Fatal Message Not Enabled Secondary Error Status Bit 1 = Error Received
NONFATAL_MES_NOT_EN_S	[34]	por_l	0x0	RW1S	Non Fatal Message Not Enabled Secondary Error Status Bit 1 = Error Received
COR_MES_NOT_EN_S	[33]	por_l	0x0	RW1S	Correctable Message Not Enabled Secondary Error Status Bit 1 = Error Received
MSI_NOT_EN_S	[32]	por_l	0x0	RW1S	MSI Not Enabled Secondary Error Status Bit 1 = Error Received
RESERVED	[31:15]				Reserved field
SPARE_P	[14:10]	por_l	0x0	RW1S	Spare Error, Primary Error Status Bit 1 = Error Received
EQ_OVER_P	[9]	por_l	0x0	RW1S	EQ Overflow Primary Error Status Bit 1 = Error Received
EQ_NOT_EN_P	[8]	por_l	0x0	RW1S	EQ Not Enabled Primary Error Status Bit 1 = Error Received

TABLE 8-31 IMU Error Status Set Register (0x00631020) *(Continued)*

MSI_MAL_ERR_P	[7]	por_l	0x0	RW1S	Malformed MSI Primary Error Status Bit 1 = Error Received
MSI_PAR_ERR_P	[6]	por_l	0x0	RW1S	MSI Data Parity Primary Error Status Bit 1 = Error Received
PMEACK_MES_NOT_EN_P	[5]	por_l	0x0	RW1S	PME to ACK Message Not Enabled Primary Error Status Bit 1 = Error Received
PMPME_MES_NOT_EN_P	[4]	por_l	0x0	RW1S	PM PME Message Not Enabled Primary Error Status Bit 1 = Error Received
FATAL_MES_NOT_EN_P	[3]	por_l	0x0	RW1S	Fatal Message Not Enabled Primary Error Status Bit 1 = Error Received
NONFATAL_MES_NOT_EN_P	[2]	por_l	0x0	RW1S	Non Fatal Message Not Enabled Primary Error Status Bit 1 = Error Received
COR_MES_NOT_EN_P	[1]	por_l	0x0	RW1S	Correctable Message Not Enabled Primary Error Status Bit 1 = Error Received
MSI_NOT_EN_P	[0]	por_l	0x0	RW1S	MSI Not Enabled Primary Error Status Bit 1 = Error Received

TABLE 8-32 IMU RDS Error Log Register (0x00631028)

Field	Bits	Reset Name	Reset Value	Type	Description TYPE
TYPE	63:58				The lowest 6 bits of the Type of the errored transaction as seen by the IMU in the RDS pipe stage 1111000 - 64 bit addressed MSI 1011000 - 32 bit addressed MSI 0110xxx - Message where xxx complies with the routing code in PCIE spec
LENGTH	57:48				The Length of the errored transaction
REQ_ID	47:32				The REQ ID of the errored transaction

TABLE 8-32 IMU RDS Error Log Register (0x00631028) *(Continued)*

TLP_TAG	31:24				The TLP tag of the errored transaction
BE_MESS_CODE	23:16				The message code of the Error is associated with a Message The First and Last Byte Enabled if the Error is associated with a MSI
MSI_DATA	15:0				The MSI data if the Error is associated with a MSI

Note – The field could be arranged for supporting MSI-X.

TABLE 8-33 IMU SCS Error Log Register (0x00631030)

Field	Bits	Reset Name	Reset Value	Type	Description TYPE
TYPE	63:58				The lowest 6 bits of the Type of the errored transaction as seen by the IMU in the RDS pipe stage 1111000 - 64 bit addressed MSI 1011000 - 32 bit addressed MSI 0110xxx - Message where xxx complies with the routing code in PCIE spec
LENGTH	57:48				The Length of the errored transaction
REQ_ID	47:32				The REQ ID of the errored transaction
TLP_TAG	31:24				The TLP tag of the errored transaction
BE_MESS_CODE	23:16				The message code of the Error is associated with a Message The First and Last Byte Enabled if the Error is associated with a MSI
RESERVED	00:00:00				Reserved field
EQ_NUM	5:0	por_l	6 bx	RW	Eq Number that the Transaction tried to go to but was not enabled

TABLE 8-34 IMU EQS Error Log Register (0x00631038)

Field	Bits	Reset Name	Reset Value	Type	Description TYPE
RESERVED	63:6				Reserved field
EQ_NUM	5:0	por_l	6 bx	RW	Eq Number that the Transaction tried to go to but was not enabled

8.5 Record Management Unit

8.5.1 RMU Function Description

The RMU is responsible for the orderly movement of the transaction records into and out of this unit on both ingress and egress pipelines. It talks to the IMU to deal with all the interrupts and accesses the TSB for transaction flow control and information tracking.

8.5.1.1 Link Receive Manger (LRM)

The LRM identifies MSIs and messages from other record types and accepts the Mondo requests from IMU. It arbitrates the Mondo interrupt requests from IMU and the interrupt requests from DIM and uses the local tag mechanism to manage the command pipeline order. Then, LRM sends interrupt records back to IMU and merges back the processed records from IMU with the other records in order and sends them up to the SRM.

8.5.1.2 Schedule Records Manager (SRM)

The SRM calculates the byte count, accesses the TSB to build entries and posts information on the TSB for non-posted DMA requests. It identifies and terminates the PIO write completion, then generates PIO transaction credits accordingly and enqueues them to the RRM. SRM builds SRM records and sends to MMU.

8.5.1.3 Retire Record Manager (RRM)

The RRM accesses TSB to read or read/clear entries to retrieve some information, such as tlp_tag, TC, attr., byte count, and lower_addr, to form the RRM records for compilations from the Retire Records from TCM. It identifies the Mondo replies in Retire Records and takes them off from the pipeline and forwards them to IMU. It forms the respective RRM records and enqueues them to DEM. The RRM sorts two sourced release records, one pushed from SRM and the other from ILU. Then, it passes the PIO credits directly to CLU.

If it's not the last packet of DMA completion, the remaining byte count after this packet needs to be recalculated and written back to TSB, which is “byte count” from TSB subtracting “byte count” from retire records. Moreover, the new value of lower_addr needs to be updated in TSB as well by adding bcnt[11:0] from retire record to lower_addr from TSB.

8.6 Transaction Scoreboard Unit (TSB)

8.6.1 TSB Function Description

The TSB is responsible for tagging and tracking all DMA Rd requests and unsupported transactions through the DMC in both ingress and egress pipelines. The storage area has 32 entries of 48 bits wide, and each entry is assigned with a tag to uniquely identify every transaction posted onto the scoreboard. The TSB manages the issuing and retiring of all tags with a free list.

8.7 Memory Management Unit (MMU)

8.7.1 IOMMU Description

The MMU translates virtual addresses to physical addresses. The MMU has a cache which stores a subset of translations in a Translation Storage Buffer (TSB) in main memory. One TSB entry is called a Translation Table Entry (TTE) which is 8 bytes. Addresses are pipelined through the MMU. If a translation misses in the cache, the pipeline is stalled until the data is fetched after the tablewalk.

When a SRM record is enqueued, the virtual address (VA) is extracted from the record to be translated and the remaining part of the record is held in the remaining data queue until being merged with the physical address to from schedule records. Then, the VA is sent to the Virtual Tag Block (VTB) for comparison.

The VTB contains 64 entries of virtual tags. The Translation Data Buffer (TDB) also has 64 entries, and each entry contains 8 TTEs. If there is a hit in the VTB, one of the eight TTEs from 64 entries in TDB will be selected and generates physical address (PA).

8.7.1.1 Required – IOMMU Bounds Check for Bypass Mode

Fire DMC follows the JBUS spec and uses {PA[42:41]==00, PA[40:36]==agent_id} for cacheable space. I n the IOMMU bypass mode, the logic does a bounds check with pa[63:42]!= 0x3FFF_00. In OpenSPARC T2, only supports a 40 bit cacheable address, with pa[39]==0 indicating cacheable. The SII and NCU have only pa[39:0], thus the upper pa bits will be thrown away at the interface to the SII and NCU blocks. SW must observe these address spaces when programming the IOMMU or IO devices. The IOMMU bypass logic will be modified to detect if pa[63:39]!= 0x1FFF_800 to conform to the new address ranges.

8.7.1.2 Required – Customized Virtual Tag Buffer Design

Fire uses the random logic to implement Virtual Tag Buffer CAM. The synthesis CAM logic in Fire ASIC costs a huge area. To reduce the area impact, OpenSPARC T2 will custom design the Virtual Tag Buffer. Please Refer to dmu_vtb_cam_spec for detail CAM implementation. The followings describe what have been changed in functionality:

In Fire design, the lookup reference address is compared to stored data in each of the 64 entries, and generates a decoded 64-bit vector, hit[63:0]. The hit output is registered in the next cycle. In OpenSPARC T2 Design CAM design, the lookup reference address is registered at the input of the CAM, The registered key is compared to stored data in each of the 64 entries, and generates a decoded 64-bit vector, hit[63:0]. The hit output is unregistered.

In Fire design, the lookup reference address is 16-bit wide (pcie[31:16]). In OpenSPARC T2 CAM design, the lookup address is increased to 29-bit (5-bit table-id plus pcie[39:16]).

8.7.1.3 Required – Customized Physical Tag Buffer Design

Fire uses the random logic to implement Physical Tag Buffer CAM. The synthesis CAM logic in Fire ASIC costs a huge area. To reduce the area impact, OpenSPARC T2 will custom design the Physical Tag Buffer. Please Refer to dmu_ptb_cam_spec for detail CAM implementation. The followings describe what have been changed in functionality:

In Fire design, the lookup reference address is compared to stored data in each of the 64 entries, and generates a decoded 64-bit vector, hit[63:0]. The hit output is registered in the next cycle. In OpenSPARC T2 Design CAM design, the lookup reference address is registered at the input of the CAM, The registered key is compared to stored data in each of the 64 entries, and generates a decoded 64-bit vector, hit[63:0]. The hit output is unregistered.

8.7.1.4 Required - Add a SUN4V Mode to support the hypervisor features:

Please refers to IOMMU MAS for detail:

8.8 Context Manager Unit (CMU)

8.8.1 CMU Function Description

The CMU is responsible for managing DMU pipelines and serves as the ordering point for transactions in both ingress and egress pipelines. The CMU keeps the order of DMA completions in the egress pipeline and the order of DMA requests and PIO completions in the ingress pipeline. The CMU contains three sub blocks.

8.8.1.1 Receive Context Manager (RCM)

The RCM dequeues Schedule Record from its input schedule record queue. It translates them into an ordered sequence of Packet Records which carry a payload segment of the requested data length in the Schedule Record with a maximum size of MaxPayload. It builds and manages a context entry for each DMA Read Schedule Record and assigns a unique Context Number to the Packet Record. The RCM builds Packet Records and enqueues them in strong order to the output Packet Record queue in the ingress pipeline destined for the Packet Record Manager (PRM).

From the Schedule Record, the RCM determines the number of Packet Record to be built, the packet sequence number for each Packet Record, the length of each Packet Record, the physical address of each Packet Record. If the Schedule Record is a DMA Mem Rd, one unique Context Record is requested from the context block. A packet sequence list array allocation is requested and a packet sequence list is constructed containing packet sequence entries for each Packet Record. The packet sequence list is bound to the unique context number.

8.8.1.2 Receive Context Entries

8.8.1.3 Transmit Context Manager (TCM)

The TCM dequeues Packet Records from its input record queue, processes the record according to its referenced context and generates Retire Records to enqueue to its output record queue to RRM. The TCM builds and maintains a context ordering for a series of Packet Records until completion by updating the Context Entry associated with the context and current packet sequence being processed.

The Packet Record is parsed to obtain the context number, packet sequence number. The context number is used to look up the context entry and the pointer to the packet sequence list. The packet sequence number is used to locate the associated entry in the packet sequence list. All type of transactions except DMA Rd Completion bypass the context lookup. If the packet satisfies the ordering bit in the context entry, the Retire Record is built and issued. If the Packet Record is returned out of order, the current Packet Record is stored to the context list pointed by its packet sequence list entry until prior Packet Records are returned. When all packet sequence for a context have been sent, the packet sequence numbers and the context number are retired. There is a strong ordering between packet sequence of the same context, but no ordering between different context.

8.8.1.4 Transmit Context Entries

8.8.1.5 Context Block (CTX)

The CTX contains the context record, the packet sequence list entry, and the context list entry. They are responsible for sequencing data in incremental address order for a DMA Rd Completion from SIU, which can be returned out of order. The context record contains the context number and the pointer to the beginning of the packet sequence entry list and the ordering bits to guarantee the packet order. Each packet sequence list belongs to a specific context and each packet sequence entry records

the completion status and the pointer to the context list entry if necessary. The context list is a temporary storage which contains the Packet Records returned out of order.

8.9 Packet Manager Unit (PMU)

8.9.1 PMU Function Description

The PMU interfaces with the CMU and CLU in the ingress pipeline. It segments packets issued by CMU into a series of cacheline oriented requests to CLU. It also interfaces with the PSB to manage and track packet transactions in the pipeline. The PMU contains only one sub-block, the Packet Receive Manager (PRM).

8.9.1.1 Packet Receive Manager (PRM)

The PRM dequeues the Packet Records in the ingress pipeline. From the address, byte enables, and length of the Packet Records, it determines the number of Cacheline Command Records to build and the physical address of each Cacheline Command Record. The PRM requests a packet tag from the PSB to put in each Cacheline Command Record of the same packet group along with the length and cacheline status. It the Packet Record carries a DMA Wr, PIO Rd completion, Mondo Interrupt Wr request, or MSI Wr request, no packet tag is required and no packet scoreboard entry is written. In case of a PIO completion, the PRM looks up the PSB to retrieve thread id and includes them in the Cacheline Command Record sent to CLU. For DMA Rd requests, the sbd_tag is replaced with pk_tag from PSB in the Cacheline Command Record.

8.10 Packet Scoreboard (PSB)

The PSB encapsulates the functions necessary to tag and track the internal packet transactions in the DMU pipelines. It is composed of two scoreboards, one tracks PIO's and another for DMA's.

8.10.0.1 Required, add jtag to thread id

The PIO scoreboard has an entry for each PIO. In the existing code, bit jbc_tag[9:0] held the transaction number, agent_id and jbus_id. The agent_id and jbus_id are concatenated to form a thread_id for N2. However this is only 64 id's, one more bit is to be added to account for jtag access by the NCU.

8.11 Cache Line Unit (CLU)

8.11.1 CLU Function Description

The CLU manages the DMC-DSN interface. It consists of two sub-blocks, Cacheline Transmit Manager (CTM) and Cacheline Receive Manager (CRM).

8.11.1.1 Cacheline Transmit Manager (CTM)

The CTM transmits requests to DSN for DMA MWr, DMA MRd, and Interrupts. It also returns PIO Rd completions to DSN. It moves data associated with these transactions from the DIU to DSN. CTM fully manages DOU DMA Rd buffer space and works with the DIM sub-block to manage the DIU buffer space for DMA Wrs, PIO Cpls, and INTs. It also issues tablewalk requests received from the MMU to DSN. Lastly, it forks unsupported requests and PCIE requests with error s to the Cacheline Receive Manager (CRM) for completion return to PCIE.

CTM release buffer space to the DIM for all transactions with data except for Mondo, which is managed by IMU. CTM exports to DIM the last DIU read pointer for DMA Wr and INT transaction data pulled from the DIU. DIM exports to CTM the last write pointer for DMA Wr and PIO Cpl transaction data writes to the DIU. CTM uses the read/write pointers to determine if the DIU is empty for a current data pull operation. CTM stalls the pipeline until the appropriate section of the DIU is not empty for either a DMA Wr or PIO Cpl data pull operation.

8.11.1.2 Cacheline Receive Manager (CRM)

The CRM receives DMA Rd/INT responses, tablewalk data responses, and PIO Rd/Wr requests form DSN. It moves data associated with these transactions to either the DOU (for DMA Rd responses and PIO Wr requests) or to the MMU (for tablewalk data responses). It manages out-of-order cacheline responses for DMA

Rds using the PSB to track packet build status. It formulates Packet Records for DMA Rd/INT responses and PIO Rd/Wr requests. It generates error completion packet records for unsupported/faulted PCIE requests forked from CTM.

For DMA Rd response, CRM uses the d_ptr field of the dmu_tag returned to quickly route the data to the DOU DMA data buffer. Data is moved independently of CRM's command processing pipeline. For PIO Wrs, the PIO Wr data buffer space in the DOU is dedicated and maximally sized. PIO Wr data is quickly moved by CRM to the DOB PIO data buffer.

CRM accesses the PSB for each PCIE DMA Rd response record received from DSN. It performs a read/modify/write operation to the PSB for tracking the responses associated with a packet. For each record response, CRM uses the pk_tag to index into PSB to check the cl_total field. If cl_total is 1, CRM builds a packet record issues it to the TCM and clears the PSB entry. It it's not, CRM decrements the cl_total field and writes the updated value back to the PSB. When 1st_cl field is set in the response, CRM updates the d_ptr field to the value of the d_ptr from the dmu_tag of the cacheline response.

8.11.1.3 Mondo Interrupt -> One Data Beat

The CTM block currently only extracts 1 data beat from the DIU ram, and then constructs the last 3 data beats and inserts 0's. The CTM block state machine which generates these extra beats will change to only output the first data beat which contains the Mondo payload.

8.12 Data In Unit (DIU)

8.12.1 DIU Function Description

The DIU is the storage buffer for all data associated with the ingress transactions and is composed of one synchronous dual port RAM and a set of storage flops. These are the DMA Wr/PIO Rd RAM and flops for the INT data. There are two separate write interfaces and one unified read port interface for the two storage elements. The DIM utilizes a write interface to the DMA Wr/PIO Rd RAM and IMU utilizes the second write interface directly to the INT DATA. The CLU will interface these elements via the unified read port interface.

For the INT data, 16 separate transactions can be stored in the registers since there are 16 entries. The storage is divided into two regions, one for 12 EQ writes and the other for 4 Mondos. In the RAM, 128 rows of total 192 rows are dedicated for DMA Wr data and the remaining are for PIO Rd completions.

8.13 Data Out Unit (DOU)

8.13.1 DOU Function Description

The DOU is the storage buffer for all data associated with the egress transactions and is composed of two synchronous dual port RAMs They are the DMA Rd RAM and PIO Wr RAM. There are two write interfaces and one unified read post interface for the two RAMs. The EIL in PEU utilizes the read interface to the DMA Rd and PIO Wr RAMs and the CRM utilizes the write interfaces to the two RAMs.

The data RAM store data and parity. The EIL uses the two most significant bits of the address supplied to select which RAM to be read from. The DMA Rd RAM has 2176 bytes available and is organized into 128 rows. The PIO Wr RAM has 1088 bytes that are organized into 64 rows for up to 16 separate transactions.

8.13.2 SRAM

8.13.2.1 Adding Test Features

Modify the SRAMs by adding JTAG and BIST functionality. Also add the required JTAG and BIST pins and logic external to the SRAMs. See the individual SRAM specifications for details on the external BIST logic added for each SRAM.

The current SRAMs are TDB, DIU, and 2 DOU rams. The existing synthesized cam in the MMU will be implemented as a custom cam/ram block.

There will be 2 BIST controllers at the DMU top level, 1 for rams and another for cams. The control/data wires into DMU will be added into the rtl. Control of the BIST engines will be external to the DMU with these wires being added at the DMU top level. DFT will be responsible for the BIST engines and the external control and registers which will be outside of the DMU.

Modify the srams to clear the inputs flops on reset, and implement the hold functionality for scan test.

8.14 DSN Overview

This is the specification for the interface block between the PCI-ex controller sub-block DMU and the core blocks NCU and SIU. The DMU<->NCU interface is for PIO read and write commands, interrupt acknowledges, DMU MMU snoop invalidate vectors and CSR reads/writes.

The DMU <-> SIU interface is for DMA reads/writes, inbound interrupts (Mondo and MSI) from the PIC-EX bus and PIO read completions.

The existing DMC (renamed DMU for OpenSPARCT2) remains unchanged, the DMU will interface through the SIU to the Level 2 cache thus there are some interface modifications needed. These modifications will be implemented in a new sub-block placed between the DMU and SIU/NCU blocks, to be called the DSN block.

The interfaces will be converted in this new block. The existing interfaces typically had separate command and data buses. The new interface adds a header cycle at the beginning of each transfer, which multiplexes the command info onto the data bus in the first cycle, thus there will be 1 extra cycle for each transaction. The following sections describes the various interfaces.

The interfaces to the DMU expect a data push model with unique credit ids, and the DSN will exploit this when modifying the transaction behaviors.

It appears that the DMU, SIU and NCU are all big endian, with byte_sel[0] matching data_bits[127:120] for all interfaces.

Also regarding the address buses:

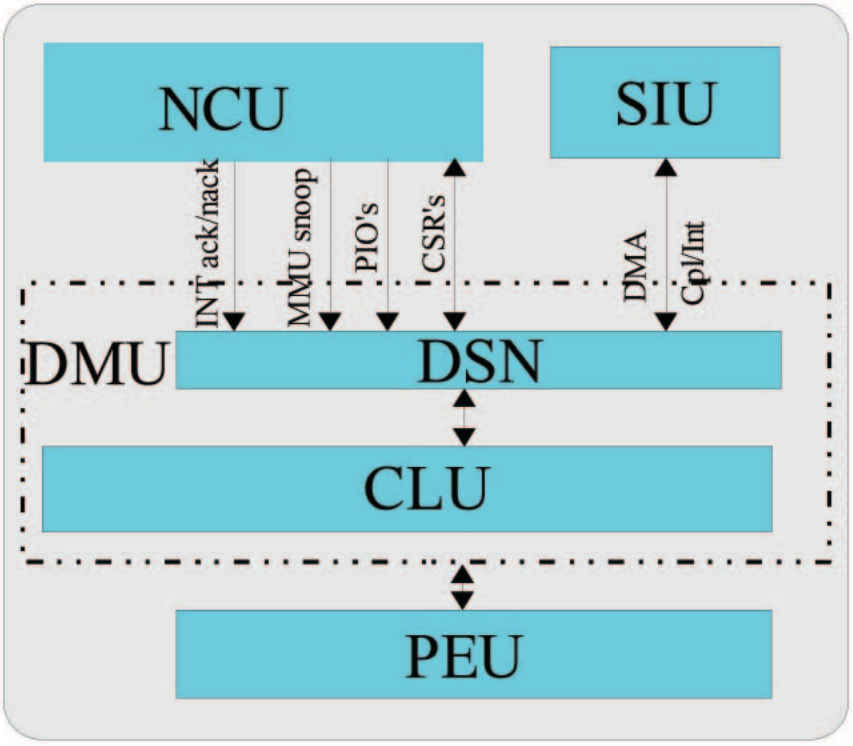
1. The DMU address bus is from [42:6] always cache line aligned, the DSN block will drop address bits[42:40].

2. The DSN PIO logic expects the NCU to send PA[35:0], always double work aligned, with the byte mask in the header.

The above address buses are then consistent with what the *OpenSPARC T2 Programmer's Reference Manual* allows if the SIU and DMU manage the upper bits. SW must manage DMA addresses so they fall into the cacheable range, and for PIO's the NCU must manage PA[39:36] such that they always map to the PA[35:0] expected by the DMU.

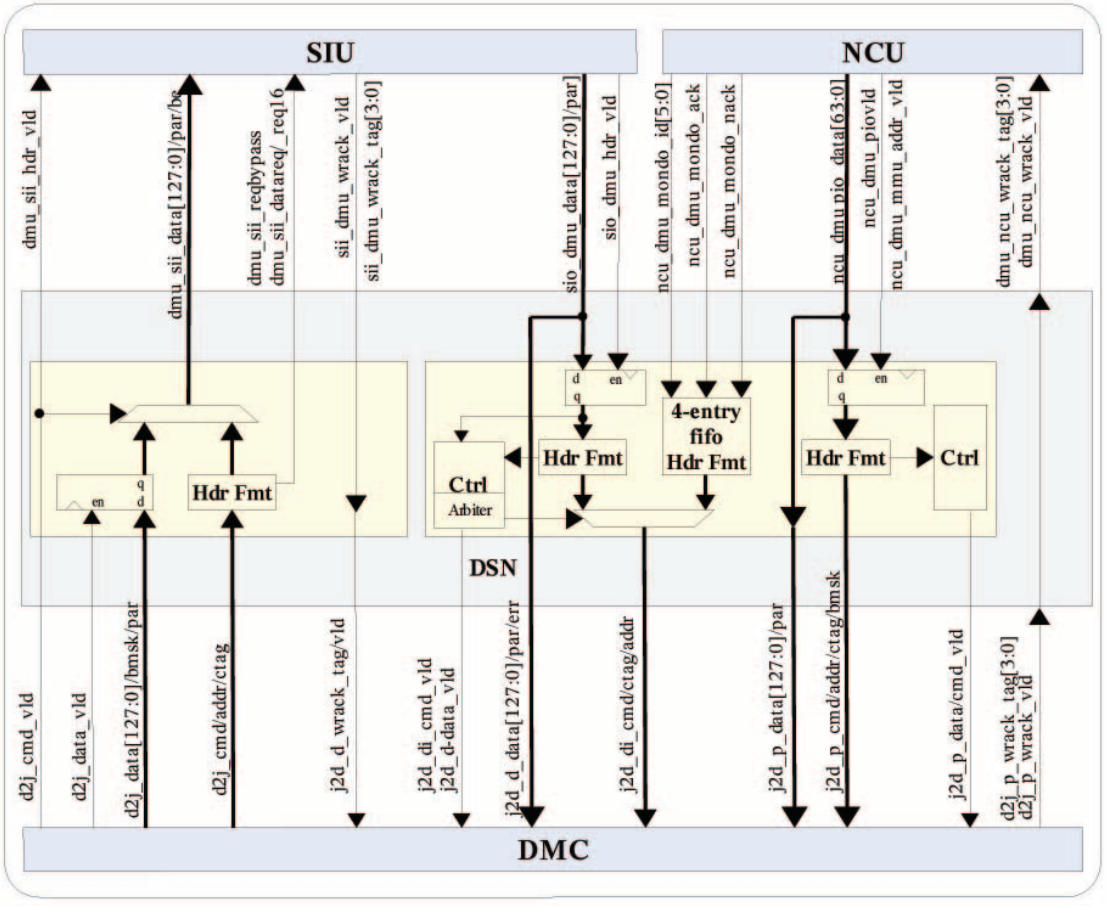
8.15 DSN Block Diagrams

FIGURE 8-4 Interface Block Diagram



8.16 DSN Detailed Block Diagram

FIGURE 8-5 Detailed Block Diagram



8.17 DSN Interface Descriptions

8.17.1 DSN-SIU Interface

The DSN-SIU interface will be used for all DMA's, sending Interrupts and PIO rd completions. It will have the following features:

1. It is expected that on all SIU responses it will return the dmc_tag[15:0] exactly as sent in the DMU->SIU command header.

2. DMA's and Interrupts will be credit id based with the DMU managing a total of 16 outstanding credits. The SIU will return the credit id for DMA writes (MSI's will be write packets) on the wrack bus, and the credit id for read completions will be returned with the data. When the credit id is returned the DMU will remove it from its credit vector and free it up for reuse. Mondo interrupts also use a credit and the SIU must forward the credit id of a Mondo interrupt to the NCU which will return this credit id along with the mondo id in the mondo ack bus, the DSN will form a completion packet and forward back to the CRM block which will update the DMU credit vector.

3. PIO read completions will first be routed through the SIU so that PIO reads can pull all preceding DMA writes into the L2 cache. The NCU will maintain a 16 entry credit scheme to limit the number of PIO's in the DMU/SIU to 16. The DMU/DSN will return the NCU credit id and thread id back to the SIU on PIO rd completions, and the SIU must pass this information on to the NCU. This information is needed by the NCU to remove the entry from its outstanding credit list and to know which thread to return the read PIO data.

4. The SIU must have sufficient buffering to hold 16 DMA writes and 16 PIO rd completions.

5. The interface from DSN to SIU will consist of control lines and a 128 bit data bus. The first 128 bits sent will be a header which contains the command information etc. subsequent cycles will contain the data.

6. The DSN block will take the information from the pins between the DSN and DMU blocks and use it to generate the header driving to the SIU, and when the SIU drives a header, it will take that information and create the pin data toward the DMU.

7. On 8 byte PIO rd cpl's the DSN block must detect which 64 bits the return data should be located and replicate these 64 bits onto the opposite 64 bits. This can be done by keeping a 2 bit scoreboard of pio_addr[3] indexed by the credit_id, written on NCU vld, and reading on pio_rd_cpl's. The duplication of the relevant

64 bits onto both halves of the 128 bit return data bus is a requirement of the core. In addition the scoreboard must track whether the returning PIO is a 16 byte read or 8 byte, if 16 byte then the data is not replicated, with a second bit in the scoreboard.

8.17.1.1 DSN-SIU Interface List

TABLE 8-35 DSN-SIU Interface List

Signal Name	I/O	Size	From/To	Description
DSN to SIU Signals				
dmu_sii_hdr_vld	O	1	DMU->SIU	Asserted during the header phase of any requests from DMU to SIU. Not asserted during the data transfer phase.
dmu_sii_reqbypass	O	1	DMU->SIU	Asserted for PIO rd cpl's
dmu_sii_datareq	O	1	DMU->SIU	Valid during the header phase only. 0: Current request is a read, with no payload; 1: Current request is a write, with 1 or 4 cycles of data payload
dmu_sii_datareq16	O	1	DMU->SIU	Valid during the header phase only. Don't care if dmu_sii_datareq is 0. 0: Current write request has 64B data payload; 1: Current write request has 16B data payload. (meant for NCU - int/PIO read data)
dmu_sii_data[127:0]	O	128	DMU->SIU	Packet header/data for L2/NCU. (Big-endian) For PIO read completions, there are 2 cases, 16 byte and <=8byte cpl's, in the case of 8byte pio cpl's the data will be replicated on both halves of the bus, DSN keeps a scoreboard to determine which 64 bits to replicate.
dmu_sii_be[15:0]	O	16	DMU->SIU	Packet data byte enables/errors. Only valid during data transfer phase. (dmu_sii_be[0] is for dmu_sii_data[7:0]).
dmu_sii_parity[7:0]	O	8	DMU->SIU	Parity of data payload cycles (127:0)
dmu_sii_be_parity	O	1	DMU->SIU	Parity for dmu_sii_be[15:0]
Note: detected parity errors on d2j_data[127:0] will be signaled by flipping dmu_sii_parity[0] to SII				
SIU to DSN Signals				
sii_dmu_wrack_tag[3:0]	1	4	SIU->DMU	j2d_d_wrack_tag[3:0] DSN/DMU name Transaction credit id for dma wrack
sii_dmu_wrack_par	1	1	SIU->DMU	Odd parity ^sii_dmu_wrack_tag[3:0]
sii_dmu_wrack_vld	1	1	SIU->DMU	j2d_d_wrack_vld DSN/DMU name Valid signal for j2d_d_wrack_tag

TABLE 8-35 DSN-SIU Interface List *(Continued)*

Signal Name	I/O	Size	From/To	Description
sio_dmu_hdr_vld	1	1	SIU->DMU	Envelops the header of any requests from SIU to DMU. Not asserted during the data transfer phase. DSN determines from the header if and how much data will follow.
sio_dmu_data[127:0]	1	128	SIU->DMU	Packet header/data for DMU
sio_dmu_parity[7:0]	1	8	SIU->DMU	Parity of payload cycles (128:0).

8.17.1.2 SIU to DSN Egress Commands

These are the commands as defined at the DSN/DMU boundary, and must be generated from the SIU to DSN header. Thus the DSN logic will take in the SIU header and generate the following commands back to the DMU. .

TABLE 8-36 SIU to DSN Egress Commands

Transaction type	cmd	ctag
Bit width **18**	**2** **[17:16]**	**16** **[15:0]**
DMA Rd Return	2'b00	dmc_tag[15:0]
DMA Rd Return Err	2'b01	dmc_tag[15:0]
Interrupt Nack	2'b10	N/A for N2
Interrupt Ack	2'b11	N/A for N2

TABLE 8-37 DMC_TAG Field Definitions

Field	Bits	Description
DMA transactions		
dmc_tag[15]	type	0b-indicates DMA/Int transactions
dmc_tag[14:11]	cl_tag[3:0]	Dmc transaction number for tracking credits
dmc_tag[10:6]	d_ptr[4:0]	Used for DMA Rds only-dou dma rd buffer address
dmc_tag[5:1]	pkt_tag[4:0]	Used for DMA Rds only-PSB index for building packet records
dmc_tag[0]	cl_sts	Used for DMA Rds only-indicates 1st cacheline in packet sequence
Int Transactions		
dmc_tag[15]	type	0b-indicates DMA/Int transactions

TABLE 8-37 DMC_TAG Field Definitions *(Continued)*

Field	Bits	Description
dmc_tag[14:11]	cl_tag[3:0]	Dmc transaction number for tracking credits
dmc_tag[10:3]	Rsv[7:0]	reserved
dmc_tag[2:1]	mdo_tag[1:0]	mondo_tag for mondo-reply to IMU
dmc_tag[0]	rsv	reserved
MMU Tablewalk Transactions		
dmc_tag[15]	type	1b-indicates MMU Tablewalk transactions
dmc_tag[14:11]	cl_tag[3:0]	Dmc transaction number for tracking credits
dmc_tag[10:6]	Rsv[4:0]	reserved
dmc_tag[5:0]	Mtag[5:0]	Used for MMU tablewalks only-MMU tag for tracking tablewalks
PIO Cpl Transactions		
dmc_tag[15:12]	Rsv[3:0]	Must be 4'b1000
dmc_tag[11:8]	jbc_trans_#[3:0]	Pio transaction credit id
dmc_tag[7]	Rsv	
dmc_tag[6:0]	thread_id[6:0]	Thread id of PIO read request. If thread_id[6]==0, then thread_id[5:0] is the thread id, if thread_id[6]==1 then it is a jtag txn.

Note – The NCU will distinguish interrupts from PIO cpl's by using dmc_tag[15]. And the NCU will use the thread_id from the mondo data to determine which thread to interrupt.

8.17.1.3 SIU to DSN Outbound Header sent by SIU (DMA rd cpl's only)

TABLE 8-38 SIU to DSN Header Bit Definitions

Header cycle siu_dsn_data	Name	Description
[127]	Command - DMA read response	1000_00
	[127] = response bit	1 = DMA read response, this can be a PCI-ex DMA read or a DMU MMU tablewalk response.
[126:122]	reserved	Ignore, may be 0 or 1
[121:84]	reserved	Must be 0
[83]	reserved	Must be 0
[82]	reserved	Must be 0
[81]	UE	1 = error detected on dmc_tag or address accumulated throughout the DSN->SII->l2$->SIO->DSN path. If this bit is a 1 the DSN will block the return of the current packet back to DMU. At this point SW must intervene and correct because now this packet will never retire. If this bit is set, the DMU will not return an error on dmu_ncu_ctag_ue
[80]	DE	1 = data payload has a detected uncorrectable error this could be: 1. timeout error 2. unmapped error 3. data ue error from dram
[79:64]	dmc_tag[15:0]	Returned dmc_tag extracted from the DSN to SIU command and returned without changes
[63:62]	reserved	Must be 0
[61:56]	Ctagecc[5:0]	Ecc on dmc_tag[15:0]
[55:37]	reserved	Must be 0
[36:0]	reserved	SIU does not return the DMA read address with the completion.

8.17.1.4 Bit Mapping from DSN to SII for DMA rd/wrt Requests

dmu_sii_data[127]= d2j_cmd[3]

dmu_sii_data[126]= ~d2j_cmd[3] & ~d2j_cmd[2]

dmu_sii_data[125]= !d2j_cmd[3] && d2j_cmd[1]

dmu_sii_data[124]= !d2j_cmd[3] && !d2j_cmd[1] && d2j_cmd[0]

dmu_sii_data[123]= !d2j_cmd[3] && !d2j_cmd[2]

dmu_sii_data[122]= d2j_cmd[3] || (!d2j_cmd[3] && d2j_cmd[2])

dmu_sii_data[121:120]= 2'b00

dmu_sii_data[119:83]= 0

dmu_sii_data[84]= ~^dmu_sii_data[39,37,35,33,31,29,27,25,23,21,19,17,15,13,11,
9,7,5,3,1]

dmu_sii_data[83]= ~^dmu_sii_data[38,36,34,32,30,28,26,24,22,20,18,16,14,12,10,
8,6,4,2,0]

dmu_sii_data[82]= d2j_cmd[3] && d2j_cmd[1] & !d2j_cmd[0]

dmu_sii_data[81]= d2j_cmd[3] && d2j_cmd[1] && d2j_cmd[0]

dmu_sii_data[80]= 1'b0

dmu_sii_data[79:64]= d2j_cmd[3] ? {1'b1,d2j_ctag[14:0]} : d2j_ctag[15:0]

dmu_sii_data[62]= ~^dmu_sii_data[127:122]

dmu_sii_data[61:56]= ecc on dmu_sii_data[79:64] (see below, or soc ras spec)

dmu_sii_data[39:6]= d2j_addr[33:0] -> pa[39:6]

dmu_sii_data[5:0]= 0

8.17.1.5 Bit Mapping from NCU/SIU Header to DMC for DMA/Int ack/nack

j2d_di_cmd[1:0]= if sio_dmu_hdr_vld = 1'b1 then 1 cycle later

j2d_di_cmd[1:0] = {1'b0, DE]

delayed by 1 clock)}

else if sio_dmu_hdr_vld 1'b0 && mondo ack/nack in

the return fifo in dsn then 1 cycle later

j2d_di_cmd[1:0] = {1'b1,mondo_dout[7]}

j2d_di_ctag[15:0]= if sio_dmu_hdr_vld = 1'b1 then 1 cycle later

j2d_di_ctag[15:0] = sio_dmu_data[79:64] delayed

by 1 cycle (ecc corrected).

Else if sio_dmu_hdr_vld = 1'b0 && mondo_fifo is valid

j2d_di_ctag[15:0] = {1'b0,mondo_dout[5:2],8'b0,

mondo_dout[1:0],1'b0}

j2d_di_data[127:0]= sio_dmu_data[127:0] no delay.

j2d_d_data_err= sio_dmu_data[80] delayed by 1 cycle, + any locally detected parity errors

---

Note – The mondo_fifo_dout is a delayed version of ncu_dmu_mondo_id[5:0] through a fifo in the DSN and these bits are defined as: [5:2] = dmc_tag[14:11], [1:0] = dmc_tag[2:1].

This fifo is needed because a dma rd return could occur at the same time as an int ack from the NCU, so the fifo buffers up the int ack until a quiescent cycle in the dma rd return. If a parity error is detected on the int ack from the NCU then this packet is not placed in the fifo, and will be dropped, SW must intervene and clean up.

Note – If dmu_sii_data[81](UE) is asserted if either the SII, l2$ or SIO detect a ctag ecc ue, or adr parity error, and the DSN will block the return of this packet, this is done so the DMC scoreboards do not get corrupted

---

Note – If dmu_sii_data[80](DE) will not be asserted for errors from the l2$. Instead the L2$ will flip a parity bit on the outbound data, and the DSN will detect this instead of using bit 80.

8.17.1.6 DMC to SIU Ingress Commands

These are the commands as defined at the DSN/DMU boundary. They must be decoded and used to generate the header for DSN to SIU ingress commands.

See the SIU specification Sec. 7.5 for the header bit specification.

The DSN logic will build the header to the SIU using the following commands from the DMU.

Note – 64 byte PIO's are not supported in OpenSPARCT2, and on PIO read completions the address will not be returned.

TABLE 8-39 DMC to SIU Ingress Command Bit Definitions

Transaction type	cmd	address	ctag
Bit width **57**	**4** **[56:53]**	**37** **[52:16]**	**16** **[15:0]**
DMA Full Wr	4'b0000	PA[42:6]	dmc_tag[15:0]
DMA Partial Wr	4'b0001	PA[42:6]	dmc_tag[15:0]
DMA Rd	4'b0010	PA[42:6]	dmc_tag[15:0]
DMA Rd Shared (tablewalk)	4'b0011	PA[42:6]	dmc_tag[15:0]
Interrupt (mondo)	4'b0100	PA[42:6]	dmc_tag[15:0]
PIO Rd Return 16	4'b1000	n. a.	Rsv[4:0] jbc_tag[10:0]
PIO Rd Return 64	4'b1001	PA[42:6]	Rsv[5:0] jbc_tag[10:0]
PIO Rd Return Tout Err	4'b1010	n. a.	Rsv[4:0] jbc_tag[10:0]
PIO Rd Return Bus Err	4'b1011	n. a.	Rsv[4:0] jbc_tag[10:0]

Note – On PIO rd return with errors, the data packet will still be sent but may be invalid.

Note – jbc_tag is 11 bits, whereas in the d2j_ctag jbc-tag[11:8] held the credit_id, the reason is because in the rtl the width of the jbc_tag value is a parameter and is also used to automatically size datapaths/fifo's etc. making in jbc_tag[11:0] would have added a bit throughout the entire crm/pmu/psb/scoreboard/ctm path.

8.17.1.7 DSN to SII Header as sent by DSN

The DSN block will take the DMU to DSN command information and concatenate this and form a header which will be sent before the data. Below is a table of the header values:

TABLE 8-40 DSN to SII Header Bit Definitions

Header cycle dsn_siu_data [msb:lsb]	Name	Description
[127:122]	Command	
	- PIO Read return	1010_01
	- DMA read Request	0010_10
	- Interrupt Mondo Write	0000_01
	- Dma Write full cacheline	0100_10
	- DMA write Merge 64 bytes	0101_10
	[127] = response bit	1 = response, 0= request Only set on PIO rd cpl's, this tells the SIU which queue to enter the DMA write data or PIO rd cpl data.
	[126]	Posted bit, 1=dma write
	[125]=read bit	1 = DMA read request 0 = DMA write request, interrupt mondo request, write response
	[124] = write bytemask active	Ignored by SIU if response bit is set or if read bit is set 1 = use byte enables 0 = all bytes active
	[123] = l2 bit	1 = to l2 set for DMA write request, DMA read request
	[122] = NCU bit	1 = to NCU set for Interrupt mondo request, PIO read response
[121:85]	reserved	Must be 0
[84:83]	address_par[1:0]	Address parity, ap[0] for even bits of pa, ap[1] for odd bits
[82]	timeouterror	1 = this packet had timed out, PIO completions only
[81]	UnmappedAddresseError	1 = this packet's address mapped to a nonexistent, reserved, or erroneous address PIO completions only
[80]	UncorrectableError	1 = data payload has a detected uncorrectable error This bit is always 0

TABLE 8-40 DSN to SII Header Bit Definitions *(Continued)*

Header cycle dsn_siu_data [msb:lsb]	Name	Description
[79:64]	dmc_tag[15:0]	for PIO read completions this is PIOID Bits [11:8] will be the credit id returned on PIO rd completions, bits [6:0] will be the thread ID. For DMA's this will be the dmc_tag value from the DMU interface
[63]	reserved	Must be 0
[62]	cmd_par	Odd Parity on bits {[127:122]}
[61:56]	Ctagecc[5:0]	SECDED ecc on bits[79:64]
[55:40]	Reserved	Must be 0
[39:6]	PA[39:6]	Valid for DMA requests only
[5:0]	PA[5:0]	0's Always cache line aligned

8.17.1.8 DSN-SII Header RAS

The header from DSN to SII will incorporate ecc and parity on all significant bits because the SII inserts the header into the same fifo ram which holds the data. RAS guidelines call for protection on all significant rams, therefore the DSN will generate these ras bits before sending to the SII.

The header is to be divided into 3 fields, ctag, address and command/status. Each group will have its own protection, ecc on the ctag and parity on the other 2.

The ctag ecc will use a SECDCD code with 6 extra check bits. The check bits are an xor of a series of bits generated as follows:

```
chk[0] =    ^{di[15],di[13],di[11],di[10],di[8],di[6],di[4],di[3],di[1],di[0]};
chk[1] =    ^{di[13],di[12],di[10],di[9],di[6],di[5],di[3],di[2],di[0]};
chk[2] =    ^{di[15],di[14],di[10],di[9],di[8],di[7],di[3],di[2],di[1]};
chk[3] =    ^{di[10],di[9],di[8],di[7],di[6],di[5],di[4]};
chk[4] =    ^{di[15],di[14],di[13],di[12],di[11]};
chk[5] =    ^{di[15],di[14],di[13],di[12],di[11],di[10],di[9],di[8],
              di[7],di[6],di[5],di[4],di[3],di[2],di[1],di[0],
              chk[0],chk[1],chk[2],chk[3],chk[4]};
```

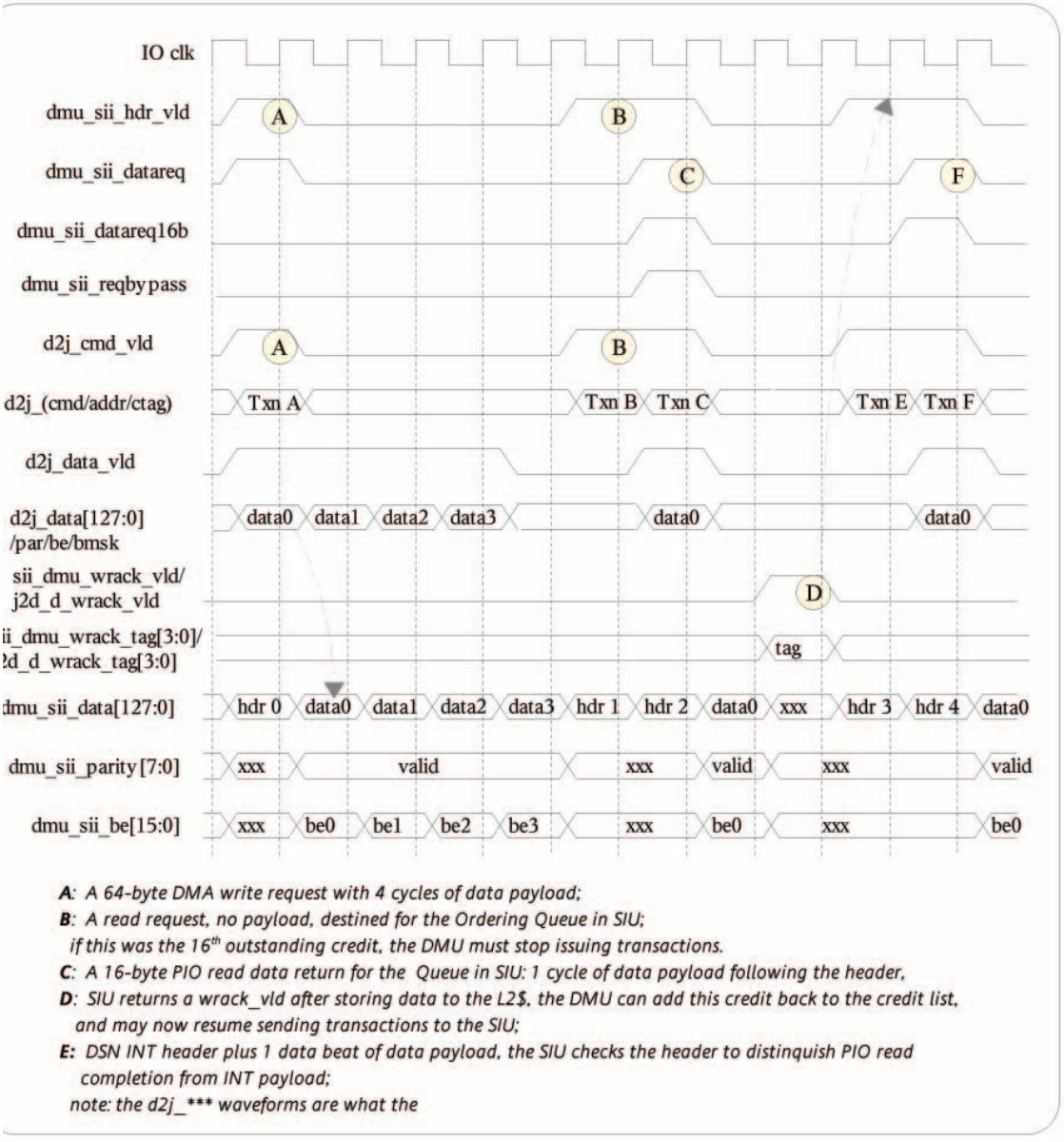
where di[15:0] = dmc_tag[15:0];

chk[5:0] will be placed on header bits [61:56];

Odd parity will be used for the address and control/status. Header bit [84:83] will hold odd parity for the address (header bits [42:6]), where [84] is for the odd address bits, [83] for even. And header bit [62] will hold odd parity for header bits {[127:122]}

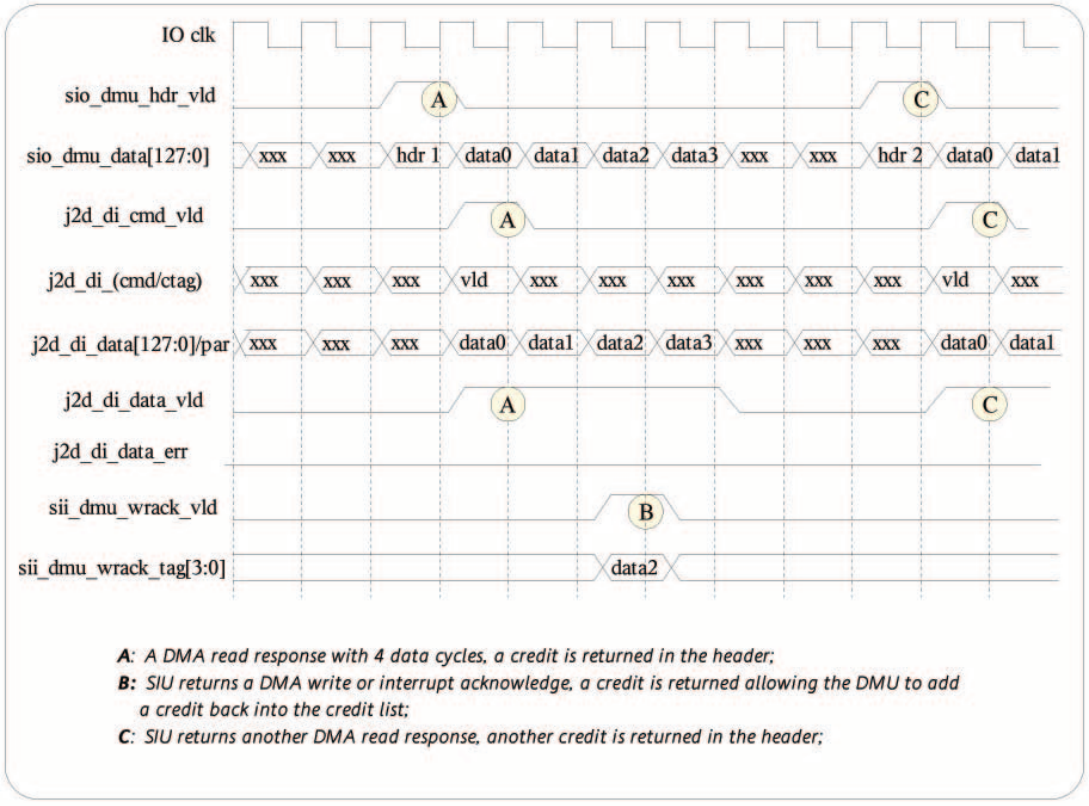
8.17.1.9 DSN-SII Interface Timing Diagrams

FIGURE 8-6 Ingress Interface Timing Diagram



A: A 64-byte DMA write request with 4 cycles of data payload;
B: A read request, no payload, destined for the Ordering Queue in SIU;
if this was the 16th outstanding credit, the DMU must stop issuing transactions.
C: A 16-byte PIO read data return for the Queue in SIU: 1 cycle of data payload following the header,
D: SIU returns a wrack_vld after storing data to the L2$, the DMU can add this credit back to the credit list,
and may now resume sending transactions to the SIU;
E: DSN INT header plus 1 data beat of data payload, the SIU checks the header to distinquish PIO read completion from INT payload;
*note: the d2j_*** waveforms are what the*

FIGURE 8-7 Egress Interface Timing Diagram



Only DMA rd cpl's and wrack's

8.17.2 DSN-NCU Interface

8.17.2.1 DSN-NCU Interface Description

The DSN-NCU interface will be used for all PIO read/write command requests, interrupt ack/nack, DMU MMU snoop invalidate vectors and CSR read/writes, but the PIO rd completions will return through the SIU block. It will have the following features:

1. The NCU will send a PIO read or write request along with a transaction credit id and the thread id for read return.

2. The NCU will only request 8 bytes or less for writes and up to 16 bytes for reads. The DSN will need to extract this information from the header and PA and construct the rest of the 16 bytemasks to the DMU. And on PIO writes, the DSN will need to replicate the 8 bytes sent by the NCU on the 16 byte SIU bus and set the bytemask correctly. The DMU does not allow 8 byte requests to cross 8 byte boundaries.

3. The DMU will store the thread id sent by the NCU in a ram structure indexed by the transaction credit id. On read completions the DMU will then return the thread id back to the SIU along with the data and the transaction credit id, the SIU must pass this along to the NCU when it returns the PIO read data to the NCU.

4. A PIO wrack from DSN to NCU will inform the NCU which write transaction credit id it may remove from its local PIO 16 entry scoreboard.

5. Interrupt egress traffic (ACK/NACK) will originate in the NCU and directly interface to the DSN. Since the data into the DMU is multiplexed onto 1 bus for dma read return data and mondo acks the DSN will have to account for simultaneous dma read return and mondo acks. The DMU will only have 4 outstanding mondo interrupts and returning dma read returns have been stretched to include an extra cycle at the beginning for a header multiplexed onto the data bus. The DSN will exploit this and queue up mondo acks if they collide with returning dma reads, and multiplex the mondo acks into this DMU dead cycle created by the new interface. A fifo of 4 entries should be sufficient since the DMU can only have 4 outstanding mondo interrupts, and cannot issue another until an ack is returned.

6. The NCU block will also be used to invalidate entries in the DMU MMU. The existing interface was used to snoop the jbus, but for OpenSPARCT2 the NCU will have a CSR writable register which when written to by SW, will trigger sending the value as a PA to be invalidated. The DMU MMU will take this value, match it against its current PA entries, and invalidate any line which matches. To save pins, the invalidate address will be multiplexed onto the NCU 64 bit data bus and a separate valid sign for invalidates will be used to distinguishing PIO commands from MMU invalidate commands.

7. DMU CSR read/writes will be interfaced through the DSN.

8.17.2.2 DSN-NCU Interface Pin List

TABLE 8-41 DSN to NCU Interface Pin List

Signal name	direction	Description
DMU PIO commands		
ncu_dmu_pio_hdr_vld	input	NCU to DMU pio_data header is valid
ncu_dmu_mmu_addr_vld	input	NCU to DMU pio_data mmu invalidate vector is valid
ncu_dmu pio_data[63:0]	input	NCU to DMU pio_data bus
DMU to NCU PIO write completions		
dmu_ncu_wrack_vld	output	Release credit id valid bit
dmu_ncu_wrack_tag[3:0]	output	4-bit release credit id
dmu_ncu_wrack_par	output	Odd parity on dmu_ncu_wrack_tag[3:0]
DMU Mondo ack's		
ncu_dmu_mondo_ack	input	Mondo Interrupt ack
ncu_dmu_mondo_nack	input	Mondo Interrupt nack
ncu_dmu_mondo_id[5:0]	input	[5:2] = cl_tag[3:0], [1:0] = mdo_tag[1:0]
ncu_dmu_mondo_id_par	input	Odd parity ^ncu_dmu_mondo_id[5:0]
This Signals are not needed, tie off between DSN/DMU blocks		
d2j_tsb_base[42:13]		n. a.
d2j_tsb_enable		n. a.
d2j_tsb_size[3:0]		n. a.
DSN to NCU error reporting Signals		
dmu_ncu_d_pe	output	Indicates parity error on DMA rd data
dmu_ncu_siicr_pe	output	Indicates parity error on dma write credit ack
dmu_ncu_ctag_ue	output	Indicates ue error on dma read return ctag
dmu_ncu_ctag_ce	output	Indicates ce error on dma read return ctag
dmu_ncu_ncucr_pe	output	Indicates parity error on mondo ack
dmu_ncu_ie	output	Indicates parity error on DMC internal
Note: the error reporting Signals to the ncu are single pulse per error.		
NCU to DSN error injections Signals		
ncu_dmu_d_pei	input	Force DMA read return pe
ncu_dmu_siicr_pei	input	Force DMA write credit return pe
ncu_dmu_ctag_uei	input	Force DMA read return header ctag ue

TABLE 8-41 DSN to NCU Interface Pin List *(Continued)*

Signal name	direction	Description
ncu_dmu_ctag_cei	input	Force DMA read return header ctag ce
ncu_dmu_ncucr_pei	input	Force NCU mondo ack pe
ncu_dmu_iei	input	Force pe on DMC/MMU rams (deviostb & tdb
Note: the error injection Signals are levels, thus will force errors on all transactions until undriven		

8.17.2.3 NCU-DSN Egress PIO Commands

Note – PIO blk operations are not supported. The NCU will implement a CSR register which when written to will force a snoop invalidate to the DMU MMU. The Signals will go through the DSN block simply to be renamed.

TABLE 8-42 NCU to DSN PIO Command Bit Definitions

Transaction type	cmd	address	bytemask	ctag
Bit width **66**	**4** **[65:62]**	**36** **[61:26]**	**16** **[25:10]**	**10** **[10:0]**
PIO Wr Blk Mem -64	4'b0000	A[35:0]	rsv	jbc_tag[9:0]
PIO Wr Blk Mem-32	4'b0001	A[35:0]	rsv	jbc_tag[9:0]
PIO Wr 16b Mem-64	4'b0100	A[35:0]	bmsk	jbc_tag[10:0]
PIO Wr 16b Mem-32	4'b0101		bmsk	jbc_tag[10:0]
PIO Wr 16b IO	4'b0110	A[35:0]	bmsk	jbc_tag[10:0]
PIO Wr 16b Config	4'b0111	A[35:0]	bmsk	jbc_tag[10:0]
PIO Rd Blk Mem-64	4'b1000	A[35:0]	rsv	jbc_tag[9:0]
PIO Rd Blk Mem-32	4'b1001	A[35:0]	rsv	jbc_tag[9:0]
PIO Rd 16b Mem-64	4'b1100	A[35:0]	bmsk	jbc_tag[10:0]
PIO Rd 16b Mem-32	4'b1101	A[35:0]	bmsk	jbc_tag[10:0]
PIO Rd 16b IO	4'b1110	A[35:0]	bmsk	jbc_tag[10:0]
PIO Rd 16b Config	4'b1111	A[35:0]	bmsk	jbc_tag[10:0]

TABLE 8-43 jbc_tag[10:0] Descriptions

Field	Bits	Description
PIO transaction tag		
jbc_tag[10:7]	jbc_trans_#[3:0]	Pio transaction number
jbc_tag[6:0]	thread_id[6:0]	Thread id used by NCU to return PIO read data to the requesting thread. thread_id[6] indicates a jtag operation. thread_id[5:0] is the cpu/thread id if thread_id[6]==0

Note – Only 7 bits are used for the thread id, the full 8 bits of thread id are not sent by the NCU to the DSN for PIO's, only cores can sent nc PIO's, bit thread_id[6]==1 implies jtag access.

8.17.2.4 Bit Mapping from NCU Header to DMC for PIO rd/wrts

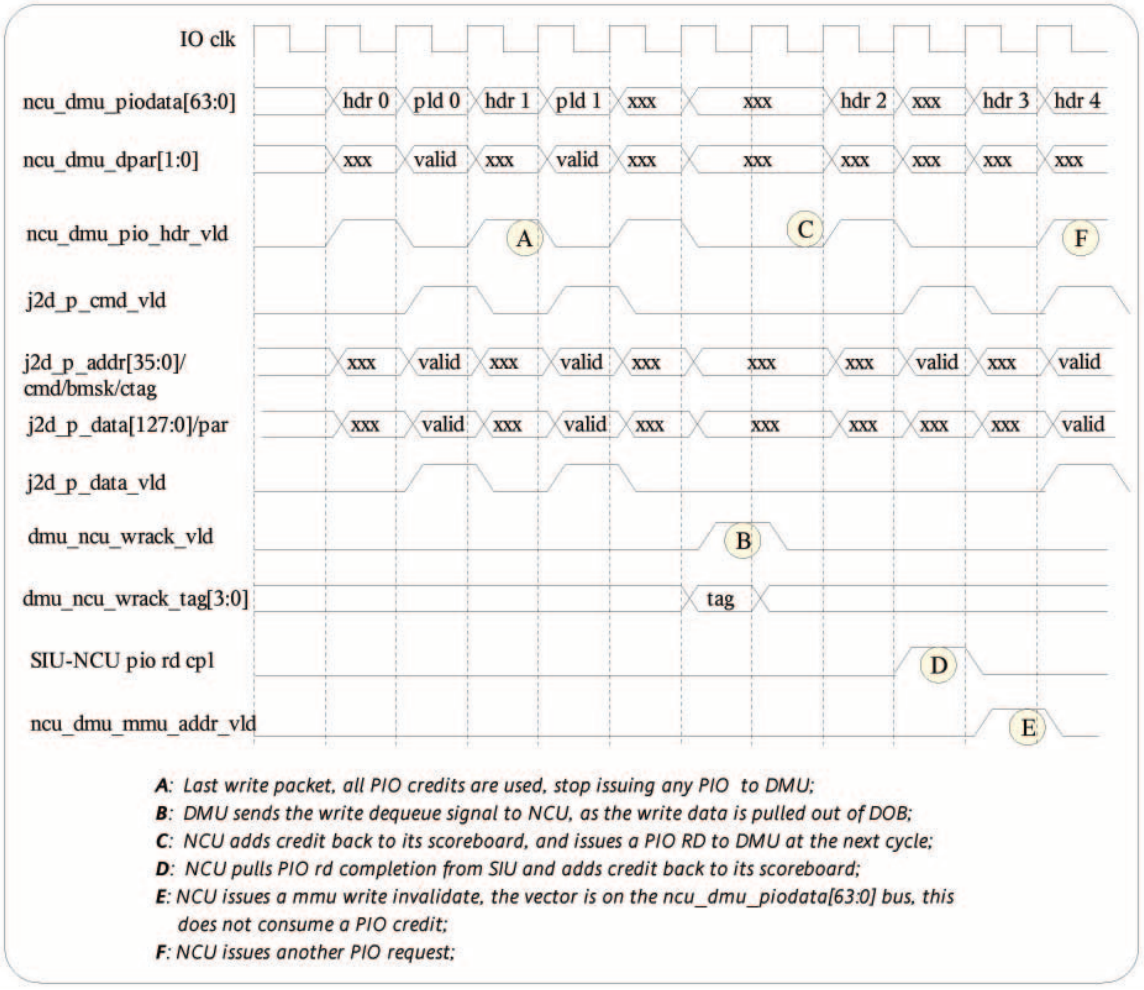
```
  j2d_p_addr[35:0]=  ncu_dmu_pio_data[35:0]
  j2d_p_cmd[3]= ncu_dmu_pio_data[60]
  j2d_p_cmd[2]= 1'b1
  j2d_p_cmd[1]=  !ncu_dmu_pio_data[37] && ncu_dmu_pio_data[36] ||
                 !ncu_dmu_pio_data[37] && !ncu_dmu_pio_data[36]
  j2d_p_cmd[0]=  ncu_dmu_pio_data[37] && !ncu_dmu_pio_data[36] ||
                !ncu_dmu_pio_data[37] && ncu_dmu_pio_data[36]
  j2d_p_ctag[10:0]=  {ncu_dmu_pio_data[59:56],ncu_dmu_pio_data[46:40]}
  j2d_p_bmsk[15:0]= if ncu_dmu_pio_data[60] == 0 {// writes
         if ncu_dmu_pio_data[3] == 1  then
                {8'b0,ncu_dmu_pio_data[55:48]}
                else if ncu_dmu_pio_data[3] == 0  then
                {ncu_dmu_pio_data[55:48],8'b0}
                }
         else ncu_dmu_pio_data[60] == 1 {// reads
         if ncu_dmu_pio_data[3] == 1 && ncu_dmu_pio_data[50] = 0 then
                {8'b0,bytemask}
         else if ncu_dmu_pio_data[3] == 0 && ncu_dmu_pio_data[50] == 0 then
                {bytemask,8'b0}
         else if  ncu_dmu_pio_data[50] == 1 then// 16 byte pio reads
                                                16'b1;
                }
            where bytemask is a string of 1's equal to the byte count in
ncu_dmu_pio_data[50:48] starting at the address specified in
ncu_dmu_pio_data[35:0].
```

Note – The bmsk for reads is used by the DSN to determine how to align the returning PIO read data, on writes only 8 byte writes are allowed.

Note – j2d_p_xx(cmd's only, not data) are delayed by 1 clock from ncu_dmu_piodata[63:0]

8.17.2.5 NCU-DSN Timing Diagram

FIGURE 8-8 NCU-DSN Timing Diagram



A: Last write packet, all PIO credits are used, stop issuing any PIO to DMU;
B: DMU sends the write dequeue signal to NCU, as the write data is pulled out of DOB;
C: NCU adds credit back to its scoreboard, and issues a PIO RD to DMU at the next cycle;
D: NCU pulls PIO rd completion from SIU and adds credit back to its scoreboard;
E: NCU issues a mmu write invalidate, the vector is on the ncu_dmu_piodata[63:0] bus, this does not consume a PIO credit;
F: NCU issues another PIO request;

8.17.2.6 NCU to DSN Command Header Info

TABLE 8-44 NCU to DSN Command Header Bit Definitions

Header cycle ncu_dmupio_data [msb:lsb]	Name	Description
[63:61]	reserved	Must be zero
[60]	PIO read	1 = PIO reads 0 = PIO write
[59:56]	Credit id	Credit id issues with the PIO command, returned dmu_ncu_wrack_tag[3:0] for PIO writes, and in the SIU header for rd completions
[55:48]	Byte count/Byte mask[7:0] data is big endian, but bmsk[0] is for bits[7:0] even though data byte 0 is data[127:120]	This field is identical to size' field from pcs packet For PIO read case: 8'bxxxx_x000: 1 Byte 8'bxxxx_x001: 2 Byte 8'bxxxx_x010: 4 Byte 8'bxxxx_x011: 8 Byte 8'bxxxx_x100: 16 Byte For PIO write case the 8bit mask indicates which of the 8B of store data should be updated.
[47:40]	NCU PIO ID	{1'b0,cpu_thrid[6:0]}
[39:38]	reserved	Must be 0
[37:36]	Command Mapping	11 = Memory space 64 10 = Memory space 32 01 = IO space(pa[28]==1'b1) 00 = Configuration space (pa[28]==1'b0)
[35:0]	PA[35:0]	36 bit PA address from CPU, note this is a full byte address

8.17.2.7 NCU to DSN Header for MMU Invalidates

When the NCU sends an IOMMU invalidate the ncu_dmu_data[63:0] contains the physical address to invalidate. The wires [39:6] will directly connect to j2d_mmu_addr[39:6].

TABLE 8-45 NCU to DSN Header Bit Definitions

Header cycle ncu_dmupio_data [msb:lsb]	Name	Description
[63:40]	N/A	
[39:6]	PA[39:6]	39 bit PA address from NCU CSR write
[5:0]	N/A	Assumed 0

8.17.3 DSN-DMU Interface

The DSN-DMU interface is left as is, pin list below, and the DSN block adapts the new SIU and NCU interfaces to this existing set of Signals.

TABLE 8-46 DSN-DMC Interface Pins

Signal name	direction	Description
Command Port		
d2j_cmd[3:0]	input	Dma/int request or pio rd completion command
d2j_addr[36:0]	input	Address of dma/int request
d2j_ctag[15:0]	input	Transaction tag for dma/int request or pio rd completion
d2j_cmd_vld	input	Valid signal for d2j_(cmd,addr,ctag)
Data Port		
d2j_data[127:0]	input	Data for dma wr/int request or pio rd completion
d2j_bmsk[15:0]	input	Bytemask for dma wr/int request
d2j_data_par[4:0]	input	Parity for dma wr/int request or pio rd completion data/bmsk
d2j_data_vld	input	Valid signal for d2j_(data,bmsk,data_par)
CTM: DMA Wrack Port		
j2d_d_wrack_tag[3:0]	input	Transaction tag for dma wrack
j2d_d_wrack_vld	input	Valid signal for j2d_d_wrack_tag
CTM: PIO Wrack Port		
d2j_p_wrack_tag[3:0]	output	Transaction tag for PIO wrack

TABLE 8-46 DSN-DMC Interface Pins *(Continued)*

Signal name	direction	Description
d2j_p_wrack_vld	output	Valid signal for d2j_p_wrack_tag
CRM Command Completion Port		
j2d_di_cmd[1:0]	output	Dma/int response cmd
j2d_di_ctag[15:0]	output	Transaction tag for dma/int response
j2d_di_cmd_vld	output	Valid signal for j2d_di_(cmd,ctag)
CRM Command Request Port		
j2d_p_cmd[3:0]	output	Pio req cmd
j2d_p_addr[35:0]	output	Address of pio req
j2d_p_bmsk[15:0]	output	Bytemask for pio req
j2d_p_ctag[10:0]	output	Transaction tag for pio req
j2d_p_cmd_vld	output	Valid signal for j2d_p_(cmd,addr,bmsk,ctag)
CRM Data Completion Port		
j2d_d_data[127:0]	output	Dma rd response data
j2d_d_data_par[3:0]	output	Parity for dma rd response data
j2d_d_data_err	output	Status of dma rd response data
j2d_d_data_vld	output	Valid signal for j2d_d_(data,data_par,data_err)
CRM Data Request Port		
j2d_p_data[127:0]	output	Pio wr data
j2d_p_data_par[3:0]	output	Parity for pio wr data
j2d_p_data_vld	output	Valid signal for j2d_d_(data,data_par)
Ring Interface (csr's are accessed through the NCU to DSN ucb interface, the DSN converts the ucb protocol to the ring protocol)		
j2d_csr_ring_out[31:0]	output	Csr ring input from JBC
d2j_csr_ring_in[31:0]	input	Csr ring output to JBC
Interrupts (these will need to be tied off in the DMU, the NCU will handle these functions)		
j2d_jbc_int_l	output	Jbu interrupt
j2d_i2c0_int_l	output	Internal interrupt
j2d_i2c1_int_l	output	Internal interrupt
j2d_jid_sel	output	
j2d_ext_int_l[19:0]	output	External interrupts from pins
Interrupts (interrupts are concentrated in the IMU and then sent out as data packets on the cmd interface.)		

TABLE 8-46 DSN-DMC Interface Pins *(Continued)*

Signal name	direction	Description
Mondo and MSI interrupts are sent as data packets on the same wires as dma writes		
MMU snoop interface (only needs to support CSR invalidates)		
j2d_mmu_addr_vld		ncu_dmu_mmu_addr_vld
d2j_tsb_base[42:13]		n. a.
d2j_tsb_enable		n. a.
d2j_tsb_size[3: 0]		

8.18 Pin Mapping

This table shows the signal mapping between the DMU pin name and the new SIU or NCU name.

TABLE 8-47 Pin Mappings from Existing DMC to DSN

DMU name	SIU/NCU name	Description
Command Port		
d2j_cmd[3:0]	dmu_sii_data[127:0]	These DMU Signals are placed in a header when d2j_cmd_vld is asserted.
d2j_addr[36:0]		
d2j_ctag[15:0]		
d2j_cmd_vld	dmu_sii_hdr_vld	
Data Port		
d2j_data[127:0]	dmu_sii_data[127:0]	
d2j_bmsk[15:0]	dmu_sii_be[15:0]	
	dmu_sii_be_parity	
d2j_data_par[4:0]	dmu_sii_parity[7:0]	Newly constructed and interleaved
d2j_data_vld	dmu_sii_hdr_vld	
CTM: DMA Wrack Port		
j2d_d_wrack_tag[3:0]		
j2d_d_wrack_vld		
CRM Command Completion Port		

TABLE 8-47 Pin Mappings from Existing DMC to DSN *(Continued)*

DMU name	SIU/NCU name	Description
j2d_di_cmd[1:0]		Derived from sio_dmu_data[127:0] header when sio_dmu_hdr_vld is asserted.
j2d_di_ctag[15:0]		
j2d_di_cmd_vld		
CRM Data Completion Port		
j2d_d_data[127:0]	sio_dmu_data[127:0]	
j2d_d_data_par[3:0]	sio_dmu_parity[1:0]	
j2d_d_data_err		Returned in the header
j2d_d_data_vld		
CRM Data Request Port		
j2d_p_data[127:0]	ncu_dmu_pio_data[63:0]	DSN must gather 64 bit data from the NCU and translate into 128 bit data for the DMU.
j2d_p_data_par[3:0]		Will the NCU provide parity?
j2d_p_data_vld		

8.19 RAS

The DSN will follow the SOC RAS ERROR Reporting Specification.

Most of the functionality required for this specification will be implemented in the DSN block. DMC internal errors such as parity on the internal rams will be reported through the existing DMC mondo interrupt mechanism. Also note that the parity is generated on an interleaved basis, i.e. p0 = parity on d0, d2,d4... p1 = parity on d1,d3,d5....

8.19.1 DSN/SII-SIO RAS Interface

The RAS features for this interface include:

1. The DMA write and PIO rd return fifo's have 1 parity bit per 32 data bits, and the SOC RAS spec requires 2 parity bits per 32. Thus parity will be checked on this bus, errors signaled and new 16 bit parity regenerated before sending the data to the SII. Any parity errors discovered at the DMC-DSN interface on data from the diu rams will be signaled by re-generating correct 32 bit parity and then flipping parity bit 1. The SII will then discover this and signal an error to the NCU.

2. DMA read return data will have 2 parity bits per 32 bits of data, so parity will be checked in the DSN, errors signaled and 1 parity bit per 32 bits will be regenerated. The dmu_sii_be[15:0] will have a separate parity bit.
3. ECC will be generated on the CTAG to the SIO, and parity on the address in the header.
4. Parity will be checked on the returning DMA write credit.
5. ECC will be checked on the CTAG DMA read return. If a ue is discovered on the ctag and bit 81 of the siu to dmu hdr is set, this error will not be signaled to the ncu on dmu_ncu_ctag_ue because a previous block has already signaled a ue for this condition.
6. Dedicated error and force error wires from the NCU will be added
7. Any ue on returning credit_id's will cause the DSN to block the return of that particular transaction back to the DMU, i.e. Dma write credit return, interrupt credit return, or dma read return header UE or ctag ue. This may cause the DMU to hang and should be considered a fatal error. SW will then have to sort out any fixes. This will mean these credits which have ue's will not get removed from the scoreboard. SW can read the syndrome register in the NCU but it may not accurately reflect the bad credit_id, since it may have been the corrupted data which caused the error. SW can also read the scoreboard registers in the DMC.

8.19.2 DSN/NCU RAS Interface

The RAS features for this interface include:

1. Interrupt response parity will be checked. If an error is encountered, the return from DSN to DMU for this interrupt response will be dropped.
2. Dedicated error and force error wires from the NCU will be added

8.19.3 DMC Internal RAS

Internal ram parity errors, those on the devtsb or tdb rams will be signaled to the NCU as mondo interrupts with an internal csr register logging which error occurred.

8.19.4 RAS Interface Signals

These are Signals between DSN and NCU/SII/SIO, in addition there are RAS bits in the header from DSN to SII and returning completion headers, and in the header from NCU to DSN. The Signals are listed here for the convenience of the reader, they are also listed previously in the section where there are used.

TABLE 8-48 RAS Signals

Signal name	direction	Description
DSN to SII RAS Signals		
dmu_sii_parity[7:0]	output	2 odd parity bits per 32 bits calculated as follows: dmu_sii_parity[0] on dmu_sii_data[0,2,4..30] dmu_sii_parity[1] on dmu_sii_data[1.3.5..31] dmu_sii_parity[2] on dmu_sii_data[32,34,36...62] dmu_sii_parity[3] on dmu_sii_data[33,35,37...63] dmu_sii_parity[4] on dmu_sii_data[64....94] dmu_sii_parity[5] on dmu_sii_data[65...95] dmu_sii_parity[6] on dmu_sii_data[96...126] dmu_sii_parity[7] on dmu_sii_data[97....127]
dmu_sii_be_parity	output	dmu_sii_be_parity is on dmu_sii_be[15:0]
Note: d2j_data[127:0] parity errors will be signaled to the SII by flipping dmu_sii_parity[1]		
SII to DSN RAS Signals		
sii_dmu_wrack_par	input	Odd parity on sii_dmu_wrack_tag[3:0]
SIO to DSN RAS Signals		
sio_dmu_parity[7:0]	input	2 odd parity bits per 32 bits calculated as follows: sio_dmu_parity[0] on sio_dmu_data[0,2,4..30] sio_dmu_parity[1] on sio_dmu_data[1.3.5..31] sio_dmu_parity[2] on sio_dmu_data[32,34,36...62] sio_dmu_parity[3] on sio_dmu_data[33,35,37...63] sio_dmu_parity[4] on sio_dmu_data[64...94] sio_dmu_parity[5] on sio_dmu_data[65....95] sio_dmu_parity[6] on sio_dmu_data[96....126] sio_dmu_parity[7] on sio_dmu_data[97....127]
Note: any detected parity errors will be signaled to DMC by asserting j2d_d_data_err synchronous with j2d_d_data		
NCU to DSN RAS Signals		
ncu_dmu_mondo_id_par	input	Odd parity on ncu_dmu_mondo_id[5:0]
dmu_ncu_wrack_par	output	Odd parity on dmu_ncu_wrack_tag[3:0]

TABLE 8-48 RAS Signals *(Continued)*

Signal name	direction	Description
DSN to NCU error reporting Signals		
dmu_ncu_d_pe	output	Indicates parity error on DMA rd data
dmu_ncu_siicr_pe	output	Indicates parity error on dma write credit ack
dmu_ncu_ctag_ue	output	Indicates ue on dma read return ctag, signaled only if ncu_dmu_ctag_uei is asserted, or if a ue was discovered on the ctag bits and bit 81 of the siu to dmu header was 0. only asserted during valid transactions.
dmu_ncu_ctag_ce	output	Indicates ce on dma read return ctag
dmu_ncu_ncucr_pe	output	Indicates parity error on mondo ack
dmu_ncu_ie	output	dmc internal error (not used in N2)
NCU to DSN parity error injection Signals		
ncu_dmu_d_pei	input	Force DMA read return pe
ncu_dmu_siicr_pei	input	Force DMA write credit return pe
ncu_dmu_ctag_uei	input	Force DMA read return header ctag ue
ncu_dmu_ctag_cei	input	Force DMA read return header ctag ce
ncu_dmu_ncucr_pei	input	Force DMA read return header ctag ce
ncu_dmu_iei	input	Force pe on DMC internal, forces parity errors on the devtsb and tdb rams in DMU/IOMMU. The error reporting is done with mondo 62 and status bits within the DMU.

The ncu_dmu_iei bit is used for parity errors on the rams within the dmu_dmc/dmu_mmu block. These are the devtsb and tdb rams. If this bit is asserted a parity error is forced when a csr write occurs to these rams. Then when the entry within these rams is accessed a parity error will be generated when the ram is read. This allows the test to more easily control what and when to cause a parity error. The tsb ram is programmed using the MMU TTE CACHE DATA REGISTER (0x648000-0x6448ff8), the devtsb ram is programmed using the MMU DEV2IOTSB Registers (0x649000-0x6449078).

8.19.5 Error Cases

TABLE 8-49 DSN Error Cases

Event Detector	Information Capture	Reporting Mechanism	Impact
DMA write data parity error	None in DMU	Generate bad parity on DMU->SII data, SII reports DMA write errors, logs address	DMA write is squashed with bad ecc on data
PIO rd return/Interrupt parity error	None in DMU	Generate bad parity on DMU->SII data, SII passes to NCU which logs, sends back to core	PIO loads get precise trap in core, interrupts are logged in NCU
ECC error on CTAG from DMU to SII	None in DMU	SII checks DMA and logs, passes to NCU for PIO read and interrupts which logs.	Single bit ecc errors are corrected, double bit errors cause writes to be squashed.
Parity on address in header from DMU to SII	None in DMU	SII reports, destination is guessed and packet is passed on in error.	DMA writes are squashed by clearing byte enables in SII
Parity on cmd field of DSN->SII header	None in DMU	SII reports, destination is guessed and packet is passed on in error.	SII squashes any writes
DMU->SII TO PIO rd cpl only	None in DMU	SII Sends to NCU	PIO rd cpl has timed out NCU handles
DMU->SII UnMapped PIO rd cpl only	None in DMU	SII Sends to NCU	PIO rd cpl with address errors are reported back to the NCU here, NCU interfaces with cores to handle.
DMA read data return parity error, pe detected locally, bad parity was forced by l2$.	Poison bit sent to ILU	Single error bit to NCU, which logs, optional interrupt, poisoned data is detected at endpoint which reports back to initiating thread.	Parity is regenerated correctly, poison bit forwarded to ILU
DMA write credit return parity error	None in DMU	DSN Signals NCU with error bit. DSN drops this ack back to DMC	One less credit id to use in DMC, no corruption but possible DMU hang.

TABLE 8-49 DSN Error Cases *(Continued)*

Event Detector	Information Capture	Reporting Mechanism	Impact
ECC error in header CTAG from SIO to DMU on DMA rd return	None in DMU	2 error bits sent to NCU for logging and optional interrupt Packet is never returned, endpoint detects this and notifies the thread.	Note that if an ecc ue is detected locally and bit 81 of the siu to dmu header is set the error from dmu to ncu (dmu_ncu_ctag_ue) will not be set.
Parity on PIO write credit return to NCU	None in DMU	NCU checks and logs	On error the credits are not released within the NCU.
Parity error on MONDO ACK from NCU	None in DMU	Single error bit to NCU which logs, optional interrupt. DSN drops credit return to IMU	One less interrupt credit in IMU, interrupts slow down.

8.19.6 IOMMU RAS

TABLE 8-50 IOMMU Error Cases

Event Detector	Information Capture	Reporting Mechanism	Impact
Parity on devtsb ram read	Single error bit, with secondary	Error bit in DMU status register, with optional interrupt if enabled.	Ingress transaction is nullified
Parity error on tdb ram	Single error bit, with secondary	Error bit in DMU status register with optional interrupt if enabled.	Ingress transaction is nullified
Error on tablewalk return.	Multiple error bits, with secondary	Error bits in MMU Error register.	Ingress transaction is nullified

Note – To force a parity error out of the devtsb or tdb ram, use the NCU force error bit described above.

8.19.7 Why is there no Syndrome Register in DSN?

Consider these cases:

1. DMA write data parity error.

i. SII logs address, write completes to l2$ with byte enables off, SW can figure out what device was doing the write from SII address syndrome.

2. PIO rd cpl and interrupt data parity error.

i. rd cpl, data is passed back to core and load buffer and it would log the address for reads (precise trap).

ii. For interrupts the NCU logs

3. DMA rd data return

i. DMU poisons the data, and the endpoint which gets the data should report this to the thread it is working for. nothing is hung or dropped. endpoint reports.

4. header address, cmd and ctag ecc ue's,

i. On ingress NCU will log

ii. On egress (DMA rd cpl's) the DSN drops this cacheline and does not return the credit id and data to the DMC. Since the DMU orders the DRC's it is possible multiple transactions will accumulate and thus lock up the DMU, thus these errors are fatal from the DMU perspective.

5. Interrupt mondo ack parity error

i. The DSN drops this mondo ack, the interrupt id never gets returned to the IMU so it cannot be reused and we have 1 less(4 total) id to process interrupts. sw knows from the NCU interrupt the DMU has 1 less interrupt credit

6. dma write credit ack,

i. The DSN drops this ack back to DMC and the DMC has 1 less credit id to use, but should not cause any error corruption since the DMA write itself has already gone before.

7. pio write credit ack

i. The NCU drops it, and does not reuse that credit it, so it will have 1 less credit id to work with, should not cause any corruption

For the IOMMU ram parity errors the address is logged in the MMU translation fault register.

8.20 Resets

The DSN block will need reset to clear the CSR logic, headers, valid bits and the interrupt fifo pointers on POR and WMR.

Refer to the individual CSR definitions in the *OpenSPARC T2 Programmer's Reference Manual* for information on any particular CSR bit as to POR or WMR reset.

8.21 CSR's

The DSN block will not have any internal control/status registers, but will include a ccc controller for the DMU csr ring. The DSN will incorporate the ucb logic common to the NIU (with slight modifications). The ccc logic from the jbc will then be interfaced to the ucb logic, and the DMU csr ring will be generated out to the DMU.

The NCU will decode all CSR accesses from the cores and only send transactions to the DSN which fall within the DMU/PEC CSR ring. The offsets for these registers will remain the same. The decode for fast/med/slow will also move to the DSN block.

Refer to the *PCI-ex Programmer's Reference Manual* for register definitions and addresses.

The ucb interface accepts CSR requests, buffers them and presents these requests in order to the CCC interface. The CCC interface will have only 1 outstanding CSR transaction on the DMU/PEC CSR ring at any given time. Writes complete without response, CSR reads will always respond, either with data or error packet.

There will be no JTAG interface to the DSN CSR block, JTAG access will be provided in the NCU block.

Note – The buf_id_in[1:0] = 2'b00 for cpu access, buf_id_in[1:0]= 2'b01 for JTAG access.

The structure of the CSR ring in the DMU is:

8.21.1 CSR Address Decoding

The DMU address decoding will be in 3 steps.

1. First the NCU will decode pa[39:32] == 0x88 as a CSR access intended for the DMU blocks and only send these CSR accesses to the DMU/DSN.
2. Then the DSN block will decode pa[19:16] as follows for the ring:
3. Then as the packet flows around the ring, each DCC will sample pa[26:0] to determine if a particular packet is meant for itself and respond.

Note – See the *OpenSPARC T2 Programmer's Reference Manual* for a description of each register and its address.

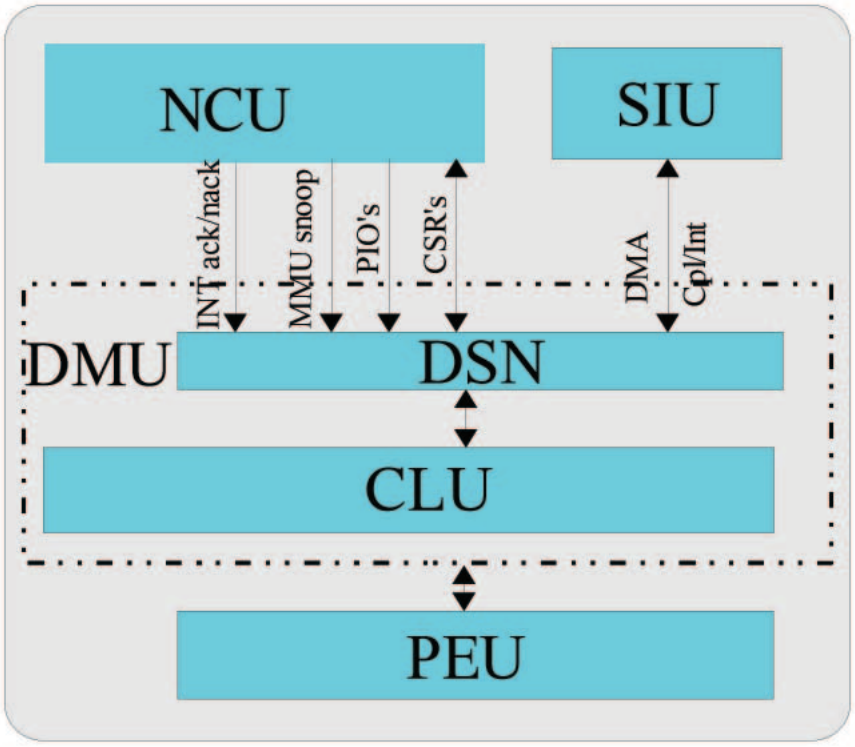
8.21.2 CSR Related Pins

TABLE 8-51 CSR Related Pins

Signal name	direction	Description
Ucb interface downstread		
ncu_dsn_vld	input	ncu_dsn_data[31:0] is valid,
ncu_dsn_data[31:0]	input	Csr hdr/data
dsn_ncu_stall	output	Dsn csr buffers are full, 1=0 stop sending to NCU
Ucb interface upstread		
dsn_ncu_vld	output	Valid on csr read return
dsn_ncu_data[31:0]	output	Csr read return data from DMU/PEU
ncu_dsn_stall	input	NCU stalls DMU/PEU csr read return data when asserted
CSR ring to DMU/PEU		
j2d_csr_ring_out[31:0]	output	Csr ring to DMU
d2j_csr_ring_in[31:0]	input	CSR ring return from DMU

8.21.3 CSR Block Diagram

FIGURE 8-9 CSR Block Diagram



8.22 Transaction Ordering

This section describes the PIO and CSR ordering within the DSN and DMU blocks. The ordering between PIO's/CSR's and DMA read/writes is not defined except that an outstanding PIO read will "pull" in all outstanding DMA writes.

It appears that the cores/crossbar/NCU follow TSO for all loads and stores for a particular thread up until the point at which an entry is dequeued from the main fifo in the NCU. There is no ordering between threads.

The DMU has two interfaces from the NCU block after its main ld/st fifo:

1. PIO reads and writes
2. CSR reads and writes

The PIO and CSR interfaces are independent to the DSN/DMU blocks. But, since the core logic load unit only supports one outstanding load per thread, PIO and CSR loads are by definition ordered within a thread. However, the cores support multiple outstanding stores. CSR stores are all placed in a fifo prior to being dispatched onto the CSR ring and the CSR ring only supports 1 outstanding transaction at a time thus all CSR stores will be ordered with respect to each other, but not PIO stores or loads.

The only exception is the MMU PA invalidates which are PIO stores directly from a decode in the NCU, these do not go through the CSR ring. The MMU PA invalidates will have a deterministic pipeline through the DSN/DMU. Thus SW may determine ordering of invalidates and other PIO/CSR's.

Also note, that a CSR read from a particular CSR ring will guarantee that all previous CSR writes to that particular ring will have completed.

8.23 DEBUG Features

This will consist of 3 new features:

1. Quiescing of the DMU/SII,SIO interfaces based on a request initiated from debug.v.
2. Implement debug busses A,B for DMU and send out to debug.v. The existing Signals used in the DMC debug busses will continue to be used and new Signals from DSN will be sent to the DMC block and muxed out. New DSN Signals are listed below.
3. On any PCI_EX error, qualify with Debug_Trig_en (new csr bit in DMU) and send out to debug.v.

Refer to the *OpenSPARC T2 Programmer's Reference Manual* register ERR NONFATAL Mapping register address 0x630008 bit 62.

8.23.1 Quiescing DMU/SII/SIO Interface

It is assumed that the NCU will be drained before the DMU is signaled to quiesce. Then the debug.v block will send a signal to the DMU to quiesce. To manage this the DSN block will keep a 4 bit counter to track the number of outstanding DMA reads and writes, and a 2 bit counter to track the number of outstanding mondo interrupts. The DSN will signal the CLU block to stop sending transactions to the DSN. The DSN will then monitor the responses from the SII, SIO and NCU blocks,

i.e. Wait for all write acks, mondo acks and DMA read responses to complete, by checking the outstanding transaction counters. It will then signal the debug.v block that the interface is quiescent by asserting the signal dmu_dbg1_stall_ack.

8.23.2 Debug Busses

The DMU has existing A and B debug busses. These are 8 bit busses which are muxed together in the DMU CRU block. Additionally new Signals will be driven from the DSN block and muxed into the same outputs using spare decodes.

See the *OpenSPARC T2 Programmer's Reference Manual* DMU registers DMC debug select definitions(*OpenSPARC T2 Programmer's Reference Manual* registers DMC Debug Select Register for Ports A and B addresses 0x653000 and 0x653008) and the list of debug Signals in the *OpenSPARC T2 Programmer's Reference Manual*.

In addition the DMU will implement a test feature enabling a training sequence. The debug busses A and B will be forced to output a pattern of alternating 3 1's and 1 0 when the debug select busses are set to 0101.

8.23.3 All PCI-Ex Error Output

Within the DMU/IMU block a new signal will be created by "or'ing" mondo 62 and 63, "and'ing" with Debug_trig_en and sending out to the debug.v block a signal which indicates an error within the DMU, called dmu_dbg_err_event.

8.23.4 Debug Interface Signals

TABLE 8-52 Debug Ports

Signal name	direction	Description
Debug Signals to dbg.v block		
dmu_mio_debug_bus_a[7:0]	output	DMU debug bus A
dmu_mio_debug_bus_b[7:0]	output	DMU debug bus B
dmu_dbg1_stall_ack	output	Ack from DMU indicating DMU -> SII interface has quiesced.
dmu_dbg1_err_event	output	An error event occurred in DMU
Debug Signals from dbg.v block		
dbg1_dmu_stall	input	Request to stall/quiesce DMU -> SII interface
dbg1_dmu_resume	input	Request to resume packets on DMU -> SII interface

The debug ports are simply mux'ed versions of internal DMU/DSN Signals, which are then flopped and driven out to the dbg block.

The signal dbg_dmu_stall is asserted for 1 cycle by the dbg block, when the DMU is quiescent, it will assert dmu_dbg_stall_done for 1 cycle. At some later time the dbg block will assert the signal dbg_dmu_resume for 1 cycle to inform the DMU to resume normal operation.

8.23.5 DSN Debug Signals

TABLE 8-53 DSN Debug Signals

Signal name	Bit number	Description
Debug Signals for dbg a[7:0] sub_sel[01]		
ncu_dmu_vld	7	Ncu request CSR access
dmu_ncu_stall	6	Dmu stalls ncu csr read req
read_pending	5	Internal dsn csr read pending
write_pending	4	Internal dsn csr write pending
dmu_ncu_stall_a1	3	Internal dsn csr stall at head of queue
rd_nack_vld	2	Dsn to ncu csr read nack
dmu_ncu_vld	1	Dsn to ncu csr read return
ncu_dmu_stall	0	ncu_to dsn stall returning csr read data
Debug Signals for dbg b[7:0] sub_sel[01]		
arb_vld	7	Internal dsn csr pending to csr ring
req_vld	6	Csr ring req return, starts timer
acc_vio	5	Csr access violation (address)
rsp_vld	4	Csr read return
timeout	3	Csr read timeout
Cmnd[2]	2	Csr ring data0 cmd
Cmnd[1]	1	Csr ring data0
Cmnd[0]	0	Csr ring data0
Debug Signals for dbg a[7:0] sub_sel[02]		
dmu_sii_hdr_vld	7	dmu_header to sii, dma req, or pio cpl
dmu_sii_reqbypass	6	Asserted for pio rd cpl's
dmu_sii_datareq	5	Valid during hdr, 0=dma 1=write
dmu_sii_datareq16	4	0=write_64, 1=write_16byte
dsn_sii_hdr[126]	3	See dsn spec for values
dsn_sii_hdr[124]	2	See dsn spec for values
dsn_sii_hdr[123]	1	See dsn spec for values
dsn_sii_hdr[122]	0	See dsn spec for values
Debug Signals for dbg b[7:0] sub_sel[02]		

TABLE 8-53 DSN Debug Signals *(Continued)*

Signal name	Bit number	Description
sio_dmu_hdr_vld	7	Sio dma rd return
sii_dmu_wrack_vld	6	Dma write ack, credit_id returned
ncu_dmu_mondo_ack	5	Mondo ack
ncu_dmu_mondo_ack	4	Mondo nack
ncu_dmu_pio_hdr_vld	3	Ncu pio req
pio_read	2	Ncu pio req is a read
dmu_ncu_wrack_vld	1	dmu_returns pio write credit id
1'b0	0	spare
Debug Signals for dbg a[7:0] sub_sel[00],[03]-[3f] = 8'b0		
Debug Signals for dbg b[7:0] sub_sel[00],[03]-[3f] = 8'b0		

CHAPTER 9

Miscellaneous I/O (MIO) Specification

This chapter contains the following sections:

9.1 Overview

This document describes OpenSPARC T2 MIO (Miscellaneous I/O) block which holds majority of non-Serdes I/O's of the chip. The I/O's in MIO block fall broadly under the functional categories of clock, reset, test (scan and ramtest),ssi interface, process control (PCM) and efuse program enable. Most of the I/O's in MIO are on Boundary Scan chain under control of TCU. All the functional flops in MIO are connected on regular scan chain with scanin,scanout and flush reset capabilities under the control of TCU.

9.1.1 MIO Interface with System and Rest of OpenSPARC T2

MIO block interfaces with the system on one side and OpenSPARC T2 clusters on the other. MIO interfaces with the following clusters of OpenSPARC T2 : db0, db1, tcu, efu, fsr, psr, esr, ccu, ncu, rst.

The I/O's in MIO fall under the broad categories of clock, reset, test (scan and ramtest),ssi interface, PLL test , process control (PCM) , efuse program enable, Power Throttle and debug. The following table shows all the I/O's in MIO along with the I/O type, direction, destination/src clusters in OpenSPARC T2 along with signal names and functional category of the I/O's.

TABLE 9-1 MIO Pinlist

Pin Name	I/O Type	Direction	Function	Shared	Description & Frequency	Src/Dest OpenSPARC T2 Block & Signal name(s)
XAUI0_LINK_LED	cmos 1.1v	output	10G Enet Status	No	link status led, port 0. 0 Hz : A level Signal	Mac xaui_link_led_0
XAUI0_ACT_LED	cmos 1.1v	output	10G Enet Status	No	activity led, port 0 5 Hz core_clk/2to26	Mac xaui_act_led_0
XAUI1_LINK_LED	cmos 1.1v	output	10G Enet Status	No	link status led, port 1. 0 Hz : A level Signal	Mac xaui_link_led_1
XAUI1_ACT_LED	cmos 1.1v	output	10G Enet Status	No	activity led, port 1 5 Hz core_clk/2to26	Mac xaui_act_led_1
XAUI_MDC	cmos 1.1v	output	10GEnet Clock Signal	No	Clock Signal 2.5 Mhz	Mac mdc
XAUI_MDIO	Open drain 1.1 v	Bidi	10G Enet OD Tristate Config signal	No	OD Tristate signal 2.5 Mhz Data Rate	Mac mdoe, mdi Requires external pull-up resister. Mdoe connects to pulldown enable of the output driver. Input of the output buffer grounded. Mdi connected to output of input buffer .
TCK	cmos 1.1v	Input	Test	No	JTAG Test Clock 200 mhz	Tcu mio_tcu_tck
TDI	cmos 1.1v	Input	Test	No	JTAG Test Data In 200 mhz	Tcu mio_tcu_tdi
TDO	cmos 1.1v	Output	Test	No	JTAG Test Data Out 200 mhz	Tcu tcu_mio_tdo

TABLE 9-1 MIO Pinlist *(Continued)*

TMS	cmos 1.1v	Input	Test	No	JTAG Test Mode Select 200 mhz	Tcu mio_tcu_tms
TRST_L	cmos 1.1v	Input	Test	No	JTAG Test Reset 200 mhz	Tcu mio_tcu_trst_l
STCIQ	cmos 1.1v	Output	Serdes Test	No	SERDES STCI Scan Chain Data Out 200 mhz	Tcu tcu_mio_stciq
STCID	cmos 1.1v	Input	Serdes Test	No	SERDES STCI Scan Chain Data In 200 mhz	Tcu mio_tcu_stcid
STCICFG[1:0]	cmos 1.1v	Input	Serdes Test	No	SERDES STCI Scan Configuration 200 mhz	Tcu mio_tcu_stcicfg
STCICLK	cmos 1.1v	Input	Serdes Test	No	SERDES ATPG/STCI Scan Clock 200 mhz	Tcu mio_tcu_stciclk
TESTCLKT	cmos 1.1v	Input	Serdes Test	No	SERDES Bypass Clock for Transmit 200 mhz	FSR[7:0],ESR,PSR mio_fsr_testclkt[7:0]mio_psr_testclkt mio_esr_testclkt
TESTCLKR	cmos 1.1v	Input	Serdes Test	No	SERDES Bypass Clock for Receive 200 mhz	FSR[7:0],ESR,PSR mio_fsr_testclkr[7:0]mio_psr_testclkr mio_esr_testclkr
TESTMODE	cmos 1.1v	Input	Test	No	Puts OpenSPARC T2 in ATPG Scan/ Manufacturing Test Mode 200 mhz	TCU mio_tcu_testmode
PLL_TESTMODE	cmos 1.1v	Input	PLL test	No	Puts OpenSPARC T2 in PLL Testmode 200 Mhz	CCU mio_pll_testmode
DIVIDER_BYPASS	cmos 1.1v	Input	Test	No	Bypasses Clock Tree Dividers 200 mhz	TCU mio_tcu_divider_bypass

TABLE 9-1 MIO Pinlist *(Continued)*

PLL_CMP_BYPASS	cmos 1.1v	Input	Test	No	CMP Clock PLL Bypass 200 mhz	TCU mio_tcu_pll_cmp_bypass
PLL_DR_BYPASS	cmos 1.1v	Input	Test	No	DR Clock PLL Bypass 200 mhz	TCU mio_tcu_pll_dr_bypass
IMP_MON_PU	cmos 1.1v	Output	Debug	No	Imped. Monitor for pull-up Drivers.	Within MIO
IMP_MON_PD	cmos 1.1v	Output	Debug	No	Imped. Monitor for pull-down Drivers.	Within MIO
TRIGIN	cmos 1.1v	Input	Debug	No	Stop clock based on external event (asynchronous, to be synchronized in TCU)	TCU mio_tcu_trigin
TRIGOUT	cmos 1.1v	Output	Debug	No	Dbg Event Signal To Logic Analyzer 700 Mhz	TCU tcu_mio_trigout
PMI[1:0]	cmos 1.1v	Input	PCM	No	process control monitor input Level Signal	PCM mio_pcm_pmi[1:0]
PMO	cmos 1.1v	Output	PCM	No	process control monitor output Level Signal	PCM pcm_mio_pmo
PGRM_EN	cmos 1.1v	Input	Efuse	No	Efuse Program enable Level Signal	EFU mio_efu_prgm_en
PB_RST_L	cmos 1.1v	Input	Reset	No	Like OpenSPARC J_RST_L Level Signal	RST mio_rst_pb_rst_l
BUTTON_XIR_L	cmos 1.1v	Input	Reset	No	Externally Initiated Reset Level Signal	RST mio_rst_button_xir_l
PEX_RESET_L	cmos 1.1v	Output	Reset	No	Reset to External PCI Express switch and devies Level Signal	RST rst_mio_pex_reset_l
PWRON_RST_L	cmos 1.1v	Input	Reset	No	Power On Reset Level Signal	RST mio_rst_pwron_rst_l

TABLE 9-1 MIO Pinlist *(Continued)*

FATAL_ERROR	cmos 1.1v	Output	Reset	No	Fatal Error has ocurred in OpenSPARC T2 Duration of warm_reset @ sys_clk	RST rst_mio_fatal_error
VREG_SELBG_L	cmos 1.1v	Input	PLL Control . When selected makes PLL use BandGap Voltage Source	No	BandGap Select Static(on or off)	CCU mio_ccu_vreg_selbg_l
SSI_MOSI	cmos 1.1v	Output	SSI Boot	No	SSI Master Out, Slave In 50 Mhz	NCU ncu_mio_ssi_mosi
SSI_MISO	cmos 1.1v	Input	SSI Boot	No	SSI Master In, Slave Out 50 Mhz	NCU mio_ncu_ssi_miso
SSI_SCK	cmos 1.1v	Output	SSI Boot	No	SSI Clock 50 Mhz	NCU ncu_mio_ssi_sck
EXT_INT_L	cmos 1.1v	Input	SSI Boot	No	External Interrupt Pin 50 Mhz	NCU mio_ncu_ext_int_l
BURNIN	cmos 1.1v	Input	PCM	No	Sets Burnin Mode for PCM Modules Level Signal	PCM mio_pcm_burnin
PLL_CHAR_OUT[1:0]	cmos 1.1v	Output	PLL Test	No	PLL Char Out bus	CCU ccu_mio_pll_char_out[1:0]
PWR_THRTTL_0[2:0]	cmos 1.1v	Input	Power Throttle	No	Power Throttle for Sparcs : 0,1,5,4 4 Hz (50 mhz clk from SP)	SPC's : mio_spc_pwr_throttle_0[2:0]

TABLE 9-1 MIO Pinlist *(Continued)*

PWR_THRTTL_1 [2::0]	cmos 1.1v	Input	Power Throttle	No	Power Throttle for Sparcs : 2,3,7,6 4 Hz (50 mhz clk from SP)	SPC's : mio_spc_pwr_throttle_1[2:0]
DBG_CK0	cmos 1.1v	Output	Debug	No	Debug Port Output Clock 350 Mhz	Within MIO
DBG_DQ[165:0]	cmos 1.1v	Bidi	Debug	Yes*	OpenSPARC T2 Debug Port 700 Mhz	DBG1 dbg1_mio_dbg_dq

TABLE 9-2 shows the sharing of pins between debug and other functionality . The sharing scheme is further explained in detail in section 5.2 of the document.

TABLE 9-2 Sharing of Debug Pins with Other Pins

Pin Name	Shared With Pin :/Pin Description	Select & Drive Enable	Src/Dest OpenSPARC T2 Block & Signal name(s)
165:161	RST_STATE[4:0] : Output Reset State from RST block.	Drive En : dbg1_mio_drv_en_op_only	DBG1 dbg1_mio_sel_soc_obs_mode
160	Not Shared		
159	SCAN_OUT31 : Output SERDES ATPG Scan Chain Data Out	Sel : mio_tcu_testmode Drive En : dbg1_mio_drv_en_muxtest_op	TCU tcu_mio_scan_out31
158	SCAN_IN31 : Input SERDES ATPG Scan Chain Data In	Drive En : dbg1_mio_drv_en_muxtest_inp	TCU mio_tcu_scan_in31
157	NIU_DBG_DAT[31] :Output PLL_CHAR_IN : Input Niu Debug port bit 31. PLL Char In	Sel : dbg1_mio_sel_niu_debug_mode Drive En : dbg1_mio_drv_en_muxtestpll_inp	NIU niu_mio_debug_data[31] CCU : mio_ccu_pll_char_in

TABLE 9-2 Sharing of Debug Pins with Other Pins *(Continued)*

156:149	Shared with both output and input pins. Output Pins : NIU_DBG_DAT[30:23] Input Pins: PLL_DIV2[5:0] PLL_TRST_L PLL_CLAMP_FLTR Shared between output pins only NIU_DBG_DAT[22:21]	Drive en = dbg1_mio_drv_en_muxtestpll_inp for 156:149 and 146: 103. 102:85, 148:147 have dbg1_mio_drv_en_op_only Sel : 156: 124 - dbg1_mio_sel_niu_debug_mode 123:91 - dbg1_mio_sel_pcix_debug_mode Pin descriptions : Outputs : NIU Debug Signals [30:0] NIU Debug Clocks[1:0] Debug Bus A from DMU[7:0] Debug Bus B from DMU[7:0] Debug Bus A from PEU[7:0] Debug Bus B from PEU[7:0] PEU Clock sent out on Debug Port Inputs : Input for PLL feedback Divider Async Reset in Testmode for PLL VCO runs at low freq PLL Div 4 External DR clk in PLL bypass External CMP clk in PLL Bypass Transition Test Mode Triggers Transition Capture Master Scan Clock	**NIU** : niu_mio_debug_data[30:0] niu_mio_debug_clock[1:0] **DMU** :dbg0_mio_debug_bus_a[7:0]dbg0_mio_debug_bus_b[7:0] **PEU** :peu_mio_debug_bus_a[7:0]peu_mio_debug_bus_b[7:0]peu_mio_debug_clk **CCU** : mio_ccu_pll_div2[5:0] mio_ccu_pll_trst_l mio_ccu_clamp_fltr mio_ccu_pll_div4[6:0]mio_ext_dr_clk mio_ext_cmp_clk **TCU** :mio_tcu_io_ac_testmode mio_tcu_io_ac_testtrig mio_tcu_io_aclk mio_tcu_io_bclk mio_tcu_io_scan_in[30:0]
148:147	Shared between output and input pins . Output pins : NIU_DBG_DAT[20:0] NIU_DBG_CLK[1:0] DMU_DBG_BUS_A[7:0] DMU_DBG_BUS_B[7:0] PEU_DBG_BUS_A[7:3] Input Pins : PLL_DIV4[6:0] PLL_EXT_DR_CLK PLL_EXT_CMP_CLK AC_TESTMODE AC_TESTRIG ACLK BCLK SCAN_IN[30:0] Shared between output pins only PEU_DBG_BUS_A[2:0] PEU_DBG_BUS_B[7:0] PEU_DBG_CLK		
146:103			
102:91	Not Shared		
90:85			

TABLE 9-2 Sharing of Debug Pins with Other Pins *(Continued)*

84 83 82:77 76:75 74	Input : PEU_CLK_EXT Scan Test Captures @ PEU Not shared Input : NIU_CLK_EXT[5:0] .Scan Test Captures @ NIU Not Shared Input : SCAN_EN	dbg1_mio_drv_en_muxtest_inp dbg1_mio_drv_en_op_only dbg1_mio_drv_en_muxtest_inp dbg1_mio_drv_en_op_only dbg1_mio_drv_en_muxtest_inp	TCU : mio_tcu_peu_clk_ext mio_tcu_niu_clk_ext[5:0] mio_tcu_io_scan_en
73:43	Outputs : SCAN_OUT[30:0] Scan Out Data	Sel : mio_tcu_testmode Drive_en : dbg1_mio_drv_en_muxtest_op	**TCU** : tcu_mio_pins_scan_out[30:0]
42:0	Outputs : DMO_SYNC DMO_DATA[39:0] Ram Test (Membist) Output MBIST_DONE Membist Status MBIST_FAIL Membist Fail	Sel : tcu_mio_jtag_membist_mode Drive En : dbg1_mio_drv_en_muxbist_op	**TCU** : tcu_mio_dmo_sync tcu_mio_dmo_data[39:0] tcu_mio_mbist_done tcu_mio_mbist_fail

TABLE 9-3 shows the functional categories and frequencies of the pins that are shared with the Debug Pins.

TABLE 9-3 Shared Pins Functionality and Frequencies

Pin Name	Functionality	Data Change rate
RST_STATE[4:0]	Debug	System Clock (in Rst block)
SCAN_OUT31	Serdes Test	200 Mhz
SCAN_IN31	Serdes Test	200 Mhz
NIU_DBG_DAT[31:0]	Debug	As specified by NIU_DBG_CLK[1:0]

TABLE 9-3 Shared Pins Functionality and Frequencies

NIU_DBG_CLK[1:0]	Debug	Upto 2 clks : 350 Mhz nominal, any of MAC clocks
DMU_DBG_BUS_A[7:0]	Debug	350 Mhz nominal
DMU_DBG_BUS_B[7:0]	Debug	350 Mhz nominal
PEU_DBG_BUS_A[7:0]	Debug	250 Mhz
PEU_DBG_BUS_B[7:0]	Debug	250 Mhz
PEU_DBG_CLK	Debug	250 mhz PEU clock
PLL_CHAR_IN	PLL Test and Characterization (CCU)	100 Mhz
PLL_DIV2[5:0]	PLL Test and Characterization (CCU)	100 Mhz
PLL_TRST_L	PLL Test and Characterization (CCU)	100 Mhz
PLL_CLAMP_FLTR	PLL Test and Characterization (CCU)	100 Mhz
PLL_DIV4[6:0]	PLL Test and Characterization (CCU)	100 Mhz
PLL_EXT_DR_CLK	PLL Test and Characterization (CCU)	100 Mhz
PLL_EXT_CMP_CLK	PLL Test and Characterization (CCU)	100 Mhz
AC_TESTMODE	Test	200 Mhz
AC_TESTRIG	Test	200 Mhz
ACLK	Test	200 Mhz
BCLK	Test	200 Mhz
SCAN_IN[30:0]	Test	200 Mhz
PEU_CLK_EXT	Test	200 Mhz
NIU_CLK_EXT[5:0]	Test	200 Mhz
SCAN_EN	Test	200 Mhz
SCAN_OUT[30:0]	Test	200 Mhz
DMO_SYNC	DMO	cmp_clk/1, 2, 4, 8, or 16 (Programmed in TCU)

TABLE 9-3 Shared Pins Functionality and Frequencies

DMO_DATA[39:0]	DMO	cmp_clk/1, 2, 4, 8, or 16 (Programmed in TCU)
MBIST_DONE	Ramtest (Membist)	cmp_clk/1, 2, 4, 8, or 16 (Programmed in TCU)
MBIST_FAIL	Ramtest (Membist)	cmp_clk/1, 2, 4, 8, or 16 (Programmed in TCU)

The sharing scheme is further explained in detail in section 5.2 of the document.

9.1.2 Internal Pullups/Pulldowns in MIO for Inputs

The following table shows the inputs in MIO that have pullups/pulldowns on them

Inputs with Pullups/Pulldowns in MIO.

Pin Name	Pullup/ Pulldown	Boundary Scan	Shared/Dedicated
TESTMODE	Pulldown	Yes	Dedicated
STCID	Pulldown	Yes	Dedicated
STCICFG[1:0]	Pulldown	Yes	Dedicated
STCICLK	Pulldown	Yes	Dedicated
TESTCLKT	Pulldown	Yes	Dedicated
TESTCLKR	Pulldown	Yes	Dedicated
PLL_TESTMODE	Pulldown	Yes	Dedicated
PLL_CHAR_IN	Pulldown	Yes	Shared
PLL_CLAMP_FLTR	Pulldown	Yes	Shared
PLL_DIV4[6:0]	Pulldown	Yes	Shared
PLL_DIV2[5:0]	Pullup	Yes	Shared
PLL_TRST_L	Pullup	Yes	Shared
TDI	Pullup	No	Dedicated
TMS	Pullup	No	Dedicated
TRST_L	Pullup	No	Dedicated

9.1.3 MIO Floorplan in OpenSPARC T2

MIO floorplan is still in the works in the context of OpenSPARC T2 . The floorplan will be captured when it becomes final.

9.1.4 MIO Clocking

MIO would be clocked off of cmp clock with io2x sync enables and with iol2clk . Both cmp clock ,iol2clk and io2x sync enables would be generated from cluster headers in MIO out of gclk and ccu_cmp_io2x_sync enable input signals from global clock tree CCU repectively . The signals that would get flopped in MIO fall under the following 3 categories :

Debug port signals from db1 module (166 wires @ cmp_clk launched off of io2x sync enables in db1)

Ramtest signals from tcu module (41 wires @ cmp clk /10 launched off of io2x sync enables in tcu).

Debug signals from DMU (16 wires @ iol2clk from db0 module)

Each I/O cell in MIO that is bi-directional or output only will contain two flops both clocked by the cmp_clk generated by MIO's cluster header (s) : one to latch the debug port signal on the io2x sync enable , the other to latch the ramtest signal on the io2x sync enable. Since the ramtest pins are shared with the debug pins, only one of these two flops will drive the output driver of the I/O cell at any time depending on whether the debug port has been enabled or testmode has been enabled (debug mode and testmode are mutually exclusive).

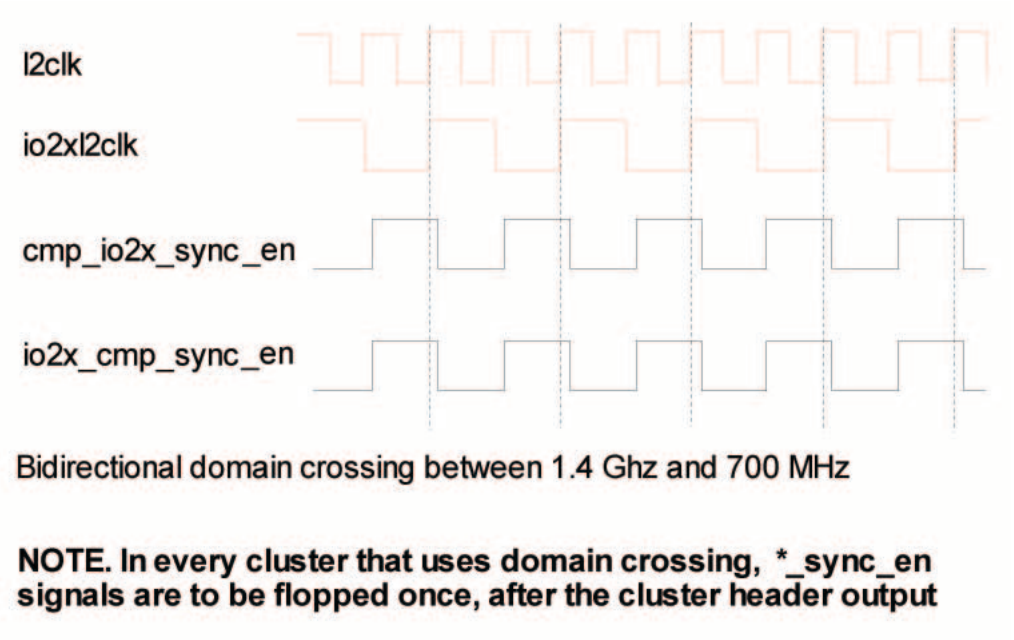
Note that there is a 3rd input to the driver which is a feedthrough path from certain OpenSPARC T2 clusters like NCU and TCU where the signal gets driven straight out of the source block in OpenSPARC T2 without any flop in MIO. W.r.t DMU , this 3rd leg also gets used after the DMU debug wires are retimed in MIO.

Thus each MIO output only or bi-di I/O cell will have a 3:1 mux before the functional input to the driver , with two legs of the mux coming from flops (debug and ramtest paths) and the 3rd leg coming as a feedthrough from some source block in OpenSPARC T2 or from retiming flops in MIO for DMU signals . This will be further illustrated in the descriptions of the I/O cells in subsequent sections of this document.

Since MIO contains 217 I/O cells which may be distributed over as much as 17 mm (depending on how the floorplan turns out to be), MIO will incorporate 4 cmp cluster headers with each cmp cluster header driving cmp_clk and io2x sync_enables to a group of I/O's . There will be 4 gclk inputs to MIO from the global clk tree feeding these 4 cluster headers .Also the cmp_clk coming out from each cluster

header will be distributed to all the I/Os being served by that cluster header over a clock distribution network with clock skews being maintained within a certain value consistent with other clusters in OpenSPARC T2 .

FIGURE 9-1 IO2X Sync Enable Timing w.r.t l2clk

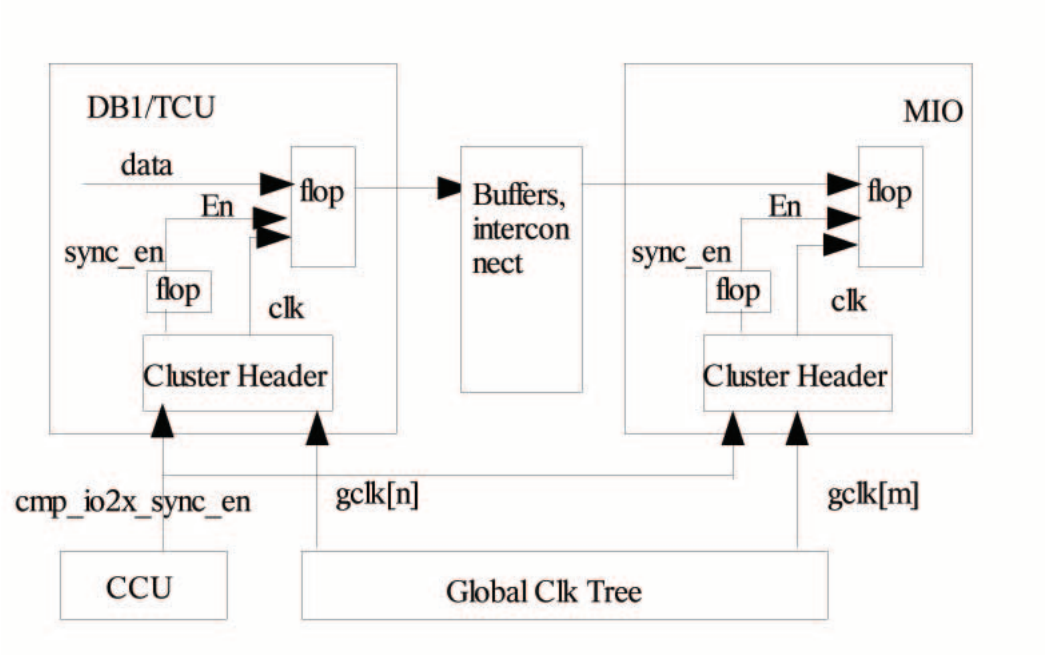


Each cmp cluster header incorporates two staging flops for the io2x sync enable . In addition, the io2x sync enable generated from each cmp cluster header will be flopped once in MIO before being distributed to all the I/O's in that group. This is also consistent with the usage model of sync enables in OpenSPARC T2 . (Please refer to OpenSPARC T2 CCU Spec). This staging will get done in the module called mio_syncreg_ctl. There are 4 instances of this module, on e per cmp cluster header.

The following timing diagram shows io2x sync enables w.r.t l2clk (cmp_clk).

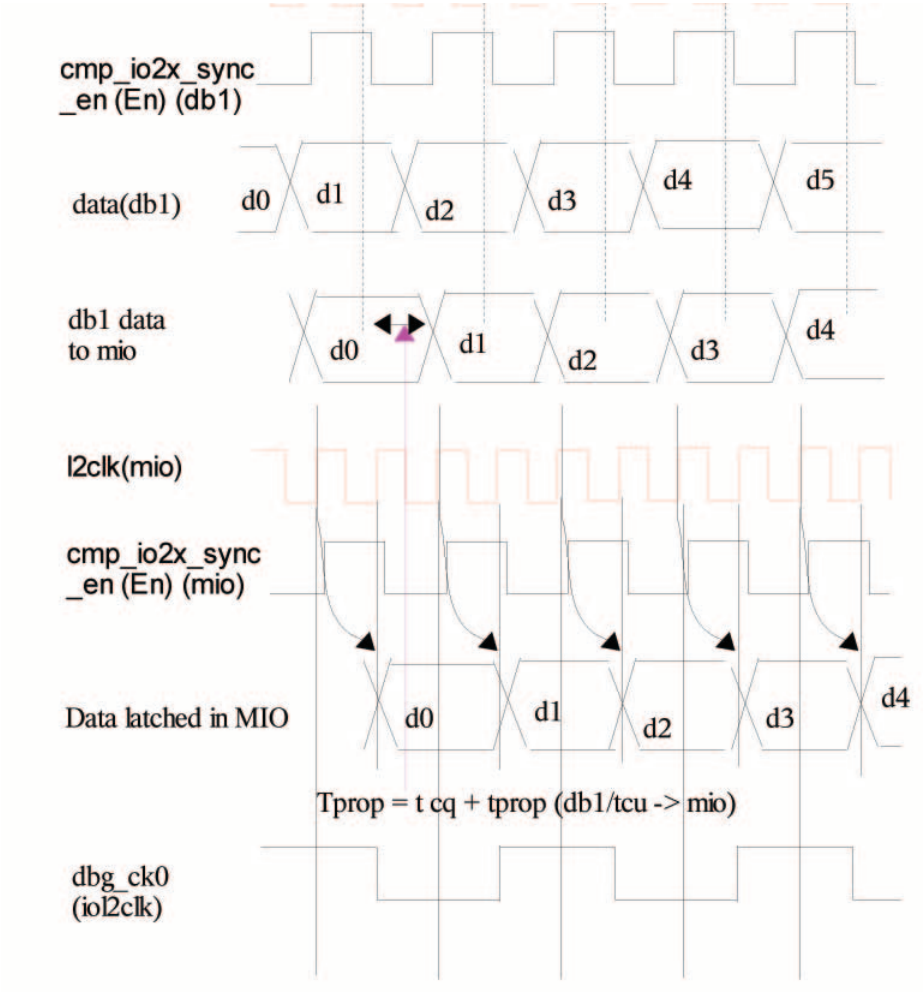
The following diagram shows the global clocking and sync enable distribution (from CCU) to DB1/TCU and MIO blocks.

FIGURE 9-2 Global Clock and sync enable distribution to DB1/TCU and MIO



The following timing diagrams show the scheme of launch of data from DB1/TCU off of cmp_clk with io2x sync enable and capture of same data in MIO I/O cell on cmp_clk with io2x sync_en.

FIGURE 9-3 Data Transfer from DB1 to MIO



Note – The Membist data transfer mechanism from TCU to MIO is identical to the one described above. The only difference is that since the membist data would be changing at the rate of cmp_clk/10 = 1.4 Ghz/10 = 140 mhz in TCU, each Membist data beat from TCU to MIO will be valid for a period of 5 cmp_io2x_sync_en pulses (5x140 = 700 mhz).

In addition to the cmp_clk cluster headers , MIO also incorporates a iol2clk cluster header by which iol2clk (350 mhz nominal) gets which also gets generated off of gclk_2 connected to MIO . This clock is used to retime the 16 DMU debug wires in MIO and is also directly fed as data input to the feedthrough leg of one of the I/Os in MIO to generate the debug port reference clock (DBG_CK0).

The idea is that the Logic Analyzer should use this clock as the reference clock when sampling the debug port signals. Eventhough the debug port signals are generated off of cmp_clk in DB1 which is generated from the same gclk tree , due to skew between the two gclks to MIO and DB1 and also skew across clock distributions in MIO and DB1, these data signals would have a skew among each other and also w.r.t DBG_CK0. Using training sequences on the debug port , the LA will be calibrated to account for this skew . Please refer to section 4.1 for description of the training sequnece . This deskewing in the LA has to be done only once and should hold valid across PVT variations as the propagational variations of the debug port wires across PVT and capacitive coupling related variations @ 700 mhz (nominal) is largely mitigated due to retiming of the debug signals in each I/O.

9.1.5 DFT Support for MIO

MIO implements the following DFT support for its I/O's :

Boundary Scan : All I/O's im MIO other than TCK,TDI,TMS,TRST_L, TDO, IMP_MON_PD, IMP_MON_PU,PMI, PMO,BURNIN,PGRM_EN implement boundary scan. Boundary scan is controlled by TCU through the following signals from TCU :

tcu_mio_bs_scan_in

tcu_mio_bs_highz_l

tcu_mio_bs_scan_en

tcu_mio_bs_clk

tcu_mio_bs_aclk

tcu_mio_bs_bclk

tcu_mio_bs_uclk

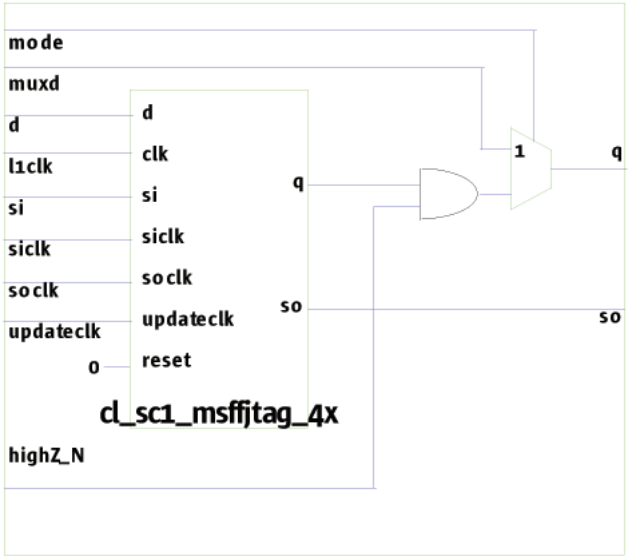
tcu_mio_bs_mode_ctl

All output only and bi-di I/O's of MIO that would be on Bscan chain would have Bscan cell on data out and output enable paths (as all output only I/O's in MIO would have tri-state control). Input only I/O's and bi-di I/O's that are on Bscan chain would have Bscan cell on receiver data in path.

The Bscan scheme in the MIO I/O cells is captured in detail in the descriptions of the MIO I/O cells in subsequent sections of this document. The Bscan cell is a library cell (cl_sc1_bs_cell2_4x) composed of a Boundary Scan Flop and a Mux to select the functional input vs Bscan flop output . Also incorporated in the Bscan scheme is support for wrap-back testing of the output driver by feeding the receiver output to the “d” input of the Bscan cell on the data out path.

The following figure shows the schematic for the cl_sc1_bs_cell2_4x cell which is the Bscan cell being used in MIO.

FIGURE 9-4 MIO's Boundary Scan Cell (cl_sc1_bs_cell2_4x) Schematic



Note – For all the dedicated input pins in MIO with Bscan, the “mode” port of the BS cell on the receiver is tied to 1'b1(enabling the “q” output of the cell to be only driven by the pin and not by the Bscan cell on an update). Thus the Bscan cell can perform a shift or capture , but can never do an update during Boundary Scan. Also, for the inputs that are shared with the debug pins, TCU will drive the tcu_mio_bs_moce_ctl as follows:

TESTMODE==1'b1 ==> TCU drives bs_mode_ctl to 1'b1

TESTMODE==1'b0 ==> bs_mode_ctl is under control of JTAG, so normalboundary scan can occur

This way, when we are in TESTMODE, all of the shared pins will bypass the boundary scan cells coming into the chip logic. This allows scan to operate correctly.When we are not in TESTMODE, bs_mode_ctl will normally be 1'b1 anyway and so the mux will be bypassed. Only if JTAG is programmed for a boundary scan test will bs_mode_ctl be 1'b0; TCU will block the effectssince TESTMODE==0, and PLL should block its shared pins with PLL_TESTMODE ==0.

Manufacturing/ATPG Scan :

All flops in MIO are on manufacturing scan chain and would support regular scan features like scanin,scanout,scandump,flush reset under control of TCU through the following signals from TCU :

```
tcu_aclk
tcu_bclk
    tcu_scan_en
    tcu_pce_ov
    scan_in
    scan_out
    tcu_mio_clk_stop
```

9.2 Debug Port

OpenSPARC T2 debug port width is defined by 166 signals for repeatability to complement Checkpoint /Replay . When not being used to monitor the repeatability signals (described in section 3.2.1.1) , the port would get used to monitor various other signals in OpenSPARC T2 in 5 different modes : SOC Observability,Tester charac/CPU debug , and core-soc debug.

These modes are programmable by SW by writing to the OpenSPARC T2 Debug Port Configuration register. In all the above 5 modes other than the NIU debug mode and PCI_EX debug modes, the debug port will be driven @ 2 x iol2clk frequency (2 x 350 mhz = 700 mhz nominal), with iol2clk being sent out on DBG_CK0 pin to the LA for sampling and aligning the data. In essence this is equivalent to data being driven on both edges of iol2clk . Commercially available LA's (like Tektroniks) do have the ability to support DDR signal sampling with the Tektroniks LA currently being able to support a max of 900 mhz DDR (both edges of 450 mhz clk). OpenSPARC T2 's debug port would employ double pumping CMOS signals @ 1.1 V and would not need to meet the timing and skew specs ascociated with traditional Memory multi-drop DDR2 interfaces. Also the Tektroniks LA probes would be connector less thereby reducing the load on the debug port drivers.

As mentioned before, the debug port pins would be shared with manufacturing scan test and membist signals so that with the debug ports disabled , some of these pins can be used for manufacturing scan and Membist of OpenSPARC T2 . The muxing of the debug port signals with the manufacturing scan test and membist signals would happen in the I/O cell itself in the mio.v block.

Upon chip reset , the debug port would come up disabled thereby saving power on the I/O's. The debug port can be enabled by writing to the Debug_en bit of the Debug Port Configuration Register (either by SW or by Jtag CREGs access) . The effect of the write would take place immediately and not after the next warm reset.

The muxing of the debug signals in OpenSPARC T2 on the debug port and also muxing of the debug port signals with the manufacturing scan test signals,membist signals and other miscellaneous signals is shown in the figure below .

The I/O's in OpenSPARC T2 debug port can be thus broadly classified as falling under 5 categories :

I/Os which are shared between debug port and DMO/membist signals that are outputs. For this group of signals , the Drive_en to the I/O's would get generated as :

assign dbg_mio_drv_en_muxbist_op = debug_en | tcu_dbg_jtag_membist_mode;

I/Os which are shared between debug port and Manufacturing Scan test signals that are outputs. For this group of signals, the Drive_en to the I/O's would get generated as follows :

```
assign dbg_mio_drv_en_muxtest_op = debug_en | mio_dbg_testmode;
```

I/O's which are shared between debug port and Manufacturing Scan test signals that are inputs. For this group of signals, the Drive_en to the I/O's would get generated as follows :

```
assign dbg_mio_drv_en_muxtest_inp = debug_en & ~mio_dbg_testmode;
```

I/O's which are shared between debug port and PLL test /char signals that are inputs. For this group of signals, the Drive_en to the I/O's would get generated as follows :

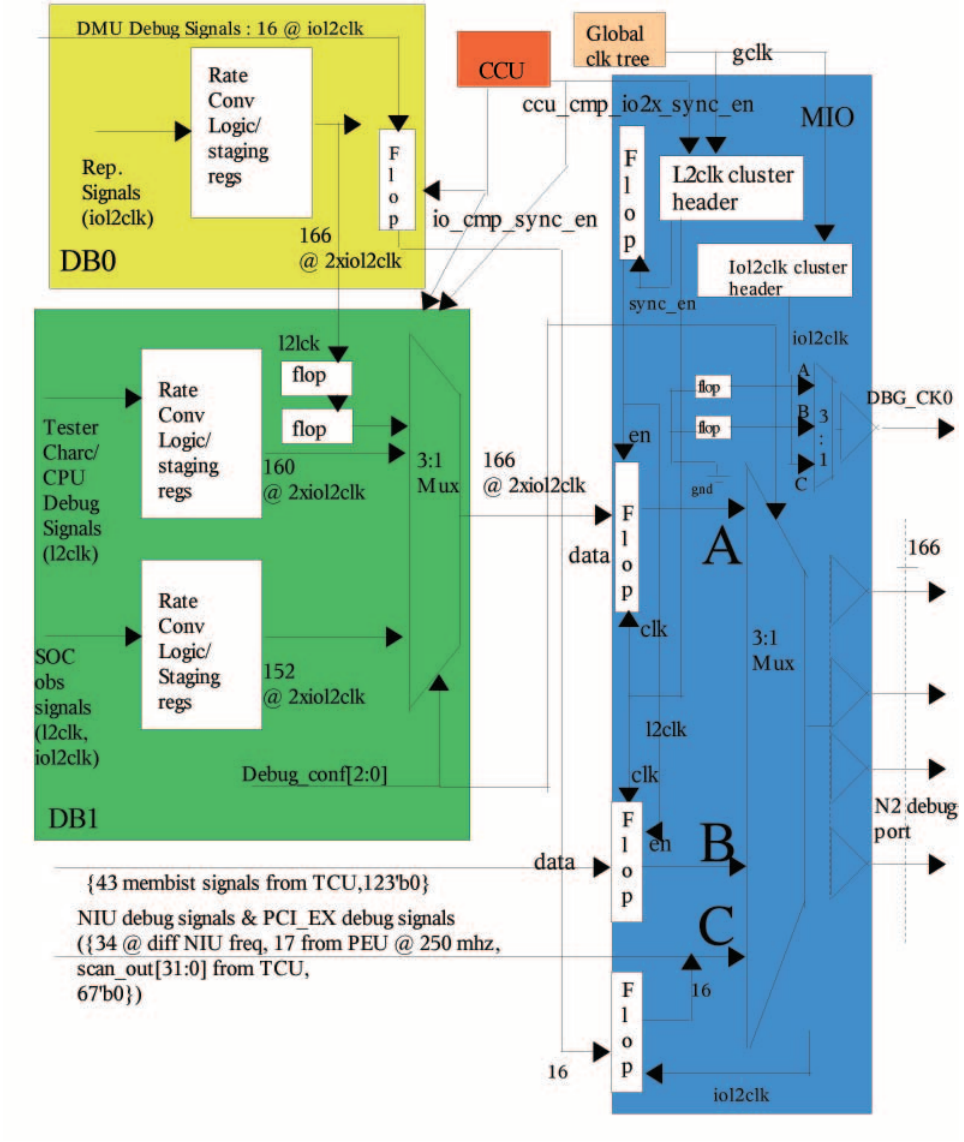
```
assign dbg_mio_drv_en_muxtestpll_inp = debug_en & ~mio_pll_testmode;
```

I/O's which are always driven as outputs in the debug mode . For this group of signals, the Drive_en to the I/O's would get generated as follows :

```
Assign dbg_mio_drv_en_op_only = debug_en.
```

Where “debug_en” is “Debug_En” bit in Debug Port Config register .

FIGURE 9-5 OpenSPARC T2 Debug Port Layout across DBG0,DBG1 and MIO



The pin sharing scheme in MIO is further explained in detail in section 5.2 of the document.

9.2.1 DTM Support in MIO

MIO I/O cells (n2_mio_cell_out_bscan,n2_mio_cell_bi_bscan, n2_mio_cell_bi_pd_bscan, n2_mio_cell_bi_pu_bscan) contains a 2:1 mux before the A flop to support DTM capability in OpenSPARC T2 . Under control of CCU, the ccu_mio_serdes_dtm signal would be asserted to configure MIO in two different DTM modes . Also CCU would be driving the cmp_io2x_sync_en to MIO with cmp_dr_sync_enable timing in these two modes.

DTM mode support on OpenSPARC T2 for MIO and DB1 modules is described in detail in the OpenSPARC T2 Debug Spec version 0.6 or later.

9.2.2 Timing Spec for Debug Port Signals for Reliable LA Sampling

For the Tektronix P6860 LA , the 166 pin debug port of OpenSPARC T2 would be connected to (166/16) = 11 pods (where each pod has 32 data connections and 2 clock connections). With the data being driven @ 700 mhz on both edges of a 350 mhz clock, the LA would be configured in a half channel mode with 11 pods providing a total of 332 memory locations storing data over every 350 mhz clock. 166 of these memory locations would be written on +ve edge of 350 mhz clock, and the other 166 on the negative edge of the clock on every cycle.

Minimum time for which data should be valid for (eye width) to be sampled reliably by the 8 Ghz internal clock of the LA is 625 psec (325 setup, 300 hold) which is a period of five 8 Ghz clocks (5 x 125 = 625 psec).

Data sampling window w.r.t 350 mhz external clk is pretty wide from -16 nsec to + 8.75 nsec. i.e signal to signal skew is 24.75 nsec max.At the beginning, the skew of each bit can be manually cancelled out before being displayed on the analyzer. This is the calibration process and would be typically done only once at the beginning on a bit by bot basis based on atraining sequenc being sent out on the debug port. The training sequence would be a repetitive pattern of 3 one's, followed by 1 zero : this assymetrical pattern would ease the alignment and deskewing of the data bits in the LA in case the skew for some bits is as large as one cycle .

Not e that once a calibration is done, the maximum cycle to cycle PVT skew that the LA can tolerate before it stops reliably sampling data across different PVT corners is measured as : clock period for data change rate – minimum eye width (625 psec). So for the 700 mhz data rate, the max PVT skew that the clock and data need to maintian through the chip,package and board is 1.4ns – 0.625 ns = 0.775 nsec. This jitter would cover PVT variations and bit to bit capacitive coupling effect related variations through the package and board. To reduce the PVT skew component within the chip , the 700 mhz debug signals would get retimed in the i/o cell (mio.v) as shown in Illustration 6.

9.3 MIO RTL Hierarchy

The MIO block (mio.v) would consist of the following design sub-blocks :

1 io cluster header (module name : clkgen_mio_io, instance name : mio_clk_header_iol2clk). This would generate iol2clk in MIO which would be fed as data input to I/O which drives the DBG_CK0 pin.

4 cmp cluster headers(module name : clkgen_mio_cmp, instance names : mio_clk_header_l2clk_0,mio_clk_header_l2clk_1,mio_clk_header_l2clk_2, mio_clk_header_l2clk_3) . Each cluster header provides the cmp_clk for a group of I/O's in MIO and staged version of ccu_cmp_io2x_sync_en from CCU to that I/O group. This sync_en(cmp_io2x_sync_en_out) gets further flopped in the mio_syncreg_ctl module to generate the final sync enable to the group of I/O's.

Sync Enable Staging Module (module name : mio_syncreg_ctl, instance names : io2xsyncen_reg0,io2xsyncen_reg1,io2xsyncen_reg2,io2xsyncen_reg3). This module contains a staging flop for the io2x sync enable generated from the corresponding cmp cluster header.

MIO glue logic (module name : mio_muxsel_ctl, instance name : muxsel). This is a very small module in MIO which would contain small amount of glue logic like invertors to generate mux selects to different MIO I/O groups . It would also contain retiming flops for the 16 DMU debug wires coming from db0 module.

Process Monitor Control Pins : PMI[1:0] and PMO. These do not have any drivers or receivers but are modelled as “assign” statements in rtl.

MIO I/O cells. There are 9 different flavors of I/O cells. Thee total number of instantiations of I/O cells equals 214. These 5 flavors of I/O cells are as follows :

Output Only (No Bscan) . Module name : n2_mio_cell_out

Pins driven : TDO,IMP_MON_PU,IMP_MON_PD

Input Only (No Bscan) . Module name : n2_mio_cell_in

Pins driven : TCK,PGRM_EN,BURNIN

Output Only (With Bscan) . Module name : n2_mio_cell_out_bscan

Pins Driven :
XAUI1_ACT_LED,XAUI1_LINK_LED,XAUI0_ACT_LED,XAUI0_LINK_LED,STCIQ, DBG_CK0,DBG_DQ[165:158],DBG_DQ[148:147],DBG_DQ[102:85], DBG_DQ[71:0],TRIGOUT,PEX_RESET_L, SSI_MOSI,SSI_SCK,FATAL_ERROR,XAUI_MDC, PLL_CHAR_OUT[1:0]

Input Only (with Bscan) . Module name : n2_mio_cell_in_bscan

Pins driven : DIVIDER_BYPASS, PLL_CMP_BYPASS,PLL_DR_BYPASS, TRIGIN, PB_RST_L, BUTTON_XIR_L, PWRON_RST_L, SSI_MISO,SSI_EXT_INT_L,VREG_SELBG_L,PWR_THRTTL_0[2:0],PWR_THRTTL_1[2:0]

Bidi (with Bscan). Module name : n2_mio_cell_bi_bscan

Pins Driven : DBG_DQ[139:103],DBG_DQ[84:72],XAUI_MDIO

Input Only (No Bscan) with Pullup . Module Name : n2_mio_cell_in_pu

Pins Driven : TDI,TMS,TRST_L

Input Only (with Bscan) with pulldown . Module name : n2_mio_cell_in_pd_bscan

Pins Driven : PLL_TESTMODE,TESTMODE,STCID,STCICFG[1:0],STCICLK,TESTCLKT,TESTCLKR

Bidi (with Bscan) with Pullup . Module Name : n2_mio_cell_bi_pu_bscan

Pins Driven : PLL_DIV2[5:0](shared with DBG_DQ[156:151]),PLL_TRST_L(shared with DBG_DQ[150])

Bidi (with Bscan) with pulldown . Module Name : n2_mio_cell_bi_pd_bscan

Pins Driven : PLL_CHAR_IN(shared with DBG_DQ[157]),PLL_CLAMP_FLTR(shared with DBG_DQ[149]),PLL_DIV4[6:0](shared with DBG_DQ[146:140])

XAUI_MDIO pin hookup is shown in the mio.sv rtl snippet below :

```
n2_mio_cell_bi_bscan cell_211 (
      .data_to_core        ( mdi ),
      .bs_scan_in          ( 1'b0 ),
      .bs_scan_out         (),
      .pad                 ( XAUI_MDIO ),
      .data_oe             ( mdoe ),
      .ain_mux_data        ( 1'b0 ),
      .bin_mux_data        ( 1'b0 ),
      .cin_mux_data        ( 1'b0 ),
      .ain_mux_sel         ( 1'b0 ),
      .bin_mux_sel         ( 1'b0 ),
```

CHAPTER 10

Debug

This chapter contains the following sections:

10.1 Overview

This document describes OpenSPARC T2 HW features for post silicon debugability which involves debugging any issues that interfere with early bringup as well as debugging the difficult, complex bugs that eluded pre-silicon verification, and are unexpected or unusual corner cases. The overall goal of implementing these features is to make silicon debug more efficient, shortening the time to root cause complex bugs and thereby reducing time to remove and replace.

10.1.1 Additional Relevant Documents

OpenSPARC T2 Programmer's Reference Manual Hardware Debug Chapter

OpenSPARC T2 Programmer's Reference Manual Error Handling Chapter

OpenSPARC T2 Programmer's Reference Manual Clocks,Reset,RED State Chapter

OpenSPARC T2 TCU Specification

10.2 OpenSPARC T2 Debug Features

10.2.1 Observability

10.2.1.1 CLK/PLL Observability

OpenSPARC T2 will provide clk/pll observability on pll_char_out[1:0] pins connected to pll_charc block in PLL. There will be two pairs of pll_char_out[1:0] pins coming out of OpenSPARC T2 : one for CMP PLL, and the other for MCU/DRAM PLL. In normal mode when the PLL's are not being characterized, these pins will be driven to 2'b0. The following tables show how the pll_char_out[1:0] pins will be driven for the respective PLL's.

TABLE 10-1 CMP PLL pll_char_out[1:0]

# of pll_char_in pulses = x	pll char decode	pll_char_out[1]	pll_char_out[0]
x< 64			
x mod 64 = 0	0xxx000	fvco/4 = 350MHz	pll_lock
x mod 64 = 1	0xxx001	fvco/4 = 350MHz	fvco/4 = 350MHz
x mod 64 = 2	0xxx010	raw_clk	fb_clk
x mod 64 = 3	0xxx011	fb_clk	raw_clk
x mod 64 = 4	0xxx100	ref	fb
x mod 64 = 5	0xxx101	fb	ref
x mod 64 = 6	0xxx110	up	dn
x mod 64 = 7	0xxx111	dn	up
x > or = 64			
64 - 95	10xxxxx	fl1clk/4 = 350MHz	fl1clk/4 = 350MHz
96 - 255	11xxxxx	fvco/4 = 350MHz	fvco/4 = 350MHz

TABLE 10-2 MCU/DRAM PLL pll_char_out[1:0]

# of pll_char_in pulses = x	pll char decode	pll_char_out[1]	pll_char_out[0]
x< 64			
x mod 64 = 0	0xxx000	Fvco/5 = 333 MHz	pll_lock
x mod 64 = 1	0xxx001	Fvco/5 = 333 MHz	Fvco/5 = 333 MHz
x mod 64 = 2	0xxx010	raw_clk	fb_clk
x mod 64 = 3	0xxx011	fb_clk	raw_clk
x mod 64 = 4	0xxx100	ref	fb
x mod 64 = 5	0xxx101	fb	ref
x mod 64 = 6	0xxx110	up	dn
x mod 64 = 7	0xxx111	dn	up
x > or = 64			
64 - 95	10xxxxx	Fl1clk/5 = 333 MHz	Fl1clk/5 = 333 MHz
96 - 255	11xxxxx	Fvco/5 = 333 MHz	Fvco/5 = 333 MHz

10.2.1.2 Debug Port

OpenSPARC T2 will have a 166 pins wide debug port which will be used as an observability vehicle to promote repeatability ,tester characterization, chip hang debug and general CPU and SOC debug. The debug port can be enabled through SW CSR access and Joint Test Action Group (JTAG) CREG access . The debug port can be configured into any one of 5 observability modes based on CSR bits (Dbg_conf[2:0]bits in OpenSPARC T2 Debug Port Configuration register : appendix 10.3.1) which are accessible by SW and also JTAG through CREG access . The following are the different observability modes of the debug port :

000 : SOC observability mode, OpenSPARC T2 Reset State (Reset State Machine Output), MCU, SII->L2,L2->SIO signals to help debug chip hangs (sent out on 159 pins)

001 : Tester charac/cpu debug mode ,{cpu_id,thread_id} on per L2 bank basis and cpu instruction commit status on per CPU basis, sent out on 160 pins

010 : Repeatability mode , SII and NCU inputs from DMU and NIU on debug port double pumped on 166 pins

011 : Core & SOC Debug , SII and NCU inputs from DMU and cpu instruction commit status on per CPU basis .

100 – 111 : Reserved for future use

These modes will be described in detail in the following sub-sections

Repeatability Mode

In this mode , a total of 353 signals (in iol2clk clk domain : cmpclk/4 or 350 MHz nominal) will be routed to debug.v (from NIU and DMU)From debug.v, 166 wires will get driven @ 700 MHz to the debug pins. These signals capture both inbound DMA and PIO returns from NIU and PCI_EX blocks in OpenSPARC T2 to SII and NCU and will be used as bus trace for checkpoint/replay scheme in OpenSPARC T2.These 353 signals and rate conversion to debug port frequency are shown below.

dmu_ncu_wrack_vld;

dmu_ncu_wrack_tag[3:0];

dmu_ncu_stall;

// total 6 bits @ 350 MHz = 3 pins @ 700 MHz (DDR)

dmu_ncu_vld;

dmu_ncu_data[31:0];

// 33 bits get driven over 4 clocks. 8 clocks minimum before next set of 4 clks

// so total of 132 bits to be emptied over 12 350 MHz clks, i.e. 66 bits DDR over

// 12 clocks , i.e. 6 pins @ 700 MHz (DDR)

niu_ncu_stall;

niu_ncu_vld;

niu_ncu_data[31:0];

// 34 bits @ 350 MHz == 17 pins @ 700 MHz (DDR)

dmu_sii_hdr_vld;

dmu_sii_reqbypass;

dmu_sii_datareq;

dmu_sii_datareq16;

dmu_sii_data [127:0];

dmu_sii_be[15:0];

// 148 bits @ 350 MHz = 74 pins @ 700 MHz (DDR)

niu_sii_hdr_vld;

niu_sii_reqbypass;

niu_sii_datareq;

niu_sii_data [127:0];

niu_sio_dq;

// 132 bits @ 350 MHz = 66 pins @ 700 MHz (DDR)

total = 66 + 74 + 17 + 3 + 6 = 166 pins @ 700 MHz (DDR)

Note – {dmu_ncu_vld,dmu_ncu_data[31:0]} take 5 iol2clk cycles to be seen on the debug port at the output of the chip from the time they are driven from dmu to ncu.

All other signals in the repeatability list take 3 iol2clk cycles to be seen on the debug port output at the output of the chip from the time they are driven to SII and NCU.

Tester Characterization / CPU Debug mode

The signals that will be observed on the debug port in this mode will be used for general CPU debug and tester characterization of multi-threaded diags and also for CPU speed binning on the tester. Each CPU will have 4 signals driven to debug.v and each L2 bank will have 6 signals driven to debug.v. All these signals will be at CMP clk frequency i.e. 1.4 GHz nominal. Since there are 8 cores and 8 L2 banks , this will lead to a total of (4+ 6) x 8 = 80 signals @ 1.4 GHz driven to debug.v. Since the debug port will drive the signals out @ 700 MHz, debug.v block will sample 2 consecutive cycles of these 80 bit wires and drive out 160 signals @ 700 MHz to the debug pins for LA sampling.

For each CPU, these 4 wires are chosen as follows :

There are 2 pipes / core and 2 thread groups per core. Since each core has 2 thread grps, we can have on 2 bits per thread grp/core : (i.e. total of 4 bits /core) :

00 : instruction non committed

01 : Control Transfer instruction committed in pipe

10 : Integer or FPU instruction committed in pipe

11 : Ld/Store instruction committed in pipe

i.e. we will see every instruction committed per cycle in each thread group . We don't want to know which thread that instruction belongs to though ..

For each L2 bank , the 6 wires are VCID[5:0] {CPU_ID[2:0],Thread_ID[2:0]} of each crossbar packet to that bank on every cycle.

The combination of these two groups of signals will be adequate to keep track of execution of instructions in both single and multi-threaded diags on the tester and also could be useful for CPU speed binning on the tester .

SOC Observability Mode

This mode will be used to capture a variety of critical SOC signals which will be helpful to debug chip hangs and also general debug of PCI_EX logic in OpenSPARC T2. The following is the breakup of the signals in this mode :

5 bit encoded state for Reset State Machine (has 20 states) from rst.sv to mio.sv to monitor reset state on the tester and LA . Sent out at sys_clk frequency from Reset block in OpenSPARC T2 (feedthrough in MIO) on 5 pins.

Each MCU will send the following NEW signals to debug.v which will be useful to debug MCU hangs/scheduler issues or MCU error handling issues on both FBDIMM channel errors and ECC errors.

mcu_dbg_rd_req_in_0 [3:0]	Read Request from L2 bank 0 to MCU (id + valid)
mcu_dbg_rd_req_in_1 [3:0]	Read Request from L2 bank 1 to MCU (id + valid)
mcu_dbg_rd_request_out[4 :0]	Read ack from MCU to L2 bank 0 or 1 (id + valid + dest_L2_bank)
mcu_dbg_wr_req_in_0	Write req valid from L2 bank 0
mcu_dbg_wr_req_in_1	Write req valid from L2 bank 1
mcu_dbg_wr_req_out[1:0]	0,1,2,3 Writes completed at DRAM indication (MCU dispatches up to a max of 3 writes on any cycle on 2 FBDIMM channels : then samples information coming FBDIMM channels to see if there were any errors , if no errors reported, MCU interprets as all writes completed)
mcu_dbg_mecc_err	MCU has detected an mecc error on a L2 read or scrub
mcu_dbg_secc_err	MCU has detected a secc error on a L2 read or scrub
mcu_dbg_fbd_err	MCU has detected a FBDIMM channel error
mcu_dbg_err_mode	FBDIMM interface logic has gone into error handling mode . This bit stays on until error handling complete.

These signals will all be synchronized by MCU to the iol2clk domain (350 MHz nominal) and sent to debug.v. This leads to a total of 21 wires / per MCU. Since there are 4 MCU's , this will lead to a total of 84 wires to debug.v from all MCU's together.

Debug.v will drive this information out on 84/2 = 42 pins of the debug port at 700 MHz.

SII and SIO will send the following signals to debug.v which will be useful to debug L2 hang cases (SII sent DMA request to L2, L2 never sends an ack or data return back) :

sii_dbg_l2t[0-7]_req[1:0] : Req type encoded on 2 bits from sii to each l2t bank

(00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)

l2t[0-7]_dbg_sii_iq_dequeue : L2 dequeue from IQ

l2t[0-7]_dbg_sii_wib_dequeue : L2 dequeue from IOWB

l2b[0-7]_dbg_sio_ctag_vld : response valid from L2 to SIO

l2b[0-7]_dbg_sio_ack_type : Read or Wr ack from L2 to SIO

l2b[0-7]_dbg_sio_ack_test : Ack to DMU or NIU

Which leads to a total of (7x8) = 56 wires for all L2 banks together @ 1.4 GHz to debug.v. debug.v will drive this information out on 56x2 = 112 pins of the debug port @ 700 MHz.

Thus total number of debug pins that will be used up in the SOC observability mode will be (42 + 112) = 154.

10.2.2 Repeatability

In order to effectively run processor tests in the post-silicon phase with or without the presence of I/O and debug them, we need to have a high level of repeatability within OpenSPARC T2's synchronous clock domains . These include the CPU clock domain (cmp clk domain:1.4 GHz nominal covers SPARCs, crossbar, L2's, portions of SII,SIO,NCU), the DRAM domain(266/333/400 MHz covers MCU logic before SerDes), and the I/O clock domain (350 MHz nominal covers rest of SII, SIO,NCU).

This will allow us to run a group of tests many times, with slightly different starting parameters (e.g. SPARC threads starting at slightly different times, or with different cache initialization) that shouldn't affect the outcome, looking for failing corner cases. When a failing case is found, the test and the particular seed parameters will be used to simulate the test in the pre-silicon environment, to see what caused the failure.

Not only that in case there are failures in some systems in the lab after days and weeks of system stress testing , this approach of recreating the failing condition in the chip RTL can reduce weeks of effort to root cause the problem which in past Sun chips has invariably resulted in pushout of RR schedules.

The overall approach involves very close interaction between some Debug Software (part of Hypervisor SW) and OpenSPARC T2 chip hardware. This is commonly known as checkpoint/replay mechanism where the debug software will periodically put the synchronous portion of the chip (as described before) into an idle state (idle all threads other than one, and also stall I/O into the synchronous domain) at what are called checkpoints. Once the synchronous portion of the chip is put into this idle state, the debug SW will dump all SW visible state of the machine to memory, and then initiate a "debug reset" of OpenSPARC T2 .

The debug software will initiate by writing a 1 to the DBR_GEN bit in RESET_GEN register. The RESET_GEN and RESET_SOURCE registers are shown in Appendix 10.3.2 and 10.3.3.

The Debug Reset is a flavor of Warm Reset in OpenSPARC T2 which is identical to the functionality of Warm Reset.

This Debug_Reset will put a majority of the synchronous domain of the OpenSPARC T2 chip into known state (all SW invisible state and some of the SW visible state also). So before invoking this reset, SW should dump the SW visible state that loses value over debug_reset to memory , and retrieve it back from memory after the reset.

Note – OpenSPARC T2, like previous Sun processors , keeps a fair amount of architected state unchanged for warm reset. Also contents of arrays (TLB's , L1/L2 caches etc) are unchanged . Please refer to Appendix 10.2 for a list of OpenSPARC T2 SW visible state that will be lost over Debug Reset and will need to be retrieved after debug_reset.

The duration of the debug reset is small enough (in the range of 40 microsecs), so that to address the data integrity in DRAM during the debug reset OpenSPARC T2 will either (1) address it through self refresh during the debug reset or (2) auto-refresh in small intervals before going to debug reset and then doing some in small intervals after coming out of debug reset. (this way we can compensate for missing about 6 or 7 refreshes over the 40 microsec).

Note – Self refresh will take additional time for link re-initialization which is :
200 mcu clks (end of self refresh -> dll lock on sdram) == 600 nsec at 333 MHz
+
12 microsecs (to re-initialize fbdimm channel : included in this is 100 nsec time for bitlock of serdes
After debug_reset, the reset vector will be fetched from memory from a different location (0x000000020) than a regular reset . This is because the boot code for a debug_reset will be different than a regular reset. The boot code will do several things at the beginning including program the Memory refresh registers, re-instate the SW visible state to the state before reset for those states that lose value over debug reset), remove the stall of inbound I/O to the synchronous domain of the chip from NIU and PCI_EX, before enabling all threads to start executing.

In normal operation POR and warm reset both trap to the RSTVaddr | 0x20 (0xFFFFFFFFF0000020) which maps to ROM. To enhance repeatibility, OpenSPARC T2 will have the capability of directing POR ,WMR or DBR to RAM. In order to POR or WMR or DBR from RAM at location ,(0x000000020), hyperprivileged software can set the ASI_WMR_VEC_MASK register.

The idea is that by capturing the SW visible state of OpenSPARC T2 (in the synchronous domain of the chip) on the last checkpoint prior to the failure and by initializing the synchronous portion of OpenSPARC T2 to known state , we can create a commong starting point between silicon and the synchronous portion of the chip rtl. Then by running the same code sequence on the sparcs from the last checkpoint to the failure point and capturing the I/O traffic to the SII,NCU inputs (synchronous I/O interface of OpenSPARC T2: debug port mode 000) from DMU ,NIU on the debug port lossless and feeding it back to the same nodes in the rtl, we can create the event sequence in rtl leading to the failure .

Note – For Checkpoint/Replay, we do not need to observe the FBDIMM interface on the debug port . This is because once the links are trained data will always come back to the MCU data return fifo in a fixed latency from the time of issue of the request. After training , MCU logic will record this latency (in terms of MCU clocks) in MCU Channel Read Latency Register (shown in Appendix 10.3.5). So the debug software can probe this value and feed that same latency to the equivalent point in the rtl and thereby achieve cycle accurateness w.r.t silicon without having to probe the fbdimm interface.

Thus this checkpoint/replay approach is intrusive on the state of the machine in the context of the tests,applications running on the chip in that it periodically halts all threads and I/O and takes the machine to reset state. This might change the timing of events to cause the bug to manifest itself later in time than usual, but eventually it will with millions of cycles of instructions executed in between checkpoints.And when it does , it can be recreated in rtl .

10.2.2.1 FBDIMM Link training after Debug Reset

Since debug reset will reset MCU , the FBDIMM links will have to be re-trained after reset deassertion and this will change the FBDIMM data round trip latency for subsequent requests till the next debug reset. Debug software can either live with this by reading the MCU Channel Read Latency Register after every debug reset or MCU needs to keep sending sync pulses during the debug reset.

To support the latter, MCU will keep a small amount of logic running during warm/debug reset while the rest of it gets reset through flush mechanism. This logic will comprise of (i)logic to keep the links enabled and generate sync pulses in a fixed repetitive manner under SW control and (ii) logic to keep incrementing the read pointers of the northbound MCU fifo's and 2 synchronizers per fifo (this way during debug/warm reset , the read and the write pointers constantly increment and are always offset by 2 : delay through the 2 synchronizers).All this logic will be physically implemented in a control block in MCU , whose clock tree will be synthesized and skew matched with the rest of MCU fed by the MCU clk grid.

Also TCU would send a separate stop signal to this block in MCU .This Stop would be asserted by TCU only during PWR_ON reset and during scandump. It will NOT be asserted during Warm /Debug Reset. Also all the flops in this block would be on the regular MCU scan chain but would be warm reset protected, so that during warm reset : (i) the functional clock would keep on running (as Stop is not asserted) (ii) while the flush happens , the flops in this block would not be affected as they are warm reset protected.The A clk going to this block for Scan would be the same as the aclk_wmr going to the rest of MCU. The B clock would be the same as the B clk going to the rest of MCU. Thus only during PWR_ON and scan dump would the flops in this block be flushed and scanned respectively.

Also MCU would support 2 new CSR bits for SW to control this feature . These 2 CSR bits are located in DRAM Debug Trigger Enable register . The 2 bits are as follows :

1. KP_LNK_UP.

   When written to 1'b1 :

   (i) Keeps the Southbound Links enabled during the duration of the Debug reset to send out the sync pulses.

   (ii) selects the output of the sync pulse gen logic in the new MCU control module to generate sync pulses.

   When written to 1'b0 :

   (i) selects the output of the regular sync pulse gen logic in MCU

   (ii) clears the counter for the regular sync pulse gen logic in MCU. (According to Aaron, the number of zeroes can be upto 42 (max) before he sends a 1 out. So he is going to start with a 1 itself when he switches over, which should be ok for the

AMB .He may be sending back to back 1's also , which is ok. Only requirement is to have two 1's not be separated by more than 42 zereos.As long as the diffference is less than 42 (even 0) , it should be ok)

(iii) takes MCU fbdimm interface state machine to L0 state, where it is ready to dispatch new read/write requests to the DIMMs.

2. MASK_ERR .

   When written to 1'b1 :

   (i) makes MCU mask all the errors it normally detects on LFSR mismatches on ALERT frame patterns coming in from AMB.

   Cleared by MCU Hardware 4K cycles after reset when the LFSR's are re-aligned by MCU.

Note – Both of KP_LNK_UP and MASK_ERR bits are protected on warm reset.

SW-HW interaction to achieve determinism on FMDIMM interface after debug reset:

1. After making sure no pending transactions in MCU , SW sets KP_LNK_UP and MASK_ERR right before initiating debug reset.
2. Debug reset happens . Whole of MCU gets reset other than the control logic module which has its clock running keeping the sync pulses going and the fifo read pointer incrementing every cycle
3. Debug Reset finishes. MCU fbdimm interface State machine comes up in DISABLED state. Sync acks keep coming but since MASK_ERR bit is set,no errors are flagged. MCU logic counts 4K cycles after reset and realigns the LFSR's and clears the MASK_ERR bit.
4. After a certain time T1 (but always fixed from the deassertion of debug reset) , SW writes a 0 to KP_LNK_UP bit. This clears the sync pulse gen counter, takes the FBDIMM interface state machine to L0 state , and selects the sync pulse gen counter output to generate the sync pulses.
5. After a time T2 from the point where SW wrote KP_LNK_UP with 0, the first fetch is issued on the southbound link. T2 should be the same all the time.

FBDIMM Interface behavior on Warm reset

The FBDIMM links would be re-trained after every warm reset. The behavior of MCU during and after warm reset is as follows :

1. Warm reset gets triggered due to PB_RST_L assertion or fatal error in OpenSPARC T2 or SW writing a CSR bit in Reset_Gen register. KP_LINK_UP and MASK_ERR = 1'b0 before OpenSPARC T2 goes into warm reset.

2. Warm reset happens. Since KP_LNK_UP = 0, the Southbound Links are shut down by MCU sometime during the duration of the warm reset. Clocks keep running to the control module in MCU while rest of MCU gets reset through flush.

3. Warm reset finishes. The MCU state machine comes up in Disabled state and SW puts it into link training state.

4. Link re-training happens.

10.2.2.2 I/O Quiescing in OpenSPARC T2 During Checkpoint

An inherent requirement for checkpoint/replay in OpenSPARC T2 is to stall I/O to the synchrnous domain of the chip (SII and NCU inputs) from NIU and PCI_EX blocks. This is part of the effort to get the chip to a quiescent state on every checkpoint before dumping SW visible state and asserting debug reset to get the synchronous portion of the chip to a known state.

This I/O quiescing will get implemented in OpenSPARC T2 under SW control by having debug.v module contain a couple of CSR bits (NIU_STALL and DMU_STALL) in OpenSPARC T2 I/O Quiesce Control Reg which SW can set to 1's by writing a 1 to them. Once these bits are set, debug.v will assert a couple of signals called dbg_niu_stall and dbg_dmu_stall to NIU and DMU respectively . On seeing the assertion of these two signals , NIU and DMU should suspend all transactions to SII and NCU at any convenient point (for NIU can be at a packet boundary , whatever is easy to implement and creates least corner cases) and send back niu_dbg_stall_ack and dmu_dbg_stall_ack to debug.v after they have received all pending acks and data returns back from SIU and NCU. At the point at which these two acks are sent to debug.v, the NIU->SII,NCU and DMU->SII,NCU interfaces will be considered as having quiesced. This applies to interrupts also. Neither DMU nor NIU should send any interrupt requests to NCU or SII after having sent the acks. On sampling “niu_dbg_stall_ack and dmu_dbg_stall_ack” signals, debug.v will set “NIU_STALL_DONE” and “DMU_STALL_DONE” bits in the 2 I/O Quiesce Control Reg. The debug software which will have been polling these status bits will then see that both bits are set and will proceed to dump SW visible state of machine to memory and then initiate a debug reset.

Note that even during the time this interface is quiesced , the Xaui and PCI_EX interface Serdes links are active and running.

After debug reset, the reset code will clear the NIU_STALL and DMU_STALL csr bits in debug.v which will cause debug.v to assert a couple of signals to NIU and DMU called dbg_niu_resume and dbg_dmu_resume. On receiving these “resume” signals, NIU and DMU will unquiesce their respective interfaces with SII and NCU and continue issuing transactions to SII and NCU.

10.2.3 Debug Events

OpenSPARC T2 will implement several debug events in SPARC Cores and SOC to aid debug. The purpose of these debug events is to comunicate with the external Logic Analyzer to start or stop taking LA traces on the debug port (in any one of the 4 modes) based on these events or to stop clocks in the chip and have the service processor initiate a full scan dump. These events are generally address matches in sparc and L2 (which are typically repeaatable by running the same code sequence) or occurrence of error events in different blocks in OpenSPARC T2.

10.2.3.1 Debug Events in SPARC Cores

Following are the different debug events in Core , under enable/disable control of SW :

Instruction Breakpoint Match (on a group of 4 threads basis : if hpstate.ibe =1 and a thread executes an instruction that matches the contents of any enabled fields of ASI_INST_MASK_REG)

Instruction VA Match (on a per thread basis , if ifetch VA matches against content of ASI_WATCHPOINT_REG , with “match on Instruction VA” enabled in ASI_LSU_CONTROL_REG)

Data Access VA Match (on a per thread basis , if data access VA matches against content of ASI_WATCHPOINT_REG , with “match on Data VA” enabled in ASI_LSU_CONTROL_REG).

Data access PA match (on a per thread basis , if data access PA matches against data PA watchpoint address stored in ASI_WATCHPOINT_REG , with “match on Data PA” enabled in ASI_LSU_CONTROL_REG).

Taken Control Transfer Instruction (if pstate.tct =1 and a control transfer instruction has been executed like conditional branch,jmps,retry,done)

Precise Error Event (recorded in I-SFSR or D-SFSR)

Disrupting Error Event (recorded in DESR)

Deferred Error Event (recorde in DFESR)

Performance Monitor Event (Counter wrap condition)

Each core will contain DECR register (Debug Event Control Register) which will give SW the ability (on a per CPU debug event basis) to do either one of the following :

Do nothing . Debug Event Disabled

Soft Stop, scan, resume (under control of TCU and CCU)

Hard stop , scan (under control of TCU and CCU)

Pulse TRIGOUT pin to trigger LA or Jtag Scan

Note – Soft-stop waits for OpenSPARC T2 core processor activity to quiesce, puts the processor or domain interfaces in an error-free but unresponsive state, then stops the clocks. Clocks turn off at the same cycle to all latches, flops and arrays within the stop domain. The quiescent conditions are domain-specific.

For the OpenSPARC T2 core, a typical sequence of activity is the following. The TCU activates a soft-stop request signal to the processor core. In response the processor stops executing instructions and waits for all activity to complete. Then it deactivates any non-TCU external core interfaces (such as the L2 interface). The processor then informs the TCU that it has achieved a soft-stop condition. The TCU then stops the processor's clocks. The main advantage of soft-stop over hard-stop is that it minimizes the likelihood that the system or chip hangs as a result of the processor terminating an in-flight command. So in case we want to resume execution in cores, soft stop should be used and not hard stop. TCU can initiate soft stop to cores on a per core basis (separate scan enables from TCU). No soft stop will be initiated to the SOC and L2 because we need to keep memory refresh running for DRAM and the PCI_EX and XAUI serdes links running.

However during the period when the clock is stopped after a soft stop, the core will be missing invalidations coming across the crossbar to it from the L2 cache due to memory operations initiated either by another core or I/O. This will result in the core losing coherence with memory so soft stop can be used with restart if there is no I/O activity in the system and the code running on the cores is partitioned in a way that there is no sharing across the L1's of different cores or if all cores are soft stopped in unison.

10.2.3.2 Debug Events in SOC

Similar to the SPARC cores. SOC portion of OpenSPARC T2 will have its own set of debug events and DECR register located in the Debug block (debug.v) in SOC. The following are the list of debug events in SOC :

- L2 PA Match in Bank 7
- L2 PA Match in Bank 6
- L2 PA Match in Bank 5
- L2 PA Match in Bank 4
- L2 PA Match in Bank 3
- L2 PA Match in Bank 2
- L2 PA Match in Bank 1

- L2 PA Match in Bank 0
- L2 Error (an error has occurred in any of 8 L2 banks)
- MCU Error (An error has occurred in any of 4 MCU's)
- SOC Error (An error has occurred in any of SII,SIO,DMU,PEU,NCU)

The Debug Block (debug.v) will contain the SOC_DECR register (SOC Debug Event Control Register) which will give SW the ability to configure the debug event to do either one of the following :

- Do nothing . Debug Event Disabled
- hard stop , scan (under control of TCU and CCU)
- Pulse TRIGOUT pin to trigger LA or JTAG Scan

Note – There will not be any soft stop initiated to the SOC and L2 because we need to keep memory refresh running for DRAM and the PCI_EX and XAUI SerDes links running : so cannot stop certain logic sections in MCU,PEU and NIU/MAC from running.

Note that each L2 bank will support a pair of registers to detect PA and VCID match. These two registers are called L2 Match Mask Register and L2 Compare Register and will be located within each L2 (l2t.sv) bank. The condition for asserting a debug event based on these two registers will be as follows :

If ((DATA & MASK == COMPARE) && Valid_data) then assert debug event.

For each of the error related debug events in MCU,L2,NCU,DMC there will be a similar DEBUG_TRIG_EN CSR bit located in those modules to cover the SOC errors . Each of those blocks will assert a wire to debug block (debug.v) when they encounter any error if the corresponding DEBUG_TRIG_EN bit is set to 1.

The debug block will accept those inputs and either initiate a hard stop request or issue a LA trigger request to TCU.

10.2.4 JTAG Access

OpenSPARC T2 provides several debug capabilities through its JTAG interface. It implements a JTAG block in its TCU (Test Control Unit) block which will be used to access not only standard JTAG services but also provides specific debug features. The JTAG architecture is designed to be compliant with IEEE 1149.1 standard. The system usage model of this JTAG access capability will be to have a Service Processor or external JTAG agent connected to OpenSPARC T2 , under whose control the following Debug Features will be possible in OpenSPARC T2 :

JTAG scan out : this can be done in system , but is destructive. The whole chip will be scanned out to get a scan dump. Very useful for debugging chip hang cases.

JTAG Shadow Scan : allows for inspection of specific registers while part is running in system , and is non-destructive. This feature is accessible through private JTAG instructions .

JTAG Boundary Scan : done in the system. Can either monitor I/O signals non-intrusively , or can over-write signals to test interconnects between components on the board. This feature is accessible through private JTAG instructions .

JTAG CREG/UCB : this allows for read or write of specific registers while part is running in system. Reads are non-destructive. This allows instructions to be sent to the NCU which then intermixes the transaction with normal requests from the CPU's. The NCU can then take the results and pass them back to the TCU which can then send the data serially out on TDO. This feature is accessible through private JTAG instructions but relies on the NCU to be working in the chip.

Clock Stretch : This feature is accessible through private JTAG instructions. A 32 bit counter called Reset Counter in TCU will be programmed through SW or JTAG with the required number of CMP clks in between the first and second warm reset . The counter will start counting down after the de-assertion of the second warm reset . When it reaches zero, TCU will initiate clock stretch. There will be a 2 bit DECR in TCU which SW/JTAG will program for Clock Stretch for this to happen. The programming of this register can happen around the same time that the Reset Counter is getting programmed (anytime between first and second warm reset). The DECR will support 4 encodings : Do Nothing,Hard Stop,Pulse Trigout and Clock Stretch.

Clock Stop : This feature is also accessible through a private JTAG instructions. With this feature the chip can be frozen (no clocks running) so the contents are left unchanged, viewable via scan. Two types of clock stop are supported : hard stop and soft stop. Soft stop is supported only for cores, while hard stop is supported both for the cores and OpenSPARC T2 as a whole.

Under control of JTAG, TCU can directly initiate Hard stop of OpenSPARC T2 after the Reset Counter has expired if TCU DECR was programmed for Hard Stop . Alternatively TCU can also directly request a hard stop if the TRIGIN pin is asserted in the system.

TCU can also be made to put individual cores in hard stop or soft stop mode through dedicated instructions from JTAG specifying hard or soft stop (TAP_CLOCK_HSTOP and TAP_CLOCK_SSTOP).

Hard and Soft Stop will be described in more detail later in the “Clock stop” section and also in “OpenSPARC T2 Core Debug Features “ section.

The purpose of hard stop is to stop as fast as possible, though due to di/dt concerns clocks in different clock domains will be stopped in a staggered fashion. After a hard stop the chip will probably need a reset before it can be started again. A hard stop can be used on the entire chip or for individual SPARC cores .

The second method of stop, called “soft stop”, is applicable only to SPARC cores; it will let the core(s) settle into a quiescent state before stopping clocks. This allows the cores to be scanned out non-destructively for examination then started up again from the point code execution left off. During the period when the core clock is stopped after a soft stop L1 cache invalidations from other cores or the I/O subsystem will be dropped, leading to a loss of data integrity unless all cores are soft stopped simultaneously and I/O operations are quiesced.

Single Stepping,Disable Overlap,Cycle Step, Run N Instructions : These are core specific execution sequences useful for debug and are available through JTAG interface for stand alone SPARCdebug. More details are in the OpenSPARC T2 Core Debug Features section.

10.2.4.1 JTAG Scan out

There are two types of scan, manufacturing scan and in system scan. Manufacturing scan is totally controlled by the pins, while the in system scan is done through the JTAG controller.

In system scan is done through JTAG instructions. The scan chains are configured into a single long chain and placed between TDI and TDO. The chains used to construct the long chain will be configurable.

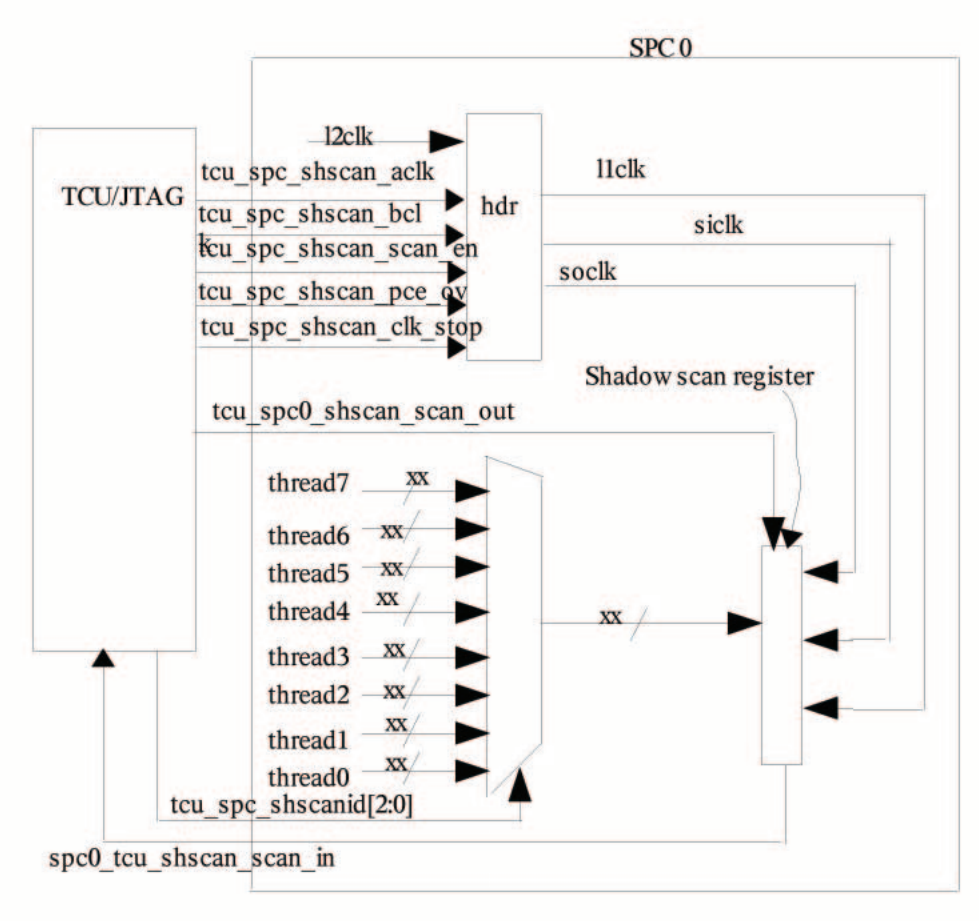
10.2.4.2 JTAG Shadow Scan

Shadow scan for the cores and L2 will be controlled via JTAG. The core shadow scan architecture is shown below; the header is a conceptual view of both the cluster and flop headers combined. Each core shadow scan will be contained in a separate scan chain, with its own clock headers and controls coming from the TCU. The following Core and L2 State flops have been identified for Shadow Scan for OpenSPARC T2 :

- PC[47:2] : 46 bits (OpenSPARC T2 does not implement VA[63:48])
- PSTATE & HPSTATE: 11 bits
- TL(Trap Level) : 3 bits
- TT (Trap Type) : 9 bits
- TPC (Trap PC)[47:2] : 46 bits
- TL_for_TT : 3 bits
- L2 Error Status register

- L2 FE/UE/CE Error Address Register
- L2 Notdata Error register

FIGURE 10-1 Core Shadow Scan Architecture



OpenSPARC T2 core will have TT, TPC, and a synchronized TL capture (TL_for_TT) to the core shadow scan with the following limitations:

TT, TPC, and TL_for_TT will update ONLY when a trap occurs. (The normal TL field will update for every change in the actual TL register.)

Software writes to TL and done/retry will NOT affect the shadow scan captured values of TT, TPC, and TL_for_TT. So, if the processor traps from TL==0 to TL==1 to TL==2 and then uses done and/or retry to get back to TL==0, shadow scan will

still reflect TT[2], TPC[2], and TL_for_TT will still be 2. Similarly, if the processor traps out to TL==2 and then software writes TL to 1 or 0, shadow scan will still show TT[2], TPC[2], and TL_for_TT will still be 2.

If multiple traps occur while the shadow scan is being scanned, the TT, TPC,and TL_for_TT updates due to all traps but the last trap will be lost.

The signals shscan_se, shscan_ce and shscan_stop are sourced from the TCK clock domain in JTAG; typically this requires synchronization using a megacell with metastability-hardened flops in a 2-flop sequence to achieve an acceptable MTBF. Assuming l1clk is stopped low, controlling bclk inactive before transitioning se will maintain the slave latch state that is captured with ce, although this allows the master latch to be exposed to metastability. This can be tolerated (since scanning with aclk will overwrite the master) so no special synchronization is required for se. The ce and stop signals will be passed through a synchronizer.

During a shadow-scan operation, the PLL is running and JTAG is used to capture the desired values into the shadow scan register. Then, JTAG turns on the stop signal into the header which drives the l1clk low, puts soclk inactive (high) and then transition se to the active state (high). The contents are then scanned-out via TDO. The core shadow scan can only be read, although any value may be scanned into it. Because TCK is specified to be at a much slower frequency than cpu_clk, the 2 cycles required for synchronization will not cause any overlapping.

All 8 core shadow scans are scanned serially as one chain, with core 0 closest to TDI and core 7 closest to TDO. Any core marked unavailable in the CMP core_available register will not be included when scanned via TDI to TDO. The shadow scan chain for a given core is placed in that cores second scan chain during ATPG test mode.

JTAG instructions to support Core Shadow Scan:

- TAP_SPCTHR0_SHSCAN Thread 0 contents for all available cores
- TAP_SPCTHR1_SHSCAN Thread 1 contents for all available cores
- TAP_SPCTHR2_SHSCAN Thread 2 contents for all available cores
- TAP_SPCTHR3_SHSCAN Thread 3 contents for all available cores
- TAP_SPCTHR4_SHSCAN Thread 4 contents for all available cores
- TAP_SPCTHR5_SHSCAN Thread 5 contents for all available cores
- TAP_SPCTHR6_SHSCAN Thread 6 contents for all available cores
- TAP_SPCTHR7_SHSCAN Thread 7 contents for all available cores

10.2.4.3 JTAG Boundary Scan

The boundary scan will allow through the use of JTAG instructions the testing of the I/O cells. The interface will provide the following instructions: Sample/Preload, Extest, HighZ, and Clamp. The boundary scan cells have also been designed such

that they will be included as part of the scan chain. Separate clock headers will be used for boundary scan cells in order to scan enable the flops without disturbing output of original flops.

Note – The BS_aclk is a pulse, width is to be determined, that is triggered by the rising edge of TCK. The BS_bclk is a pulse that is triggered by the negative edge of TCK.

10.2.4.4 JTAG CREG/UCB Access

The UCB interface is implemented inside the TCU and allows access via JTAG to IO mapped registers. A register's address and data in the case of writes are loaded via JTAG into holding registers in the TCU. The TCU then uses its UCB interface to communicate to the NCU which puts the new transaction (packet) into the data flow. The interface allows both reading and writing.

Note that in OpenSPARC T2 there is no way to access any SPARC CSR or L2 CSR through this NCU UCB interface. OpenSPARC T1 could access L2 CSR's and some SPARC CSRs by routing the packet through the crossbar to the lowest-numbered available SPARC physical core as specified by the CORE_AVAIL register, which then forwarded the packet to the L2 . This mechanism is not supported in OpenSPARC T2.

So with this NCU UCB protocol all the SOC CSRs (NCU,MCU,PCI_EX and NIU) are accessible from JTAG.

For a WRITE, a 40-bit address and 64 bits of data must be provided by JTAG to the UCB. For a READ, a 40-bit address is needed, with the data received from the NCU captured into a register in the TCU. To implement a READ, a sentinel bit is used since the exact timing of the read return is not deterministic. The system is only allowed to have 1 read outstanding at one time. There is no protection built in against this, adherence is left to the user. The buffer ID programmed through the JTAG data coming in needs to be set to 2'b01. This tells the NCU that the data is returned to the TCU.

For details on the JTAG CREG/UCB Access please refer to the OpenSPARC T2 TCU Specification.

Note – The UCB interfaces of NCU should not hang w.r.t any access to MCU,PCI_EX or NIU, i.e. JTAG CREG accesses should be able to make forward progress.

MCU will never hang on on-chip CSR accesses. Also, off chip PIO accesses MCU will send backs nacks on illegal addresses and also for channel errors. Even in error mode, MCU will wait to send out the off chip PIO's after recovering from the error. In case of a fatal error, it will send back a Nack to the PIO access and not wait for the error recovery. Architecturally there are no cases where MCU will not send back response to NCU for CSR/PIO accesses.

10.2.4.5 Clock Stretch

This feature is accessible through private JTAG instructions. A 32 bit counter called Reset Counter in TCU will be programmed through SW or JTAG with the required number of CMP clks in between the first and second warm reset . The counter will start counting down after the desertion of the second warm reset . When it reaches zero, TCU will initiate clock stretch. There will be a 2 bit DECR in TCU which SW/JTAG will program for Clock Stretch for this to happen. The programming of this register can happen around the same time that the Reset Counter is getting programmed (anytime between first and second warm reset). The TCU DECR will support 4 encodings : Do Nothing,Hard Stop,Pulse Trigout and Clock Stretch.

10.2.4.6 Clock Stop

Clock stop is the ability to stop the part after a given event. The part may or may not be in a state where it can continue operation. After the stop data can then be scanned out for debug. This allows the user to determine the state of the chip at meaningful times.

There are 2 types of clock stop a hard stop, and a soft stop. The purpose of the hard stop is to stop as fast as possible, but because of di/dt concerns this will mean that there will be some delay because the chip will stop in a staggered fashion. Because of the immediate stop the chip is now in a state that it cannot be restarted in system. It must be started from a reset again.

The 2nd method, soft stop, only applies to the cores and upon receiving a request the TCU will wait for the requesting core to settle into a quiescent state (via the core_running register) before stopping the clock to that core. This allows the core the possibility to start up again given the right system circumstances. Because of constraints such as keeping DRAM refresh running and XAUI and PCI_EX SerDes interfaces running on the chip , this is too hard to implement in OpenSPARC T2. Instead only the SPARC cores will have the ability for soft stop.

Soft stop should only be used on all cores at once if one wishes to start the cores after a soft stop.

Hard stop will be supported for both SPARC cores and OpenSPARC T2 chip as a whole.

To trigger a stop a debug register must first be set. Examples of these debug registers are instruction address breakpoint register, data address breakpoint register, architectural event(errors, performance register), and possibly others. These registers will then have fields that say what action should be taken if this event is enabled and occurs. The 2 different stops are 2 of the possible actions.

On OpenSPARC T2 the ability to stop clocks to various sections of the chip is provided via the TCU. Clocks can be stopped via JTAG directly or as a result of a debug event in SOC or Cores.

Under control of JTAG , TCU can directly initiate Hard stop of OpenSPARC T2 after Reset Counter has expired and the TCU DECR was programmed for Hard Stop . Alternatively , TCU can also request a direct hard stop if the TRIGIN pin was asserted in the system.

TCU can also be made to directly put individual cores in hard stop or soft stop mode through dedicated instructions from JTAG specifying hard or soft stop (TAP_CLOCK_HSTOP and TAP_CLOCK_SSTOP).

Clocks for the chip can be stopped either in parallel or serially across clock domains. After a clock stop, data can then be shifted out for debug via JTAG which allows the user to determine the state of the chip.

Serial and Parallel Clock Stop Modes

Stopping all clock domains in parallel may not be advisable due to excessive current fluctuations across the chip. Because of these di/dt concerns there is a serial clock stop mode where the clocks are stopped over several predefined clock domains with 128 CPU clock cycles between each clock stop activation. Stopping the clocks in such a staggered fashion with intervening delays is expected to lessen the di/dt concern. In the serial mode, via JTAG or software the user can update a clock domain register to specify which clock domain should be stopped first. Subsequent domains will then be stopped in a predetermined order, but the order is fixed.

During a parallel clock stop, the clocks will all be stopped at the same CPU clock cycle from the TCU. For both the serial and parallel clock stop methods, due to varying division ratios between the CPU and other clock domains, the actual CPU clock cycle at which a non CPU clock domain stops may vary between those domains, although it should be repeatable. To specify a parallel stop, all bits in the clock domain register should be set to 1, signifying they should all stop first. There is currently no provision for mixing serial and parallel clock stop modes across the clock domains.

Hard Stop

A hard clock stop request will result in the clocks being stopped without waiting for the chip to quiesce. The clocks may be stopped either in serial or parallel mode and will be stopped over all the chip

Soft Stop

A soft clock stop request will be handled as if it was a hard clock stop but will not be serviced until the domain requesting the soft clock stop is quiesced. The cores are the only domains that can request a soft clock stop, and only the clocks to the cores will be stopped by any soft stop request .

Data integrity will be lost after a soft stop unless all cores are stopped in unison.

Clock Stop Domains

Clock domains are partitioned so that control is achieved and that there is some commonality in the respective scan chains, and to minimize interactions because of the staggered stop. The sequence of stopping the clocks serially will always be the same given a specific start point and defaults to the order given in Table 4 . The user can program the starting point, but then the domains will stop in the predetermined order and wrap around until reaching the first domain stopped. For instance, stopping with spc7 first will result in spc6 being stopped last.

An 8-bit counter provides a delay of 128 CPU clock cycles between generation of successive clock stop signals from the TCU. This may be bypassed by setting all 18 bits in the clock domain register via JTAG, so that all clocks stop in parallel. The general structure of the clock stop control logic in the TCU is shown in FIGURE 10-2.

FIGURE 10-2 .TCU Clock Stop Logic

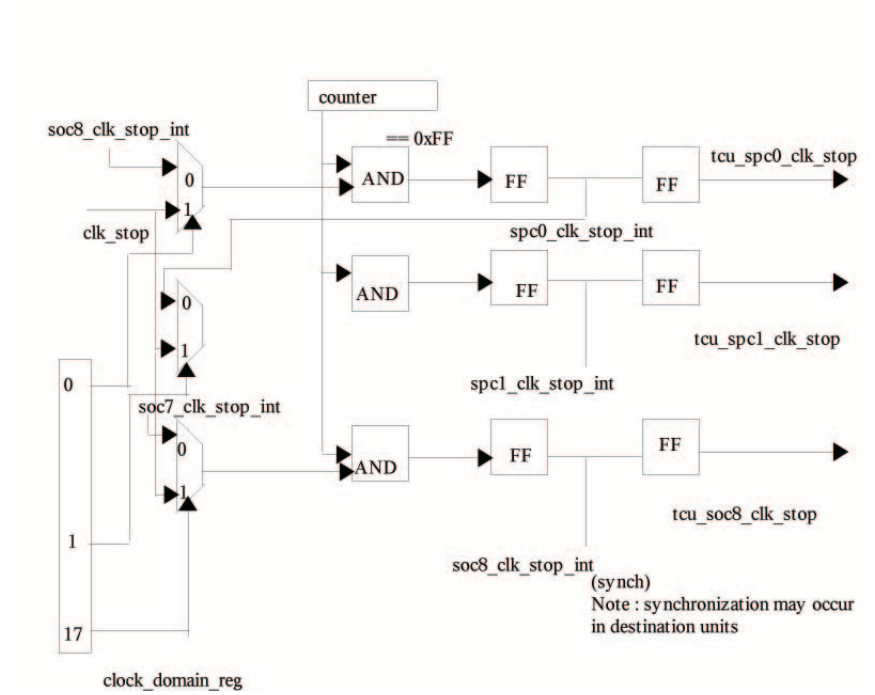
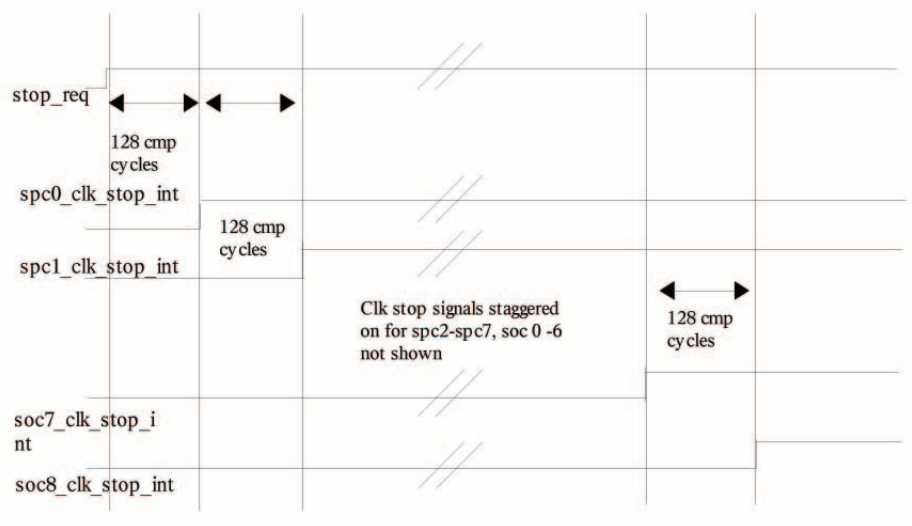


FIGURE 10-3 Clock Stop Sequencing through Clock Domains



All clock stop control logic in the TCU is in CMP clock domain. At this time the non-CMP clock domain stop signals are synchronized before leaving TCU. If this synchronization moves into the respective units, the outputs from TCU will be relative to the global clock grid. Clocks are restarted by turning off clk_stop signals. When started serially, the 128 CMP cycle delay is used again to reduce di/dt concerns.

10.2.4.7 Single Stepping,Disable Overlap,Cycle Step, Run N Instructions :

These are core specific execution sequences useful for debug and are available through JTAG interface for stand alone SPARC debug. More details are in the OpenSPARC T2 Core Debug Features section.

10.2.5 Fatal Error Indication on Pin

OpenSPARC T2 has a FATAL_ERROR pin that will get asserted when any of OpenSPARC T2 logic blocks encounter a Fatal Error. This will notify the Service Processor about OpenSPARC T2's error state. On a fatal error, OpenSPARC T2 asserts Warm Reset and also asserts PCI_EXPRESS_RESET_L pin to reset the external PCI-Express devices. The sources of Fatal Error in OpenSPARC T2 are L2 cache (each L2 bank can detect its own VUAD Uncorrectable ECC and Directory

Parity fatal errors) and NCU (SOC errors in blocks like SII,SIO,DMU,NIU,MCU which can be turned fatal by SW enabling fatal_error reporting for them in SOC Fatal Error Enable Register at location 0x80-0000-0018).

10.2.6 TRIGIN and TRIGOUT pins

TRIGOUT and TRIGIN pins will be asserted and sampled by TCU.

TRIGIN when asserted from the system will require TCU to do a hard stop of OpenSPARC T2 after it cycle counter expires , followed by a scan dump.

TRIGOUT will be asserted by TCU also after the cycle counter in TCU expires under any of the following conditions :

TCU gets a Pulse Trigger Pin request from any of the cores or the debug block based on some debug event having happened either in any of the cores or any SOC block.

TCU DECR has been programmed for Pulse Trigger and the Reset Counter has expired. (In this case first the reset counter will expire , then the cycle counter will count down to zero and then TRIGOUT will be asserted)

Debug SW (as part of Checkpoint/Replay) chooses to pulse the TRIGOUT pin after taking a checkpoint to start taking LA traces from OpenSPARC T2's debug port. To support this one , TCU will have a CSR bit that SW can write to pulse TRIGOUT.

10.2.7 DTM Support in DB1,MIO modules

DB1 and MIO modules would contain logic to support DTM capability in OpenSPARC T2. Under control of CCU, the ccu_dbg1_serdes_dtm and ccu_mio_serdes_dtm signals would be asserted to configure DB1 and MIO in two different DTM modes.

CCU is has a pair of CSR bits (serdes_dtm1, serdes_dtm2 in PLL_CTL reg) which will control DTM mode 1 and 2 respectively as follows:

TABLE 10-3 OpenSPARC T2 DTM Modes

Serdes_DTM 1	Serdes_DTM2	ccu_mio_ serdes_dtm	ccu_dbg1_ serdes_dtm	Description/Comments
0	0	0	0	Normal Mode (DTM off)
1	0	1	1	DTM mode 1 (MCU ECC and PEU TX info sent out at dr_clk frequency of ~100 mhz on debug port). In MIO, the data is clocked at cmp_clk with dr sync enables. Debug port to be configured in any mode **other** than modes : 3'b000,3'b100,3'b101 On debug port : 87:0 : MCU CRC data 93:88 : Dont care 165:94 : PEU TX Data ccu_serdes_dtm asserted to all other blocks in OpenSPARC T2 by CCU.
0	1	0	0	DTM mode 2 (Debug port to be configured in **any** of the 6 debug modes. Debug data sent out at cmp_clk with dr sync enables in all modes other than NIU debug mode and PEU debug signals in PEU debug modes). So there would be data loss but should be repeatable. On debug port : Debug signals for the debug mode chosen. ccu_serdes_dtm asserted to all other blocks in OpenSPARC T2 by CCU.
1	1	1	1	Invalid Programming by SW. HW treats it as DTM mode 1.

FIGURE 10-4 and FIGURE 10-5 show the paths through DB1 and MIO modules to get the DTM mode signals out of the chip in DTM modes 1 and 2 respectively

FIGURE 10-4 DTM Mode 1 Configuration for db1,mio in OpenSPARC T2

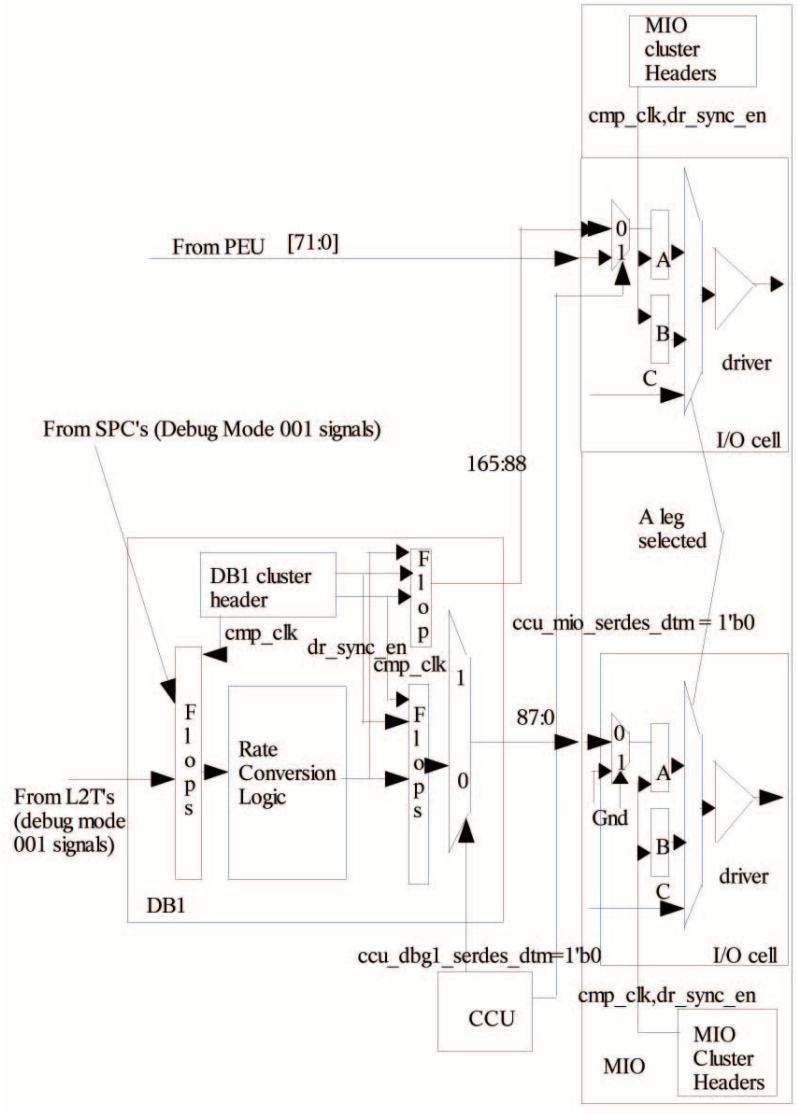
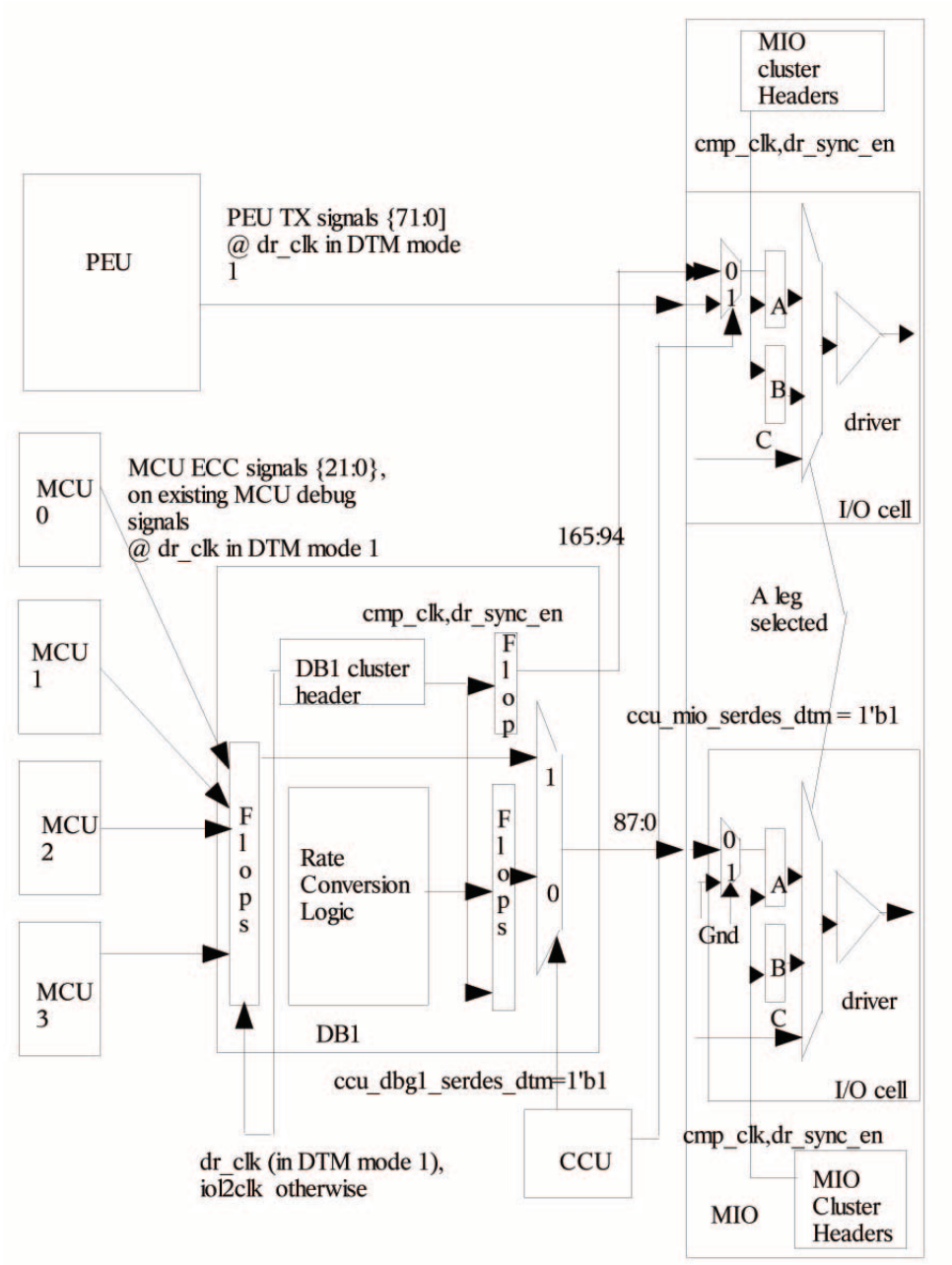


FIGURE 10-5 DTM Mode 2 Configuration for db1,mio in OpenSPARC T2



Note – For DTM mode 2 to be used effectively on the tester, the relationship between cmp_clk and dr_sync_en from CCU has to be the same all the time after every reset. This has to be guaranteed by CCU.

The DTM mode control CSR bits, the timing diagram of dr_sync_en w.r.t cmp_clk and also the mechanism of OpenSPARC T2 entering DTM modes would all be covered in the CCU MAS.

10.2.7.1 MCU DTM Mode Signals

The MCU's DTM debug information is 22 bits of CRC information from the southbound (transmit) link. The MCU communicates with the AMBs with 120-bit frames. Each frame consists of two sections, a 26-bit commandA section with 14-bit CRC and a 72-bit commandBC/data section with 22-bit CRC. 14 bits of the commandBC/data CRC is XORed with the 14-bit commandA CRC from the preceding frame to reduce the total number of CRC bits to 22.

Each MCU has 2 southbound FBD channels, and each channel has 22 bits of CRC per frame. The 22 bits from each channel are bitwise XORed to provide 22 bits total to the debug port.

For details on the CRC algorithms, please refer to the FBDIMM Arch and Protocol spec., section 5.4.

On debug port :

DBG_DQ[87:0] = {MCU3_CRC[21],MCU0_CRC[20:0],

MCU2_CRC[21],MCU1_CRC[20:0],

MCU1_CRC[21],MCU2_CRC[20:0],

MCU0_CRC[21],MCU3_CRC[20:0]}

10.3 OpenSPARC T2 Core Debug Features

This section describes the OpenSPARC T2 core debug features.

From a system and business perspective, the goal of these features is to minimize time-to-revenue by providing means to speed silicon and system bringup and debug. If a failure occurs at a customer site, these features can be used to capture

and analyze the failure, so that customer downtime is minimized. From a chip and core perspective, these features are simple, general, and powerful enough to enable debug both in stand-alone test fixtures as well as in-situ systems.

The OpenSPARC T2 core is a full-scan design: every latch (in arrays) and register bit is concatenated into a scan string. The core has 3 scan string inputs and outputs. The length of each scan string is limited, but the scan strings can be connected by the Test Control Unit (TCU) to form one long scan chain for each physical core.

The TCU can flush the scan strings by holding the scan clocks active and forcing a '0' at the input of each string.

It can also serially scan data into or out of the long scan chain.

Data can be scanned out of the long scan chain non-destructively by logically wrapping the scan chain output to the scan chain input: once the scan-out has been completed, all latch and flop bits contain their pre-scan values.

Data can also be scanned into the long scan chain with any arbitrary subset of the bits being altered with respect to their pre-scan value.

TCU will provide scan chain control on a per core basis (3 scan chains per core) for non-destructive scan out after a core soft stop without disturbing other cores or the rest of OpenSPARC T2. So whatever can be done on a long scan chain from the TCU can be done on a per core basis over 3 scan chains also.

The flush and scan operations can be controlled externally to the OpenSPARC T2 chip via commands sent to the TCU's JTAG interface. Scan string data can be observed or loaded via the JTAG interface. This allows an external agent (such as a PC or workstation with a JTAG interface card) to observe and change any storage location in the chip. By using several sequences of scan in and scan out commands and appropriate clock control, any on-chip memory location (flops through scan and arrays through Macrotest) can be read out non-destructively or changed.

Full-scan and flush capabilities gives OpenSPARC T2 core a solid foundation to support more sophisticated debug features. These features complement and augment traditional Sun debug features and do not preclude their use. For example, OpenSPARC T2 core includes debug features such as instruction watchpoint virtual address, data watchpoint virtual and physical address, instruction breakpoint, and software traps on hardware-detected error conditions.

The features described in this chapter are more useful for hardware or system designers debugging possible chip functional or circuit failures as opposed to software designers debugging code errors.

10.3.1 Basic Features

The TCU has a JTAG interface. The TCU can be controlled via this interface. In particular, an external agent connected to the JTAG port can issue commands to the TCU and the TCU provides data in response.

The following basic features either currently exist (are defined architecturally), or come as a side-effect of having full-scan :

1. 1. Existing architecturally visible debug capabilities. Existing means specified by V9/JPS1 or other SPARC processors, such as Millenium or OpenSPARC T1. These include instruction breakpoints, instruction watchpoint virtual address, data watchpoint virtual and physical address, trap-on-taken-control transfer, and trap on hardware-detected errors. These debug features are visible to and primarily used by software and can be invoked by programs running on the chip. In OpenSPARC T2, these debug features can be activated via scan also. However, instead of causing a trap, one of these debug events either stops the clocks (soft or hard stop, see below) or pulses an external pin. These features are described in section below.

   Since these features are shared with S/W, using them via scan may conflict with programs running on the core.

2. Start and stop clocks to the core. Stopping OpenSPARC T2 core clocks is performed via the TCU clock enable function in the core clock network.

3. Configure scan chains for non-destructive scan-out and scan-out those chains. The data scanned out appears on the JTAG interface. When performing a non-destructive scan, logic which may be affected by random scan values must be conditioned. For example, the TCU gates off write-enable lines to non-scannable arrays to prevent data corruption. If it is useful to scan a processor core independent of other cores and the L2 interface, then that processor's interfaces must be conditioned not to create phantom interface transactions while the interface registers are being scanned.

4. Configure scan chains for scan-in. In conjunction with scan-out, this can be used as a read-modify-write operation to update machine state. The data to be scanned in is presented over the JTAG interface as part of the scan-in command. In the past this approach has been used for speed path analysis on silicon or to validate logic bug fixes in silicon even before change is made in RTL.

5. Ability to read and write any non-scannable array location in the OpenSPARC T2 core. This capability is provided as a macro of scan-out and scan-in commands issued over the JTAG interface. The TCU translates these commands to sequences of scan operations and core functional clock cycles to read or write OpenSPARC T2 core array contents.

   Ability to configure various debug features via JTAG scan or direct commands. These features and commands are the subject of the next section.

Shadow scan facility: The shadow scan facility allows an external agent to query a subset of the state of the chip without requiring the chip clocks or domain clocks to be stopped. Due to hardware cost only a small fraction of the on-chip registers have shadow scan capability.

10.3.2 Enhanced Features

The following is a list of enhanced debug features. Each of these features is only available via the JTAG interface. These features are not visible to or configurable by software running on a core.

Hard-stop: Hard-stop is used as a noun and an adjective. As an adjective it describes the stop type; namely the clocks are stopped immediately without regard to any chip or system activity. The core comes to an immediate, synchronized stop (all latches/flops/arrays see the clock stopped at the same cycle) when clocks are shut off. As a noun hard stop is a command to immediately stop functional clocks to the entire chip or perhaps an individual clock domain. It is issued as a command over the JTAG interface to the TCU, which in turn disables the functional clocks. It is used as a prelude to other commands, such as a scan-out. In general, hard-stop is not recoverable. In particular, if a processor's clocks are stopped during an external bus cycle or memory cycle, it won't respond to further requests. This can cause other system errors. Usually hard-stop is used in a stand-alone debug environment or as a last resort due to the core/chip not responding to a soft-stop. TCU will implement hard stop to stop clocks to all blocks of OpenSPARC T2 (including SOC blocks).

Soft-stop: Soft-stop is used as a noun and an adjective. As a noun it refers to a soft-stop command issued via JTAG to the TCU. As an adjective it describes a more graceful stop than hard stop.

For the OpenSPARC T2 core a typical sequence of activity is the following. The TCU activates a soft-stop request signal to the processor core. In response the processor stops executing instructions and waits for all activity to complete. Then it deactivates any non-TCU external core interfaces (such as the L2 interface). The processor then informs the TCU that it has achieved a soft-stop condition. The TCU then stops the processor's clocks.

Soft stop can also be initiated via soft stop requests from the core due to certain events occurring. Soft-stop waits for OpenSPARC T2 core processor activity to quiesce, puts the processor or domain interfaces in an error-free but unresponsive state, then stops the clocks. Clocks turn off at the same cycle to all latches, flops and arrays within the stop domain. The quiescent conditions are domain-specific.

Data integrity will be lost unless all cores are stopped in unison.

TCU will initiate soft stop only to cores on a per core basis (separate scan enables from TCU) . There will not be any soft stop initiated to the SOC and L2 because we need to keep memory refresh running for DRAM and the PCI_EX and XAUI SerDes links running : so cannot stop certain logic sections in MCU,PEU and NIU/MAC from running.

Stop-clocks on event: This feature allows triggers to be set up so that if one of them is activated, the TCU stops the clocks to the core in as few cycles as possible. Each processor core has an event list consisting of the overflow of a performance monitoring counter, any core-specific error reflected in setting an ESR bit, or other events. The event(s) to be enabled for stopping can be specified by setting a bit in a core-specific control register during a scan-in operation. Each bit in the control register is associated with a particular event. Multiple bits may be set at once. Each cycle the contents of the control register are ANDed with the corresponding events in that domain and the output is ORed together and fed to the TCU. The TCU collects domain stop signal outputs. If any stop signal is asserted, it stops the corresponding domain's clocks.

Cycle counter: This feature tunes when clocks are stopped/stretched or TRIGOUT pin pulsed relative an event occurring. A decrementing counter is used in conjunction with :

1. JTAG initiated hard-stop/ soft-stop,pulse TRIGOUT,clock-stretch request

2. TRIGIN initiated hard stop request and

3. a debug event based hard-stop,soft stop,pulse TRIGOUT request .

Once the debug event trigger or JTAG stop command or TRIGIN pin has been activated, the counter starts decrementing and debug action is initiated when the counter reaches 0.

W.r.t. clock stops, by programming this counter (32 bits wide) and knowing the minimum round trip delay from core to TCU to core and then issuing a hard-stop or soft-stop command through JTAG, one has control over when the clocks are stopped. This is useful since once the clock-stop order has been received by the TCU, it takes several cycles to stop the clocks to the domain. The counter allows an earlier triggering event to be specified, delaying the clock stop to line up with the later (desired) event. The counter enables fine-grain control to isolate a failing cycle. The counter is located in the TCU.

Debug Event Counter : In addition to the cycle counter , TCU will support a 32 bit event counter at address ahead of it, which is also SW/JTAG programmable to count a specific debug event (decrement on every occurrence of the event) . When this event counter decrements to zero, TCU will start decrementing the cycle counter , and when the cycle counter decrements to zero TCU will take the debug action (soft stop,hard stop or TRIGOUT assertion). TCU will just OR the event sources and use the result of the OR to decrement the event counter. when using this event counter, SW/JTAG will make sure that only one debug event is enabled so that the event

counter will decrement for only one event. The debug event counter works only in conjunction with SOC and Core Debug Events , and NOT with TRIGIN pin or JTAG initiated Debug actions associated with Reset Counter in TCU.

Usage model :

Assume only one event is enabled for debug as instruction breakpoint match in 1 of 8 cores, all other debug events are disabled in core and SOC DECR's. Assume the debug action programmed for instruction breakpoint match is hard stop. So depending on the value programmed in the event counter, TCU will keep sampling hard stop requests from that core which has the instruction breakpoint debug event enabled and keep decrementing the event counter every time it gets a hard stop request.

When the counter decrements to zero, TCU starts decrementing the cycle counter and when that decrements to zero, TCU asserts hard stop and shuts off the clock to that core.

Hard stop and pulse TRIGOUT are the two debug actions with which this event counter can be used. It wont work for soft stop request originating from the core as the core quiesces before asserting soft stop request.

So SW/JTAG will make sure of the following when using this event counter (normally it will always be programmed to zero, so that the TCU will simply ignore it):

1. only one debug event enabled in OpenSPARC T2
2. debug action for that event programmed as hard stop or pulse TRIGOUT
3. TCU will not detect these two conditions, this is a requirement from SW/JTAG programmer.

External pulse on triggered event: TRIGOUT pin asserts on the occurrence of a configured event. Like the stop-clocks on event, an event (or set of events) may be scanned in to an event control register. If the event occurs and this feature is configured, an external, dedicated pin (TRIGOUT) is pulsed when the event occurs. This pin is pulsed at some low frequency generated off of the core clk: can pulse at core clk frequency. It gives an external indication that the event has occurred and allows external logic to sync up or start capturing bus cycles for further debug or analysis.

Single instruction step: This feature allows the debug agent to execute one processor instruction among the available, enabled, and running threads, then report quiesce state. Each physical OpenSPARC T2 core can be configured by JTAG to have a single-instruction step feature through hyperprivileged ASI_OVERLAP_MODE reg located at ASI 45, VA 0x10 . Typically this feature is used by the user issuing a single-instruction step command via the JTAG interface. This feature allows

designers to debug possible instruction execution problems by checking that the results of an instruction's execution match expected values (by non-destructive scan out and comparing with expected values in simulation). In conjunction with external frequency, temperature and voltage control, it can provide some evidence or information to help figure out what the critical path is.

Run N instructions: This is a sequence of single instruction step commands, and will be controlled by a sequence of JTAG single instruction step commands issued to TCU from the service processor. The usage model is specified in section 5.3.6.

Disable overlap mode: This feature causes each of the available,enabled, and running threads to execute one instruction and quiesce all activity before fetching the next instruction (essentially pipelining is disabled).

Cycle step: This feature allows one to sequence the domain pipeline N cycles at a time, where N can be 1 to the value of the cycle counter in TCU described above. With N set to 1, it is typically used to non-destructively scan out the domain pipeline for loading into a logic simulator and comparing the simulator values with the hardware values. It can also be used to check for critical paths. The feature is controlled by the TCU, which enables domain clocks for N cycles.

The usage model for cycle step is as follows :

1. User writes to a counter in TCU through JTAG interface, N number of cycles which TCU will use to cycle step core(s).
2. User reads back counter making sure counter has been written correctly.
3. User issues a TAP_CLOCK_HSTOP to hard stop the core(s) that need to be cycle stepped . TCU will stop clocks low for selected cores.
4. User issues a JTAG Command to do Cycle step .
5. TCU turns the clock on for the selected core(s) and starts decrementing counter.
6. TCU counter reaches zero.
7. TCU sets a bit in a register indicating Cycle Stepping done and stops the clocks to the cycle stepped cores.
8. User reads this register and sees Cycle Stepping done.
9. User issues a TAP_Serial_scan instruction to the core(s) that were cycle stepped to serially scan out the contents of the core non-destructively (by using the scan loopback scheme)
10. If user wants to continue execution on the cycle stepped cores, it will issue a TAP_CLOCK_START command to the core(s) that were cycle stepped.
11. TCU turns clocks on to the cores that were cycle stepped.
12. Cores resume execution

Note that after hard stop of cores and cycle steps, restarting without a reset will not produce correct behavior as during clock stop period , the core will be missing all responses from the crossbar on prior accesses . So the use of cycle step is to mainly to do some very focused logic debug and critical timing path analysis without any ability to restart.

All the modes of operation defined in the ASI_OVERLAP_MODE reg have been implemented in OpenSPARC T2 core.

10.3.3 Details of the OpenSPARC T2 Core Debug Features

This section details the OpenSPARC T2 core debug features.

10.3.3.1 Instruction Breakpoints

Like OpenSPARC T1 the OpenSPARC T2 core provides an instruction breakpoint capability. Each thread group has a hyperprivileged, read-write ASI_INST_MASK_REG at ASI 0x42, VA 0x8. Threads 0, 1, 2, and 3 share one register, and threads 4, 5, 6, and 7 share another register. The contents of this register are described below. All bits are initialized to 0 at POR. Reserved bits read as zeroes and are ignored on writes.

TABLE 10-4 ASI_INST_MASK_REG Contents

Bit Index	Register Field Name	Description
63:39	-	Reserved
38	ENB31_30	Enable matching on bits 31:30 of the instruction
37	ENB29_25	Enable matching on bits 29:25 of the instruction
36	ENB24_19	Enable matching on bits 24:19 of the instruction
35	ENB18_14	Enable matching on bits 18:14 of the instruction
34	ENB13	Enable matching on bit 13 of the instruction
33	ENB12_5	Enable matching on bits 12:5 of the instruction
32	ENB4_0	Enable matching on bits 4:0 of the instruction
31:0	Instr	The instruction pattern to match (opcode, reg address : whole instruction)

If HPSTATE.IBE is set to '1', instruction breakpoints are enabled. If a thread executes an instruction which matches the contents of all enabled fields in the INST field of the ASI_INST_MASK_REG, the thread takes an Instruction_Breakpoint trap. Non-privileged accesses to this register cause a Privileged_Action trap; supervisor accesses cause a Data_Access_Exception trap.

Additionally, if Core DECR is configured for ASI_VA_BREAKPOINT events , the OpenSPARC T2 core will take a debug action as configured by that register.

Since this register is shared between software and scan, debug agents should take care to ensure that only one agent (software : Software debugger/Emulator or hardware : Service Processor) is configured to use this facility at a time.

10.3.3.2 Instruction and Data Address Watchpoints

Each thread has a hyperprivileged ASI_WATCHPOINT register located at ASI 0x58, VA 0x38 which controls address watchpoint traps. The OpenSPARC T2 core can take a Instruction_VA_Watchpoint trap when this register is configured for an instruction fetch whose fetch address matches. The OpenSPARC T2 core takes a VA_watchpoint trap when this register is configured for a data access, and the core executes a memory reference instruction whose memory reference virtual address matches. Each thread can be configured to match only on instruction virtual addresses or data virtual address at one time.

Additionally, a physical address watchpoint for data accesses is implemented, the data PA watchpoint address will be stored in ASI_WATCHPOINT register bits 39:3. The contents of the ASI_WATCHPOINT register are described in TABLE 10-6i.

TABLE 10-5 ASI_WATCHPOINT Contents

Bit Index	Register Field Name	Description
63:48	-	Reserved
47:40	VA_47_40	Virtual Address bits to match for Instruction or Data Virtual Address comparison; ignored for data physical address comparisons
39:3	Addr_39:3	Virtual or Physical address bits 39:3 to match
2	VA_2	Instruction Virtual Address bit to match; ignored for Data comparisons
1:0	-	Reserved

Reserved bits read as zeroes and are ignored on writes.

Matching is controlled by the ASI_LSU_Control_Register as shown in TABLE 10-7. Each virtual core has a hyperprivileged, read/write ASI_LSU_Control_Register located at ASI 0x45, VA 0x0. Reserved bits read as zeroes and are ignored on writes.

TABLE 10-6 ASI_LSU_CONTROL_REG Contents

Bit Index	Register Field Name	Description
63:35	-	Reserved
34:33	Mode	00 - Disabled 01 – Match on Instruction VA 10 – Match on Data PA 11 – Match on Data VA
32:25	ByteMask	Byte mask to be used with data VA or PA; ignored for instruction virtual address comparison
24	ReadEnable	If 1, enable comparisons for Ifetch or Read accesses
23	WriteEnable	If 1, enable comparisons for data writes
22:5	-	Reserved
4	SpecEnable	If 1, the OpenSPARC T2 core operates in speculative mode (predicts branches not taken, predicts loads to hit in L1, predicts no FP exceptions)
3	DM	If 1, DMMU is enabled
2	IM	If 1, IMMU is enabled
1	DC	If 1, Data Cache is enabled
0	IC	If 1, Instruction Cache is enabled

Other details of masking are described in the OpenSPARC T2 PRM. Virtual address matches are never enabled in hyperprivileged mode.

PSTATE.AM masks instruction and data virtual addresses (so that bits 47:32 of the virtual address are '0') before being presented to the ASI_WATCHPOINT register for comparison. Thus, bits 47:31 of the VA in the ASI_WATCHPOINT register must be set to '0' to match instruction or data virtual addresses when PSTATE.AM is set to '1'.

Additionally, if Core DECR is configured for ASI_WATCHPOINT events the OpenSPARC T2 core will take a debug action as configured by that register.

Since this register is shared between software and scan, debug agents should take care to ensure that only one agent (software or hardware) is configured to use this facility at a time.

10.3.3.3 Trap on Taken Control Transfer

If PSTATE.TCT is set to '1', the OpenSPARC T2 core will take a Control_Transfer_Instruction_Trap each time it executes a taken control transfer instruction. These include conditional branches, jumps, retry, and done instructions. The trap occurs before the instruction has been executed (e.g., is precise). TPC contains the VA of the CTI; TNPC contains the NPC of the CTI. PSTATE.TCT is cleared if the trap is taken.

Additionally, if Core DECR is configured for TCT events the OpenSPARC T2 core will take a debug action as configured by that register.

10.3.3.4 Single Instruction Step

The usage model of Single Instruction Step along with the low level hardware protocols between TCU and OpenSPARC T2 core is described in section 5.3.6.

Each physical OpenSPARC T2 core can be configured to have a single-instruction step feature through hyperprivileged ASI_OVERLAP_MODE reg located at ASI 45, VA 0x10.This register is shown in Appendix 10.3.16. When single step is enabled (on a per core basis), and when the user has issued a Single Step JTAG instruction to TCU , the selected OpenSPARC T2 core(s) will execute one instruction among the available, enabled, and running threads, then stop. The OpenSPARC T2 core will not execute additional instructions until the TCU issues a “resume” command to the core. After a sequence of single steps executed this way, the user might issue a hard stop request to the core being single stepped (as all of the threads of the single stepped core will have been parked at the end of the single steps), and when TCU has turned the clock off to that core, will scan out the core through TAP_SERSCAN instruction non-destructively (by looping back the scan out values) . Then the user will restart the clocks of the core by issuing a TAP_CLOCK_START command . After the clocks have started to run in the single stepped core, the user will issue a JTAG command to Stop Single Step to TCU. TCU will disable the single step mode to the core , unpark all the threads in the core and the core will resume operation on all enabled threads.

Note that OpenSPARC T2 core executes instructions pick, decode, and execute one instruction from each enabled and running thread in series.

Since the ASI_OVERLAP_MODE register is shared between software and scan, debug agents should take care to ensure that only one agent (software : Software debugger/Emulator or hardware : Service Processor) is configured to use this facility at a time.

10.3.3.5 Disable Overlap

Each physical OpenSPARC T2 core can be configured to have a disable overlap feature through hyperprivileged ASI_OVERLAP_MODE reg located at ASI 45, VA 0x10.In this mode, each thread executing on that core will issue one instruction, wait for the instruction to commit and any memory operations to be globally observed, then fetch and execute the next instruction. This mode essentially disables pipelining of all thread's operation. Usage model is same as single instruction step .

Since this ASI_OVERLAP_MODE register is shared between software and scan, debug agents should take care to ensure that only one agent (software or hardware) is configured to use this facility at a time.

10.3.3.6 Soft-Stop Request from TCU to Core

Soft stop is a debug feature controlled by the TCU via the tcu_core_running inputs to the core. The TCU will transition tcu_core_running from 1 to 0 for all threads on a physical core. Each thread will stop issuing instructions, wait for any outstanding cache or TLB misses and operations to complete, and wait for all pending memory accesses issued by the thread to be globally observed. Then each thread will transition spc_core_running_status from 1 to 0. The TCU uses these signals to detect that all threads have quiesced. The TCU will then stop the clocks for that core.

Once the TCU has stopped the core clocks, the core may be scanned without regard to in-flight operations since all crossbar activity initiated by the core will have stopped. However, the core will not respond to any crossbar requests initiated by other agents while it is being scanned. Note that invalidation requests will not be honored while the clocks are stopped or the core is being scanned. This means that the core may become incoherent with the rest of the system unless all cores are stopped in unison.

Following a scan operation, the TCU should restart functional clocks and transition tcu_core_running for each thread from 0 to 1 to allow the core to resume instruction execution.

10.3.3.7 Shadow Scan

OpenSPARC T2 core shadow scan provides access to the PC, HPSTATE,PSTATE ,TL ,TT,TPC, TL_for_TT registers for a given thread. Only one thread can be sampled at a time. The TCU will issue a “shadow scan load” command to the OpenSPARC T2 core. Subsequently, OpenSPARC T2 core will decode the command, and load the appropriate state into the shadow scan string. Then the TCU can scan out the shadow scan string.

OpenSPARC T2 core will have TT, TPC, and a synchronized TL capture (TL_for_TT) to the core shadow scan with the following limitations:

TT, TPC, and TL_for_TT will update ONLY when a trap occurs. (The normal TL field will update for every change in the actual TL register.)

Software writes to TL and done/retry will NOT affect the shadow scan captured values of TT, TPC, and TL_for_TT. So, if the processor traps from TL==0 to TL==1 to TL==2 and then uses done and/or retry to get back to TL==0, shadow scan will still reflect TT[2], TPC[2], and TL_for_TT will still be 2. Similarly, if the processor traps out to TL==2 and then software writes TL to 1 or 0, shadow scan will still show TT[2], TPC[2], and TL_for_TT will still be 2.

If multiple traps occur while the shadow scan is being scanned, the TT, TPC,and TL_for_TT updates due to all traps but the last trap will be lost.

10.3.3.8 Debug Event Control Register

Each physical OpenSPARC T2 core has one hyperprivileged, read/write, Core Debug Event Control Register located at ASI 0x45, VA 0x8, shared by all strands. The DECR controls the stop type (hard or soft) or a trigger pin for an associated event if that event occurs. This register is shown in Appendix 10.3.8.

TCU Action in Response to a soft-stop request asserted by the Core :

If the Core DECR bits for a particular event are configured for a soft-stop (set to 2'b01), and that event occurs, the following sequence of operations results. The OpenSPARC T2 waits for all core activity to quiesce. This means that all in-flight instructions completed (or took an exception), all memory references issued by the core been globally observed, and all activity completed. Then, the OpenSPARC T2 core asserts a spc_softstop_request[7:0] to the TCU, and the TCU subsequently stops the OpenSPARC T2 core's clocks after its cycle counter expires.

The cycle when the stop occurs is a function of the value of the TCU Cycle Counter as well as the transmission delay from the core to the TCU and from the TCU to the clock network in the core. If the TCU Cycle Counter is non-zero when the core generates a soft-stop request, the TCU will decrement the Cycle Counter until it reaches 0. When it reaches 0, the TCU will stop the processor core's clocks (note that it may take several cycles before the processor clocks stop after the counter reaches 0 due to the propagation delay from the TCU to the core clock network).

TCU Action in response to a hard-stop request asserted by the core:

If the Core DECR bits for an event are set to 2'b10, and that event occurs, the OpenSPARC T2 core requests the TCU to stop the clocks as soon as the TCU Cycle Counter reaches 0. The core does not wait for internal core activity to quiesce before raising the spc_hardstop_request[7:0] to the TCU.

TCU Action in response to a trigger request by the core :

If the Core DECR bits for an event are set to 2'b11, and that event occurs, the OpenSPARC T2 core will issue a request on spc_trigger_pulse [7:0] bus to pulse TRIGOUT pin.

If routed to chip I/O, and synchronized to a reasonable lower frequency, the trigger pin may be used to trigger an external agent to begin capturing bus activity or issuing JTAG commands. The pulsing of the pin does not affect operation of the OpenSPARC T2 core in any way. Currently the plan is to use TRIGOUT pin in OpenSPARC T2 for this function.

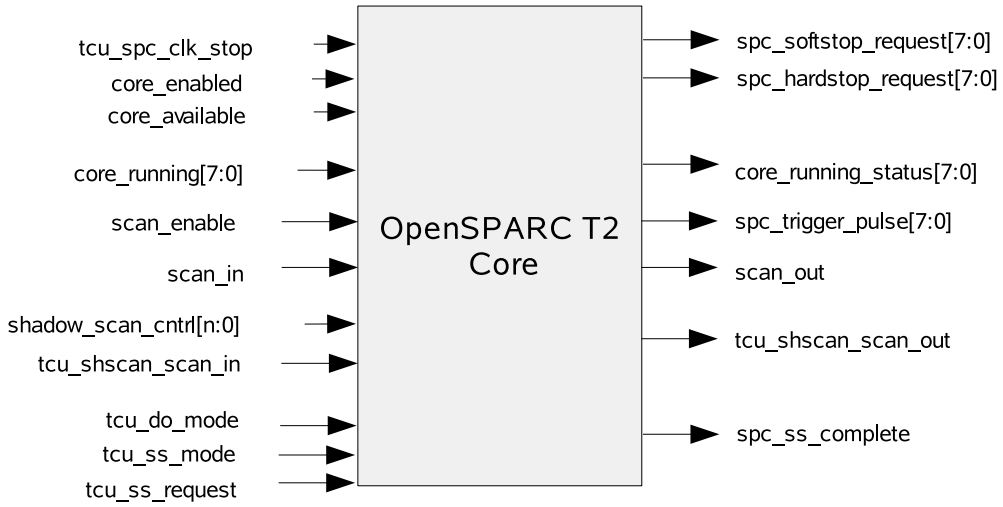
10.4 Core Interface with the TCU

This section outlines the interface between the OpenSPARC T2 core and the TCU for the purposes of describing debug functions. Illustration below shows a high-level diagram of the relevant interface signals (per core).

10.4.1 Clock Interface

The TCU provides a clock stop signal to the flop headers in the core, and drives this signal active when the core is unavailable. The core_enabled signal go to cluster headers .

FIGURE 10-6 OpenSPARDC T2 Core to TCU Debug Interface



10.4.1.1 Tcu_spc_clk_stop

This signal is deasserted to allow the OpenSPARC T2 core's clocks to run. This is the main signal the TCU uses to control the OpenSPARC T2 core's clocks. This signal can be set to '1' at any time to cause stop the OpenSPARC T2 core's clocks.

10.4.1.2 Core_available & Core_enabled

Core_available is set via efuse at manufacturing time and determines whether the physical core can be used in normal operation. It serves as a clock gate and if '0' will result in the clk_stop being asserted to the core (this happens in the TCU). Core_enabled is driven from the ASI_CMP_CORE_ENABLED register and is also used as a clock gate via the cluster header .

10.4.1.3 Core_running[7:0] & Core_running_status[7:0]

The core_running[7:0] bus is an input from the NCU by which the TCU requests the core to park/unpark threads . Parking does not involve stopping the clocks. But, soft stopping requires that the threads be parked before clocks stop.

10.4.1.4 Scan_enable

Besides configuring the scan chains for scanning, this signal also gates off OpenSPARC T2 core's interface signals so that other SOC units do not respond to spurious OpenSPARC T2 core interface activity during scanning. At least the crossbar PCX interface is protected in this way by the tcu_clk_stop signal.

10.4.1.5 Spc_hardstop_request[7:0] & Spc_softstop_request[7:0]

These busses are outputs to the TCU, one bit per thread, which indicate that the core has reached either a hard-stop or a soft-stop condition based on some debug event . These busses are OR'ed inside TCU since stopping can only be done on an entire SPC core. When the Spc_hardstop_request[7:0] or Spc_softstop_request[7:0] is received, the TCU will begin decrementing the Cycle Counter ; when the Cycle Counter reaches 0 the TCU will turn clock off to the requesting core by asserting the tcu_spc_clk_stop signal.

OpenSPARC T2 core asserts core_running_status[7:0] to TCU when all aspects of the instruction have completed (all memory operations globally observed, no pending TLB/Icache misses) and the physical core is completely quiescent. For store operations, the stop will not occur until the store has been globally observed by L2, and the thread's store queue is empty.

10.4.2 Debug Event Interface

This group of core outputs are used to signal either an error or that a debug trigger event has occurred.

10.4.2.1 spc_trigger_pulse[7:0]

This is a bus from the core to TCU covering the 8 threads. If the OpenSPARC T2 core is configured to trigger on an event in the DECR, and the associated event occurs for a thread , the corresponding signal transitions from a '0' to a '1'. It then transitions back to '0', unless another enabled DECR trigger event occurred that cycle. The TCU will pass this signal to a TRIGOUT pin as the OR of the (64) bits from all cores.

10.4.3 Scan Interface

Not all signals relevant to the scan interface are detailed here (e.g., not all the scan clocks and controls are listed).

10.4.3.1 Scan_in

There are three scan chains in each core. All flops on this scan string are reset both at POR and during warm reset unless protected via use of the "warm_reset_flop_header".

10.4.3.2 Scan_out

There are three external scan-out signals per core; each corresponds to a scan-in signal. During JTAG access via scan an entire physical core may be scanned; in this mode the TCU will concatenate the three scan chains in the core, in addition to any JTAG private scan chains such as for shadow scan or memory BIST.

10.4.3.3 Shadow_scan_in

This is the scan-in for the shadow-scan string.

10.4.3.4 Shadow_scan_cntrl[n:0]

This is the control for a shadow scan operation which identifies which thread's state will be sampled to the shadow scan string . The clock will be at JTAG frequency but synchronized to the CPU block by the TCU.

When the TCU wants to do a shadow scan on a particular core, it asserts a tcu_shscan_pce_ov capture signal to that core. At some time later, OpenSPARC T2 core will capture the state requested by the TCU on the internal shadow scan flops. At that point the TCU can scan out the state by accessing the shadow scan scan string. The shadow scan flops are normal flops dedicated to shadow scan and are free-running. When TCU wants to sample, it stops the functional clocks for these flops and scans them.

The signals included in this bus are:

tcu_shscanid[2:0] : selects one of 8 threads

tcu_shscan_pce_ov : provides a capture signal to the shadow scan reg.

tcu_shscan_clk_stop : stops the clock to the shadow scan register to allow it to be scanned via JTAG

tcu_shscan_aclk & tcu_shscan_bclk : shift clocks to perform the scan operation

tcu_shscan_scan_en : a separate scan_enable for the shadow scan register

10.4.3.5 Shadow_scan_out

This is the scan-out of the core's shadow-scan string.

10.4.4 Single Step Mode Signals (and Single Step Usage Model)

Each physical core can be placed in single step mode by the TCU via JTAG. JTAG agent can enter into single step mode at any time without stopping core clocks, but in order to enter into the single step mode at a precise point and have knowledge of the state of the core at that point, the JTAG agent will typically initiate a soft stop based on some specific core debug event (e.g. Instruction VA Watchpoint), in response to which TCU will stop the clock so that JTAG agent can scan out the core to determine state of the core before the single step sequence. After that , putting a physical core in single step mode sequence is controlled by the JTAG agent as follows :

User writes to ASI_overlap_register in TCU (reg R/W by JTAG and SW) to enable single step for any particular core(s), through the JTAG interface to OpenSPARC T2.

User issues a JTAG command to do a Single Step (TAP_single_step)

TCU parks all threads to the core(s) enabled for single step by deasserting core_running[7:0] to the core(s).

All threads indicate they are parked via core_running_status[7:0] to TCU.

TCU asserts tcu_ss_mode to the core(s).

TCU asserts core_running[7:0] for all enabled threads to the core . The thread or threads will not unpark at this time because the single step mode control is asserted. At this point the physical core is in single step mode.

TCU pulses tcu_ss_request for one CMP clk.

Each enabled thread gets unparked and will fetch/execute a single instruction (all unparked threads single step in parallel) and will park again . The TLU will redirect fetch for a single instruction for each unparked thread. These instructions will flow through the pipe. When all threads have quiesced (execution and write back have completed and the store buffers are empty) and parked , the core(s) will pulse spc_ss_complete.

TCU sets a bit in a register visible to JTAG indicating Single Step done.

User reads this register through JTAG to know that Single Step done.

for a sequence of N single steps, execute steps 2:10 N-1 times

User issues a TAP_CLOCK_HSTOP to hard stop the core(s) that were single stepped. Note that hard stop can be used because the cores that were being single stepped have all threads parked/idle, so can be hard stopped.

User issues a TAP_Serial_scan instruction to the core(s) that were hard stopped to serially scan out the contents of the core non-destructively (by using the scan loopback scheme)

After examining the contents of the core regs this way, user issues a TAP_CLOCK_START command to the core(s) that were hard stopped .

User writes to ASI_overlap_register to put the core(s) back to normal mode of operation.

User reads the ASI_Overlap_register to know that core(s) have been put back to normal mode.

User issues a TAP_STOP_SINGLE_STEP command to TCU to get the core(s) out of single step mode.

TCU deasserts core_running[7:0] to the core(s)

Core(s) indicate all threads parked on core_running_status[7:0] to TCU

TCU deasserts tcu_ss_mode to the core(s) taking the core(s) out of single step mode

TCU unparks enabled threads by asserting respective bits in core_running[7:0] bus

Core resumes execution on all enabled threads in normal mode

Note that data integrity may be lost unless all cores are run in single step in unison.

10.4.5 Disable Overlap Mode Signals (and Usage Model)

JTAG agent can enter into disable overlap mode at any time without stopping core clocks, but in order to enter into the disable overlap mode at a precise point and have knowledge of the state of the core at that point, the JTAG agent will typically initiate a soft stop based on some specific core debug event (e.g. Instruction VA Watchpoint), in response to which TCU will stop the clock so that JTAG agent can scan out the core to determine state of the core before the disable overlap sequence. After that , putting a physical core in disable overlap mode sequence is controlled by the JTAG agent as follows:

User writes to Asi_overlap_register in TCU (reg R/W by JTAG and SW) to enable disable overlap for any particular core(s), through the JTAG interface to OpenSPARC T2.

User writes to Counter in TCU (could be same as the one to be used for cycle stepping) to program a count of cycles that TCU is going to the run the cores in disable overlap mode.

User issues a JTAG command to do a Disable Overlap (TAP_disable_overlap)

TCU parks all threads to the core(s) enabled for disable overlap by deasserting core_running[7:0] to the core(s).

All threads indicate they are parked via core_running_status[7:0] to TCU

TCU asserts tcu_do_mode to the core(s).

TCU asserts core_running[7:0] to the core(s), unparking all the enabled threads in the core(s)

Core(s) keep running in disable overlap mode. The TLU will redirect fetch of a single instruction for each unparked thread. These instructions will flow through the pipe. When a given thread has quiesced (execution and write back have completed and the store buffers are empty), the TLU will redirect fetch of a single instruction for that thread.

TCU counter decrements to zero indicating end of disable overlap.

TCU parks all threads by deasserting core_running[7:0] to the core(s)

Core(s) indicate all threads parked on core_running_status[7:0] to TCU

TCU sets a bit in a register visible to JTAG indicating Disable Overlap done.

User reads this register to know that Disable Overlap done.

User issues a TAP_CLOCK_HSTOP to hard stop the core(s) that were disable overlapped. Note that hard stop can be used because the cores that were being disable overlapped have all threads parked/idle, so can be hard stopped.

User issues a TAP_Serial_scan instruction to the core(s) that were hard stopped to serially scan out the contents of the core non-destructively (by using the scan loopback scheme)

after examining the contents of the core regs this way, user issues a TAP_CLOCK_START command to the core(s) that were hard stopped .

TCU turns on the clocks to the core(s) that were hard stopped.

User writes to ASI_overlap_register to put the core(s) back to normal mode of operation.

User reads the ASI_Overlap_register to know that core(s) have been put back to normal mode.

User issues a TAP_STOP_DISABLE_OVERLAP command to TCU to get the core(s) out of disable overlap mode.

TCU deasserts tcu_do_mode to the core(s) taking the core(s) out of disable overlap mode

TCU unparks enabled threads by asserting respective bits in core_running[7:0] bus

Core resumes execution on all enabled threads in normal mode

Note that data integrity may be lost unless all cores are run in single step in unison.

10.5 Debug Block Interface Signals

TABLE 10-7 Debug Block Interface Signal

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
DMU					
dmu_ncu_wrack_ vld	I	1	DMU	iol2clk	CSR Wr Ack from DMU to NCU
dmu_ncu_wrack_tag[3:0]	I	4	DMU	iol2clk	CSR Wr Tag [3:0] from DMU to NCU
dmu_ncu_data[31:0]	I	32	DMU	iol2clk	CSR read data from DMU to NCU
dmu_ncu_vld	I	1	DMU	iol2clk	CSR Data return valid from DMU to NCU
dmu_ncu_stall	I	1	DMU	iol2clk	Stall asserted by DMU to NCU
dmu_sii_hdr_vld	I	1	DMU	iol2clk	DMU requesting to send DMA/Pio Read return/Interrupt packet to SII
dmu_sii_reqbypass	I	1	DMU	iol2clk	DMU requesting to send packet to bypass queue of SII
dmu_sii_datareq	I	1	DMU	iol2clk	DMU requesting to send packet w/data to SII
dmu_sii_datareq16	I	1	DMU	iol2clk	DMU requesting to send packet w/16B only
dmu_sii_data[127:0]	I	128	DMU	iol2clk	Packet from DMU to SII
dmu_sii_be[15:0]	I	16	DMU	iol2clk	Packet byte enables from DMU to SII
dbg1_dmu_stall	O	1	DMU	iol2clk	Request to stall / quiesce DMU -> NCU and DMU -> SII interfaces
dmu_dbg1_stall_ack	I	1	DMU	iol2clk	Ack from DMU indicating DMU -> NCU and DMU -> SII interfaces have quiesced
dbg1_dmu_resume	O	1	DMU	iol2clk	Request to resume packets on DMU -> NCU and DMU -> SII interfaces
dmu_dbg0_debug_bus_a[7:0]	I	8	DMU	iol2clk	Debug Bus A from DMU to DBG0
dmu_dbg0_debug_bus_b[7:0]	I	8	DMU	iol2clk	Debug Bus B from DMU to DBG0
dmu_dbg1_err_event	I	1	DMU	iol2clk	An error event occurred in DMU

TABLE 10-7 Debug Block Interface Signal *(Continued)*

PEU					
peu_mio_debug_bus_a[7:0]	I	8	PEU	pcl2clk	Debug Bus A from PEU to MIO
peu_mio_debug_bus_b[7:0]	I	8	PEU	pcl2clk	Debug Bus B from PEU to MIO
peu_mio_debug_clk	I	1	PEU	Clock	PEU clock to be sent out on Debug port
NIU					
niu_ncu_vld	I	1	NIU	iol2clk	CSR Data return/Interrupt valid from NIU to NCU
niu_ncu_data[31:0]	I	32	NIU	iol2clk	CSR data/ Interrupt packet from NIU to NCU
niu_ncu_stall	I	1	NIU	iol2clk	Stall asserted by NIU to NCU
niu_sii_hdr_vld	I	1	NIU	iol2clk	NIU requesting to send packet to SII
niu_sii_reqbypass	I	1	NIU	iol2clk	NIU requesting to send packet to bypass queue of SII
niu_sii_datareq	I	1	NIU	iol2clk	NIU requesting to send packet w/data to SII
niu_sii_data[127:0]	I	128	NIU	iol2clk	Packet from NIU to SII
niu_sio_dq	I	1	NIU	iol2clk	flow control or credit return signal from NIU to SIO
niu_mio_debug_clock[1:0]	I	2	NIU	Clock	Up to two clocks that niu_dbg_debug_data[31:0] reference
niu_mio_debug_data[31:0]	I	32	NIU	different	NIU debug port signals, coming from upto 2 different NIU clk domains
dbg1_niu_stall	O	1	NIU	iol2clk	Request to stall / quiesce NIU -> NCU and NIU -> SII interfaces
niu_dbg1_stall_ack	I	1	NIU	iol2clk	Ack from NIU indicating NIU -> NCU and NIU -> SII interfaces have quiesced
dbg1_niu_resume	O	1	NIU	iol2clk	Request to resume packets on NIU -> NCU and NIU -> SII interfaces
mio_niu_io2x_clk_ext	O	1	NIU	Clock	Ext NIU clock to NIU from MIO
dbg1_niu_dbg_sel[4:0]	O	5	NIU	static	NIU Debug port select from DBG1
MCU 0					

TABLE 10-7 Debug Block Interface Signal *(Continued)*

mcu0_dbg1_rd_req_in_0[3:0]	I	4	MCU 0	iol2clk	Read Request from L2 bank 0 to MCU 0 (id + valid)
mcu0_dbg1_rd_req_in_1[3:0]	I	4	MCU 0	iol2clk	Read Request from L2 bank 1 to MCU 0 (id + valid)
mcu0_dbg1_rd_request_out[4:0]	I	5	MCU 0	iol2clk	Read ack from MCU to L2 bank 0 or 1 (id + valid + dest_L2_bank)
mcu0_dbg1_wr_req_in_0	I	1	MCU 0	iol2clk	Write req valid from L2 bank 0
mcu0_dbg1_wr_req_in_1	I	1	MCU 0	iol2clk	Write req valid from L2 bank 1
mcu0_dbg1_wr_req_out[1:0]	I	2	MCU 0	iol2clk	0,1,2,3 Writes completed to DRAM
mcu0_dbg1_mecc_err	I	1	MCU 0	iol2clk	MCU 0 has detected an mecc error on a L2 read or scrub
mcu0_dbg1_secc_err	I	1	MCU 0	iol2clk	MCU 0 has detected a secc error on a L2 read or scrub
mcu0_dbg1_fbd_err	I	1	MCU 0	iol2clk	MCU 0 has detected a fbdimm channel error
mcu0_dbg1_err_mode	I	1	MCU 0	iol2clk	Fbdimm interface logic of MCU 0 has gone into error handling mode. This bit stays on until error handling complete.
mcu0_dbg1_err_event	I	1	MCU 0	iol2clk	An error event occurred in MCU 0
MCU 1					
mcu1_dbg1_rd_req_in_0[3:0]	I	4	MCU 1	iol2clk	Read Request from L2 bank 0 to MCU 1 (id + valid)
mcu1_dbg1_rd_req_in_1[3:0]	I	4	MCU 1	iol2clk	Read Request from L2 bank 1 to MCU 1 (id + valid)
mcu1_dbg1_rd_request_out[4:0]	I	5	MCU 1	iol2clk	Read ack from MCU 1 to L2 bank 0 or 1 (id + valid + dest_L2_bank)
mcu1_dbg1_wr_req_in_0	I	1	MCU 1	iol2clk	Write req valid from L2 bank 0
mcu1_dbg1_wr_req_in_1	I	1	MCU 1	iol2clk	Write req valid from L2 bank 1
mcu1_dbg1_wr_req_out[1:0]	I	2	MCU 1	iol2clk	0,1,2,3 Writes completed at DRAM
mcu1_dbg1_mecc_err	I	1	MCU 1	iol2clk	MCU 1 has detected an mecc error on a L2 read or scrub
mcu1_dbg1_secc_err	I	1	MCU 1	iol2clk	MCU 1 has detected a secc error on a L2 read or scrub

TABLE 10-7 Debug Block Interface Signal *(Continued)*

mcu1_dbg1_fbd_err	I	1	MCU 1	iol2clk	MCU 1 has detected a fbdimm channel error
mcu1_dbg1_err_mode	I	1	MCU 1	iol2clk	Fbdimm interface logic of MCU 1 has gone into error handling mode. This bit stays on until error handling complete.
mcu1_dbg1_err_event	I	1	MCU 1	iol2clk	An error event occurred in MCU 1
MCU 2					
mcu2_dbg1_rd_req_in_0[3:0]	I	4	MCU 2	iol2clk	Read Request from L2 bank 0 to MCU 2 (id + valid)
mcu2_dbg1_rd_req_in_1[3:0]	I	4	MCU 2	iol2clk	Read Request from L2 bank 1 to MCU 2 (id + valid)
mcu2_dbg1_rd_request_out[4:0]	I	5	MCU 2	iol2clk	Read ack from MCU 2 to L2 bank 0 or 1 (id + valid + dest_L2_bank)
mcu2_dbg1_wr_req_in_0	I	1	MCU 2	iol2clk	Write req valid from L2 bank 0
mcu2_dbg1_wr_req_in_1	I	1	MCU 2	iol2clk	Write req valid from L2 bank 1
mcu2_dbg1_wr_req_out[1:0]	I	2	MCU 2	iol2clk	0,1,2,3 Writes completed at DRAM
mcu2_dbg1_mecc_err	I	1	MCU 2	iol2clk	MCU 2 has detected an mecc error on a L2 read or scrub
mcu2_dbg1_secc_err	I	1	MCU 2	iol2clk	MCU 2 has detected a secc error on a L2 read or scrub
mcu2_dbg1_fbd_err	I	1	MCU 2	iol2clk	MCU 2 has detected a fbdimm channel error
mcu2_dbg1_err_mode	I	1	MCU 2	iol2clk	Fbdimm interface logic of MCU 2 has gone into error handling mode. This bit stays on until error handling complete.
mcu2_dbg1_err_event	I	1	MCU 2	iol2clk	An error event occurred in MCU 2
MCU 3					
mcu3_dbg1_rd_req_in_0[3:0]	I	4	MCU 3	iol2clk	Read Request from L2 bank 0 to MCU 3 (id + valid)
mcu3_dbg1_rd_req_in_1[3:0]	I	4	MCU 3	iol2clk	Read Request from L2 bank 1 to MCU 3 (id + valid)
mcu3_dbg1_rd_request_out[4:0]	I	5	MCU 3	iol2clk	Read ack from MCU 3 to L2 bank 0 or 1 (id + valid + dest_L2_bank)

TABLE 10-7 Debug Block Interface Signal *(Continued)*

mcu3_dbg1_wr_req_in_0	I	1	MCU 3	iol2clk	Write req valid from L2 bank 0
mcu3_dbg1_wr_req_in_1	I	1	MCU 3	iol2clk	Write req valid from L2 bank 1
mcu3_dbg1_wr_req_out[1:0]	I	2	MCU 3	iol2clk	0,1,2,3 Writes completed at DRAM
mcu3_dbg1_mecc_err	I	1	MCU 3	iol2clk	MCU 3 has detected an mecc error on a L2 read or scrub
mcu3_dbg1_secc_err	I	1	MCU 3	iol2clk	MCU 3 has detected a secc error on a L2 read or scrub
mcu3_dbg1_fbd_err	I	1	MCU 3	iol2clk	MCU 3 has detected a fbdimm channel error
mcu3_dbg1_err_mode	I	1	MCU 3	iol2clk	Fbdimm interface logic of MCU 3 has gone into error handling mode. This bit stays on until error handling complete.
mcu3_dbg1_err_event	I	1	MCU 3	iol2clk	An error event occurred in MCU 3
SII					
sii_dbg1_l2t0_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 0 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
sii_dbg1_l2t1_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 1 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
sii_dbg1_l2t2_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 2 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
sii_dbg1_l2t3_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 3 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
sii_dbg1_l2t4_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 4 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)

TABLE 10-7 Debug Block Interface Signal *(Continued)*

sii_dbg1_l2t5_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 5 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
sii_dbg1_l2t6_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 6 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
sii_dbg1_l2t7_req[1:0]	I	2	SII	l2clk	Req type encoded on 2 bits from sii to L2t 7 (00 : no request, 01 : RDD, 10 : WRI, 11 : WR8)
L2t [7:0]					
l2t0_dbg1_sii_iq_dequeue	I	1	L2t 0	l2clk	L2t 0 dequeue from IQ
l2t1_dbg1_sii_iq_dequeue	I	1	L2t 1	l2clk	L2t 1 dequeue from IQ
l2t2_dbg1_sii_iq_dequeue	I	1	L2t 2	l2clk	L2t 2 dequeue from IQ
l2t3_dbg1_sii_iq_dequeue	I	1	L2t 3	l2clk	L2t 3 dequeue from IQ
l2t4_dbg1_sii_iq_dequeue	I	1	L2t 4	l2clk	L2t 4 dequeue from IQ
l2t5_dbg1_sii_iq_dequeue	I	1	L2t 5	l2clk	L2t 5 dequeue from IQ
l2t6_dbg1_sii_iq_dequeue	I	1	L2t 6	l2clk	L2t 6 dequeue from IQ
l2t7_dbg1_sii_iq_dequeue	I	1	L2t 7	l2clk	L2t 7 dequeue from IQ
l2t0_dbg1_sii_wib_dequeue	I	1	L2t 0	l2clk	L2t 0 dequeue from IOWB
l2t1_dbg1_sii_wib_dequeue	I	1	L2t 1	l2clk	L2t 1 dequeue from IOWB
l2t2_dbg1_sii_wib_dequeue	I	1	L2t 2	l2clk	L2t 2 dequeue from IOWB
l2t3_dbg1_sii_wib_dequeue	I	1	L2t 3	l2clk	L2t 3 dequeue from IOWB
l2t4_dbg1_sii_wib_dequeue	I	1	L2t 4	l2clk	L2t 4 dequeue from IOWB
l2t5_dbg1_sii_wib_dequeue	I	1	L2t 5	l2clk	L2t 5 dequeue from IOWB
l2t6_dbg1_sii_wib_dequeue	I	1	L2t 6	l2clk	L2t 6 dequeue from IOWB
l2t7_dbg1_sii_wib_dequeue	I	1	L2t 7	l2clk	L2t 7 dequeue from IOWB
l2t0_dbg1_err_event	I	1	L2t 0	l2clk	An Error event occurred in l2t 0
l2t1_dbg1_err_event	I	1	L2t 1	l2clk	An Error event occurred in l2t 1
l2t2_dbg1_err_event	I	1	L2t 2	l2clk	An Error event occurred in l2t 2
l2t3_dbg1_err_event	I	1	L2t 3	l2clk	An Error event occurred in l2t 3
l2t4_dbg1_err_event	I	1	L2t 4	l2clk	An Error event occurred in l2t 4

TABLE 10-7 Debug Block Interface Signal *(Continued)*

l2t5_dbg1_err_event	I	1	L2t 5	l2clk	An Error event occurred in l2t 5
l2t6_dbg1_err_event	I	1	L2t 6	l2clk	An Error event occurred in l2t 6
l2t7_dbg1_err_event	I	1	L2t 7	l2clk	An Error event occurred in l2t 7
l2t0_dbg1_pa_match	I	1	L2t 0	l2clk	A PA match detected in l2t 0
l2t1_dbg1_pa_match	I	1	L2t 1	l2clk	A PA match detected in l2t 1
l2t2_dbg1_pa_match	I	1	L2t 2	l2clk	A PA match detected in l2t 2
l2t3_dbg1_pa_match	I	1	L2t 3	l2clk	A PA match detected in l2t 3
l2t4_dbg1_pa_match	I	1	L2t 4	l2clk	A PA match detected in l2t 4
l2t5_dbg1_pa_match	I	1	L2t 5	l2clk	A PA match detected in l2t 5
l2t6_dbg1_pa_match	I	1	L2t 6	l2clk	A PA match detected in l2t 6
l2t7_dbg1_pa_match	I	1	L2t 7	l2clk	A PA match detected in l2t 7
l2t0_dbg1_xbar_vcid[5:0]	I	6	L2t 0	L2clk	VCID[5:0] from Xbar to L2t 0
l2t1_dbg1_xbar_vcid[5:0]	I	6	L2t 1	L2clk	VCID[5:0] from Xbar to L2t 1
l2t2_dbg1_xbar_vcid[5:0]	I	6	L2t 2	L2clk	VCID[5:0] from Xbar to L2t 2
l2t3_dbg1_xbar_vcid[5:0]	I	6	L2t 3	L2clk	VCID[5:0] from Xbar to L2t 3
l2t4_dbg1_xbar_vcid[5:0]	I	6	L2t 4	L2clk	VCID[5:0] from Xbar to L2t 4
l2t5_dbg1_xbar_vcid[5:0]	I	6	L2t 5	L2clk	VCID[5:0] from Xbar to L2t 5
l2t6_dbg1_xbar_vcid[5:0]	I	6	L2t 6	L2clk	VCID[5:0] from Xbar to L2t 6
l2t7_dbg1_xbar_vcid[5:0]	I	6	L2t 7	L2clk	VCID[5:0] from Xbar to L2t 7
L2b[7:0]					
l2b0_dbg1_sio_ctag_vld	I	1	L2b 0	l2clk	Ctag valid from L2b 0 to SIO
l2b1_dbg1_sio_ctag_vld	I	1	L2b 1	l2clk	Ctag valid from L2b 1 to SIO
l2b2_dbg1_sio_ctag_vld	I	1	L2b 2	l2clk	Ctag valid from L2b 2 to SIO
l2b3_dbg1_sio_ctag_vld	I	1	L2b 3	l2clk	Ctag valid from L2b 3 to SIO
l2b4_dbg1_sio_ctag_vld	I	1	L2b 4	l2clk	Ctag valid from L2b 4 to SIO
l2b5_dbg1_sio_ctag_vld	I	1	L2b 5	l2clk	Ctag valid from L2b 5 to SIO
l2b6_dbg1_sio_ctag_vld	I	1	L2b 6	l2clk	Ctag valid from L2b 6 to SIO
l2b7_dbg1_sio_ctag_vld	I	1	L2b 7	l2clk	Ctag valid from L2b 7 to SIO
l2b0_dbg1_sio_ack_type	I	1	L2b 0	l2clk	Read or Wr ack from L2b 0 to SIO
l2b1_dbg1_sio_ack_type	I	1	L2b 1	l2clk	Read or Wr ack from L2b 1 to SIO
l2b2_dbg1_sio_ack_type	I	1	L2b 2	l2clk	Read or Wr ack from L2b 2 to SIO

TABLE 10-7 Debug Block Interface Signal *(Continued)*

l2b3_dbg1_sio_ack_type	I	1	L2b 3	l2clk	Read or Wr ack from L2b 3 to SIO
l2b4_dbg1_sio_ack_type	I	1	L2b 4	l2clk	Read or Wr ack from L2b 4 to SIO
l2b5_dbg1_sio_ack_type	I	1	L2b 5	l2clk	Read or Wr ack from L2b 5 to SIO
l2b6_dbg1_sio_ack_type	I	1	L2b 6	l2clk	Read or Wr ack from L2b 6 to SIO
l2b7_dbg1_sio_ack_type	I	1	L2b 7	l2clk	Read or Wr ack from L2b 7 to SIO
l2b0_dbg1_sio_ack_dest	I	1	L2b0	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 0 to SIO
l2b1_dbg1_sio_ack_dest	I	1	L2b1	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 1 to SIO
l2b2_dbg1_sio_ack_dest	I	1	L2b2	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 2 to SIO
l2b3_dbg1_sio_ack_dest	I	1	L2b3	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 3 to SIO
l2b4_dbg1_sio_ack_dest	I	1	L2b4	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 4 to SIO
l2b5_dbg1_sio_ack_dest	I	1	L2b5	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 5 to SIO
l2b6_dbg1_sio_ack_dest	I	1	L2b6	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 6 to SIO
l2b7_dbg1_sio_ack_dest	I	1	L2b7	l2clk	Read or Wr ack dest (NIU/DMU) from L2b 7 to SIO
TCU					
tcu_mio_dmo_data[39:0]	I	39	TCU	L2clk /1,2,4,8,16	DMO data from TCU to MIO
tcu_mio_dmo_sync	I	1	TCU	L2clk/1,2, 4,8,16	DMO Sync from TCU to MIO
tcu_mio_mbist_done	I	1	TCU	L2clk /10	Membist done from TCU to MIO
tcu_mio_mbist_fail	I	1	TCU	L2clk/ 10	Membist fail from TCU to MIO
tcu_mio_jtag_membist_ mode	I	1	TCU	Static	Membist mode from TCU to MIO
tcu_mio_pins_scan_out[31:0]	I	32	TCU	100 – 200 MHz (tester)	Scan out pins during manufacturing scan
mio_tcu_io_aclk	O	1	TCU	100 – 200 MHz (tester)	A clock during manufacturing scan

TABLE 10-7 Debug Block Interface Signal *(Continued)*

mio_tcu_io_bclk	O	1	TCU	100 – 200 MHz (tester)	B clock during manufacturing scan
mio_tcu_io_scan_en	O	1	TCU	100 – 200 MHz (tester)	Scan Enduring manufacturing scan
mio_tcu_io_ac_test_mode	O	1	TCU	static	AC Testmode
mio_tcu_io_ac_testtrig	O	1	TCU	100 – 200 MHz (tester)	AC TestTrig
mio_tcu_io_scan_in[31:0]	O	32	TCU	100 – 200 MHz (tester)	Scan in pins during manufacturing scan
dbg1_tcu_soc_hard_stop	O	1	TCU	Iol2clk	Hard Stop request to TCU fron SOC
dbg1_tcu_soc_asrt_trigout	O	1	TCU	Iol2clk	Assert TRIGOUT request to TCU from SOC
MIO					
dbg1_mio_drv_imped[1:0]	O	2	MIO	Static	MIO driver impedance control
dbg1_mio_imped_mon	O	1	MIO	Static	Impedence monitoring on/off for IMPED_MON_PU, IMPED_MON_PD pins in OpenSPARC T2.
mio_dbg1_testmode	I	1	MIO	static	Dedicated test mode pin for manufacturing scan
dbg1_mio_dbg_dq[165:0]	O	166	MIO	L2clk/2	OpenSPARC T2 Debug port signals from dbg1
dbg_mio_dbg_ck0	O	1	MIO	Clock	OpenSPARC T2 debug port clock , now generated in MIO
dbg1_mio_drv_en_op_only	O	1	MIO	Static	Drive en to pins configured only as debug port
dbg1_mio_drv_en_muxtest_ op	O	1	MIO	Static	Drive en to pins configured both as debug port and scan out[31:0] pins
dbg1_mio_drv_en_muxbist_ op	O	1	MIO	Static	Drive en to pins configured both as debug port and mbist output pins.
dbg1_mio_drv_en_muxtest_ inp	O	1	MIO	Static	Drive en to pins configured as debug port and testmode input pins

TABLE 10-7 Debug Block Interface Signal *(Continued)*

dbg0_mio_**debug_bus_a[7:0]**	O	8	MIO	iol2clk	Debug Bus A from DBG0 to MIO
dbg0_mio_**debug_bus_b[7:0]**			MIO	iol2clk	Debug Bus B from DBG0 to MIO
CCU					
mio_ccu_cmp_clk_ext	O	1	CCU	Clock	Ext CMP Clk to CCU from MIO
mio_ccu_dr_clk_ext	O	1	CCU	Clock	Ext MCU/DRAM clock to CCU from MIO
mio_ccu_io_clk_ext[11:0]	O	12	CCU	Clock	Ext IO clk to CCU from MIO
io_cmp_sync_en	I	1	CCU	Sync _ en	I/O to cmp clk Sync en consumed by both dbg0 and dbg1
cmp_io2x_sync_en	I	1	CCU	Sync _ en	Cmp to io2x clk Sync En consumed by dbg1 and MIO
Sparcs [7:0]					
spc0_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC0	L2clk	Instruction Commited in Thread Group 0 for SPC 0
spc0_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC0	L2clk	Instruction Commited in Thread Group 1 for SPC 0
spc1_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC1	L2clk	Instruction Commited in Thread Group 0 for SPC 1
spc1_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC1	L2clk	Instruction Commited in Thread Group 1 for SPC 1
spc2_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC2	L2clk	Instruction Commited in Thread Group 0 for SPC 2
spc2_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC2	L2clk	Instruction Commited in Thread Group 1 for SPC 2
spc3_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC3	L2clk	Instruction Commited in Thread Group 0 for SPC 3
spc3_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC3	L2clk	Instruction Commited in Thread Group 1 for SPC 3
spc4_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC4	L2clk	Instruction Commited in Thread Group 0 for SPC 4
spc4_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC4	L2clk	Instruction Commited in Thread Group 1 for SPC 4
spc5_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC5	L2clk	Instruction Commited in Thread Group 0 for SPC 5
spc5_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC5	L2clk	Instruction Commited in Thread Group 1 for SPC 5

TABLE 10-7 Debug Block Interface Signal *(Continued)*

spc6_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC6	L2clk	Instruction Commited in Thread Group 0 for SPC 6
spc6_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC6	L2clk	Instruction Commited in Thread Group 1 for SPC 6
spc7_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC7	L2clk	Instruction Commited in Thread Group 0 for SPC 7
spc7_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC7	L2clk	Instruction Commited in Thread Group 1 for SPC 7
NCU					
ncu_dbg1_error_event	I	1	NCU	Iol2clk	An Error event occurred in NCU (covers some errors in SOC blocks like NIU,DMU,MCU,SII,SIO)
ncu_dbg1_stall	I	1	NCU	Iol2clk	NCU back Pressure control signal to Dbg
ncu_dbg1_vld	I	1	NCU	Iol2clk	NCU to Dbg UCB data valid
ncu_dbg1_data[3:0]	I	4	NCU	Iol2clk	NCU to Dbg UCB data bus
dbg_ncu1_stall	O	1	NCU	Iol2clk	Dbg back pressure control signal to NCU
dbg_ncu1_vld	O	1	NCU	Iol2clk	Dbg to NCU UCB data valid
dbg_ncu1_data[3:0]	O	1	NCU	Iol2clk	Dbg to NCU UCB data
RST					
rst_mio_rst_state[4:0]	I	5	RST	Sys clk	Reset State to MIO

10.6 Debug Blocks (dbg0.v and dbg1.v)

To mitigate wiring congestion issues in OpenSPARC T2, the debug port logic, checkpoint replay logic and SOC debug event logic will be distributed in two top level modules called dbg0.v and dbg1.v. Dbg1 will be located closer to the middle of the chip (close to EFU) as this module will receive signals from all different modules on chip like spc0[7:0],l2t[7:0],l2b[7:0],mcu[3:0],tcu,ncu and sii. While dbg0 will be receiving the repeatability wires from DMU and NIU and will be located closer to those modules. This will have a progressive muxing effect on the debug port signals which will distribute the wires more uniformly over the chip mitigating wiring congestion.

Following are the functions performed by the dbg0.v block :

Converts signals coming from DMU and NIU for the repeatability mode to debug port width of 166 wires @ 2xiol2clk (700 MHz nominal) . This is done through rate conversion logic shown later in the document.

Drives the resultant 166 wide bus to dbg1.v

Following are the functions performed by the dbg1.v block :

Drives and samples several manufacturing test related signals when Debug Port is disabled. Also drives MemBIST signals when debug port is disabled.

Responds to CSR read/write requests from NCU in accordance to UCB protocol . For this purpose it supports a 4 bit UCB interface with NCU which is identical to NCU's UCB interface with RST module.

Hosts I/O mapped CSR to control I/O quiescing of NIU and DMU interfaces to complement Checkpoint/Replay debug feature for OpenSPARC T2. Communicates with NIU and DMU to control quiescing of NIU->SII,SIO,NCU and DMU->SII,SIO,NCU interfaces for checkpoint/replay.

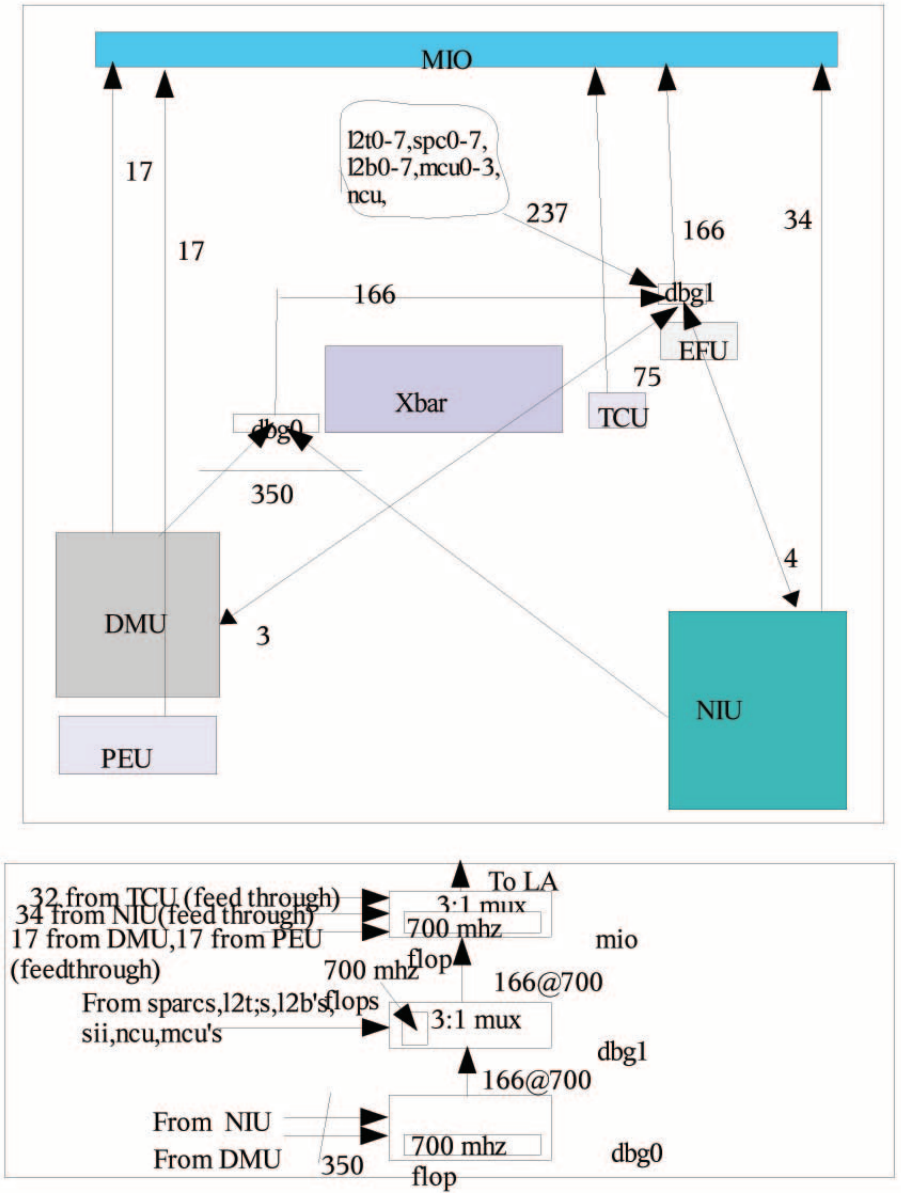
Hosts I/O mapped SOC DECR register to assert Hard Stop or pulse TRIGOUT request to TCU based on various SOC debug events.

Hosts I/O mapped CSR to configure debug port in any one of 5 modes . Generates mux select s to mio.sv to select between NIU debug mode, PCI_EX debug mode and OpenSPARC T2 Repeatability/Tester Charac mode/SOC Obs mode.

Converts signals coming from rest of the chip for Tester charac/cpu debug mode of debug port and SOC observability mode of debug port to debug port width of 166 wires @ 2xiol2clk (700 MHz nominal) . This is done through rate conversion logic shown later in the document.

Muxes signals coming from dbg0.v (repeatability signals : 166 wires) with the tester charac/CPU debug mode and SOC obs mode signals and drives the 166 wire debug port bus to mio.sv at a data rate of 2xiol2clk (700 MHz nominal).

FIGURE 10-7 DBG0 and DBG1 in OpenSPARC T2 Floorplan



10.6.1 OpenSPARC T2 Debug Port

OpenSPARC T2 debug port width is defined by 166 signals for repeatability to complement Checkpoint /Replay . When not being used to monitor the repeatability signals (described in section 3.2.1.1), the port will get used to monitor various other signals in OpenSPARC T2 in 4 different modes : SOC Observability,Tester charac/CPU debug , PCI_EX debug and NIU Debug.

These modes are programmable by SW by writing to the OpenSPARC T2 Debug Port Configuration register. In all the above 5 modes other than the NIU debug mode and PCI_EX debug modes, the debug port will be driven @ 2 x iol2clk frequency (2 x 350 MHz = 700 MHz nominal), with iol2clk being sent out on DBG_CK0 pin to the LA for sampling and aligning the data. In essence this is equivalent to data being driven on both edges of iol2clk . Commercially available LA's do have the ability to support DDR signal sampling with the LA currently being able to support a max of 900 MHz DDR (both edges of 450 MHz clk). OpenSPARC T2's debug port will employ double pumping CMOS signals @ 1.1 V and will not need to meet the timing and skew specs associated with traditional Memory multi-drop DDR2 interfaces. Also the LA probes will be connector less thereby reducing the load on the debug port drivers.

As mentioned before, the debug port pins will be shared with manufacturing scan test and memBIST signals so that with the debug ports disabled , some of these pins can be used for manufacturing scan and MemBIST of OpenSPARC T2. The muxing of the debug port signals with the manufacturing scan test and memBIST signals will happen in the I/O cell itself in the mio.v block.

Upon chip reset , the debug port will come up disabled thereby saving power on the I/O's. The debug port can be enabled by writing to the Debug_en bit of the Debug Port Configuration Register (either by SW or by JTAG CREGs access) . The effect of the write will take place immediately and not after the next warm reset.

The muxing of the debug signals in OpenSPARC T2 on the debug port and also muxing of the debug port signals with the manufacturing scan test signals,memBIST signals and other miscellaneous signals is shown in the figure below .

The I/O's in OpenSPARC T2 debug port can be thus broadly classified as falling under 5 categories :

I/Os which are shared between debug port and memBIST signals that are outputs. For this group of signals , the Drive_en to the I/O's will get generated as :

```
assign dbg_mio_drv_en_muxbist_op = debug_en  | tcu_dbg_jtag_memBIST_mode;
```

I/Os which are shared between debug port and Manufacturing Scan test signals that are outputs. For this group of signals, the Drive_en to the I/O's will get generated as follows :

```
assign dbg_mio_drv_en_muxtest_op = debug_en | mio_dbg_testmode;
```

I/O's which are shared between debug port and Manufacturing Scan test signals that are inputs. For this group of signals, the Drive_en to the I/O's will get generated as follows :

assign dbg_mio_drv_en_muxtest_inp = debug_en & ~mio_dbg_testmode;

I/O's which are always driven as outputs in the debug mode . For this group of signals, the Drive_en to the I/O's will get generated as follows :

assign dbg_mio_drv_en_op_only = debug_en.

Where “debug_en” is “Debug_En” bit in Debug Port Config register.

Legend : 2x {bus} implies twice the data contained in {bus} gets driven out on debug port on every io2xclk cycle x (bus} implies half the data contained in {bus} gets driven out on

debug port on every io2xclk cycle, with the other half following in the next io2xclk cycle.

TABLE 10-8 Mapping

000 : SOC Observability	165: 160 155:0	{rst_mio_rst_state[5:0],4'b0, 2x(sii_dbg1_l2t7_req[1:0],l2t7_dbg1_sii_iq_dequeue,l2t7_dbg1_sii_wib_dequeue, l2b7_dbg1_sio_ctag_vld,l2b7_dbg1_sio_ack_type, l2b7_dbg1_sio_ack_dest, sii_dbg1_l2t6_req[1:0],l2t6_dbg1_sii_iq_dequeue,l2t6_dbg1_sii_wib_dequeue, l2b6_dbg1_sio_ctag_vld, l2b6_dbg1_sio_ack_type, l2b6_dbg1_sio_ack_dest, sii_dbg1_l2t5_req[1:0],l2t5_dbg1_sii_iq_dequeue, l2t5_dbg1_sii_wib_dequeue, l2b5_dbg1_sio_ctag_vld, l2b5_dbg1_sio_ack_type, l2b5_dbg1_sio_ack_dest, sii_dbg1_l2t4_req[1:0],l2t4_dbg1_sii_iq_dequeue, 2t4_dbg1_sii_wib_dequeue, l2b4_dbg1_sio_ctag_vld, l2b4_dbg1_sio_ack_type, l2b4_dbg1_sio_ack_dest, sii_dbg1_l2t3_req[1:0],l2t3_dbg1_sii_iq_dequeue, l2t3_dbg1_sii_wib_dequeue, dbg0_dbg1_l2b3_sio_ctag_vld, dbg0_dbg1_l2b3_sio_ack_type, dbg0_dbg1_l2b3_sio_ack_dest, sii_dbg1_l2t2_req[1:0],dbg0_dbg1_l2t2_sii_iq_dequeue, dbg0_dbg1_l2t2_sii_wib_dequeue, dbg0_dbg1_l2b2_sio_ctag_vld, dbg0_dbg1_l2b2_sio_ack_type, dbg0_dbg1_l2b2_sio_ack_dest, sii_dbg1_l2t1_req[1:0],l2t1_dbg1_sii_iq_dequeue,l2t1_dbg1_sii_wib_dequeue, dbg0_dbg1_l2b1_sio_ctag_vld,dbg0_dbg1_l2b1_sio_ack_type, dbg0_dbg1_l2b1_sio_ack_dest, sii_dbg1_l2t0_req[1:0], dbg0_dbg1_l2t0_sii_iq_dequeue, dbg0_dbg1_l2t0_sii_wib_dequeue, dbg0_dbg1_l2b0_sio_ctag_vld, dbg0_dbg1_l2b0_sio_ack_type, dbg0_dbg1_l2b0_sio_ack_dest), 2'b0, (mcu0_dbg1_rd_req_in_0[3:0],mcu0_dbg1_rd_req_in_1[3:0],mcu0_dbg1_rd_req_out[4:0], mcu0_dbg1_wr_req_in_0,mcu0_dbg1_wr_req_in_1,mcu0_dbg1_wr_req_out[1:0],mcu0_dbg1_mecc_err, mcu0_dbg1_secc_err,mcu0_dbg1_fbd_err,mcu0_dbg1_err_mode,mcu1_dbg1_rd_req_in_0[3:0],mcu1_dbg1_rd_req_in_1[3:0], mcu1_dbg1_rd_req_out[4:0],mcu1_dbg1_wr_req_in_0, mcu1_dbg1_wr_req_in_1, mcu1_dbg1_wr_req_out[1:0],mcu1_dbg1_mecc_err, mcu1_dbg1_secc_err, mcu1_dbg1_fbd_err, mcu1_dbg1_err_mode, mcu2_dbg1_rd_req_in_0[3:0], mcu2_dbg1_rd_req_in_1[3:0],mcu2_dbg1_rd_req_out[4:0],mcu2_dbg1_wr_req_in_0, mcu2_dbg1_wr_req_in_1,mcu2_dbg1_wr_req_out[1:0],mcu2_dbg1_mecc_err, mcu2_dbg1_secc_err, mcu2_dbg1_fbd_err, mcu2_dbg1_err_mode, mcu3_dbg1_rd_req_in_0[3:0], mcu3_dbg1_rd_req_in_1[3:0],mcu3_dbg1_rd_req_out[4:0], mcu3_dbg1_wr_req_in_0,mcu3_dbg1_wr_req_in_1, mcu3_dbg1_wr_req_out[1:0], mcu3_dbg1_mecc_err,mcu3_dbg1_secc_err,mcu3_dbg1_fbd_err,mcu3_dbg1_err_mode) }

TABLE 10-8 Mapping *(Continued)*

001 : Tester Charac/CPU Debug	159:0	{6'b0, 2x (spc7_dbg1_instr_cmt_grp1[1:0],spc7_dbg1_instr_cmt_grp[1:0], spc6_dbg1_instr_cmt_grp1[1:0], spc6_dbg1_instr_cmt_grp[1:0], spc5_dbg1_instr_cmt_grp1[1:0],spc5_dbg1_instr_cmt_grp0[1:0], spc4_dbg1_instr_cmt_grp1[1:0], spc4_dbg1_instr_cmt_grp0[1:0], spc3_dbg1_instr_cmt_grp1[1:0],spc3_dbg1_instr_cmt_grp0[1:0], dbg0_dbg1_spc2_instr_cmt_grp1[1:0], dbg0_dbg1_spc2_instr_cmt_grp0[1:0], spc1_dbg1_instr_cmt_grp1[1:0], spc1_dbg1_instr_cmt_grp0[1:0], dbg0_dbg1_spc0_instr_cmt_grp1[1:0], dbg0_dbg1_spc0_instr_cmt_grp0[1:0], l2t7_dbg1_xbar_vcid[5:0], l2t6_dbg1_xbar_vcid[5:0], l2t5_dbg1_xbar_vcid[5:0],l2t4_dbg1_xbar_vcid[5:0], l2t3_dbg1_xbar_vcid[5:0],dbg0_dbg1_l2t2_xbar_vcid[5:0], l2t1_dbg1_xbar_vcid[5:0],dbg0_dbg1_l2t0_xbar_vcid[5:0]) }

010 : Repeatability	165:0	{
		x (niu_ncu_vld,niu_ncu_data[31:0],niu_ncu_stall,
		niu_sii_hdr_vld,niu_sii_reqbypass, niu_sii_datareq, niu_sio_dq,niu_sii_data[127:0]),
		x (dmu_ncu_data_fnl[11:0],dmu_ncu_wrack_vld,
		dmu_ncu_wrack_tag[3:0],dmu_ncu_stall, dmu_sii_hdr_vld,
		dmu_sii_reqbypass,dmu_sii_datareq, dmu_sii_datareq16,
		dmu_sii_be[15:0],dmu_sii_data[127:0])
		}
		where, dmu_ncu_data_fnl[11:0] = 1/3
		{{dmu_ncu_vld_r,dmu_ncu_data_r[10:0], dmu_ncu_vld_r,
		dmu_ncu_data_r[21:11], dmu_ncu_vld_r, 1'b0,dmu_ncu_data_r[31:22]}

TABLE 10-8 Mapping *(Continued)*

011 : CORE_SOC debug	149:86 82:0	{ 16'b0, 2x (spc7_dbg1_instr_cmt_grp1[1:0],spc7_dbg1_instr_cmt_grp0[1:0],spc6_dbg1_instr_cmt_grp1[1:0],spc6_dbg1_instr_cmt_grp0[1:0],spc5_dbg1_instr_cmt_grp1[1:0],spc5_dbg1_instr_cmt_grp0[1:0],spc4_dbg1_instr_cmt_grp1[1:0],spc4_dbg1_instr_cmt_grp0[1:0],spc3_dbg1_instr_cmt_grp1[1:0],spc3_dbg1_instr_cmt_grp0[1:0],dbg0_dbg1_spc2_instr_cmt_grp1[1:0],dbg0_dbg1_spc2_instr_cmt_grp0[1:0],spc1_dbg1_instr_cmt_grp1[1:0],spc1_dbg1_instr_cmt_grp0[1:0],dbg0_dbg1_spc0_instr_cmt_grp1[1:0],dbg0_dbg1_spc0_instr_cmt_grp0[1:0]), 3'b0, x (dmu_ncu_data_fnl[11:0],dmu_ncu_wrack_vld, dmu_ncu_wrack_tag[3:0],dmu_ncu_stall, dmu_sii_hdr_vld, dmu_sii_reqbypass,dmu_sii_datareq, dmu_sii_datareq16, dmu_sii_be[15:0],dmu_sii_data[127:0]) } where, dmu_ncu_data_fnl[11:0] = 1/3 {{dmu_ncu_vld_r,dmu_ncu_data_r[10:0], dmu_ncu_vld_r, dmu_ncu_data_r[21:11], dmu_ncu_vld_r, 1'b0,dmu_ncu_data_r[31:22]}
100 : NIU Debug	157:124	165:158 : dont care 157:124 : {niu_mio_debug_data[31:0], niu_mio_debug_clock[1:0]} 123:0 : dont care
101 : PCI_EX Debug	123:9182:0	123:91 : {dbg0_mio_debug_bus_a_r[7:0],dbg0_mio_debug_bus_b_r[7:0], peu_mio_debug_bus_a[7:0],peu_mio_debug_bus_b[7:0],peu_mio_debug_clk} 82:0 : x (dmu_ncu_data_fnl[11:0],dmu_ncu_wrack_vld, dmu_ncu_wrack_tag[3:0],dmu_ncu_stall, dmu_sii_hdr_vld, dmu_sii_reqbypass,dmu_sii_datareq, dmu_sii_datareq16, dmu_sii_be[15:0],dmu_sii_data[127:0]) } where, dmu_ncu_data_fnl[11:0] = 1/3 {{dmu_ncu_vld_r,dmu_ncu_data_r[10:0], dmu_ncu_vld_r, dmu_ncu_data_r[21:11], dmu_ncu_vld_r, 1'b0,dmu_ncu_data_r[31:22]}

FIGURE 10-8 OpenSPARC T2 Debug Port layout across DBG0,DBG1 and MIO

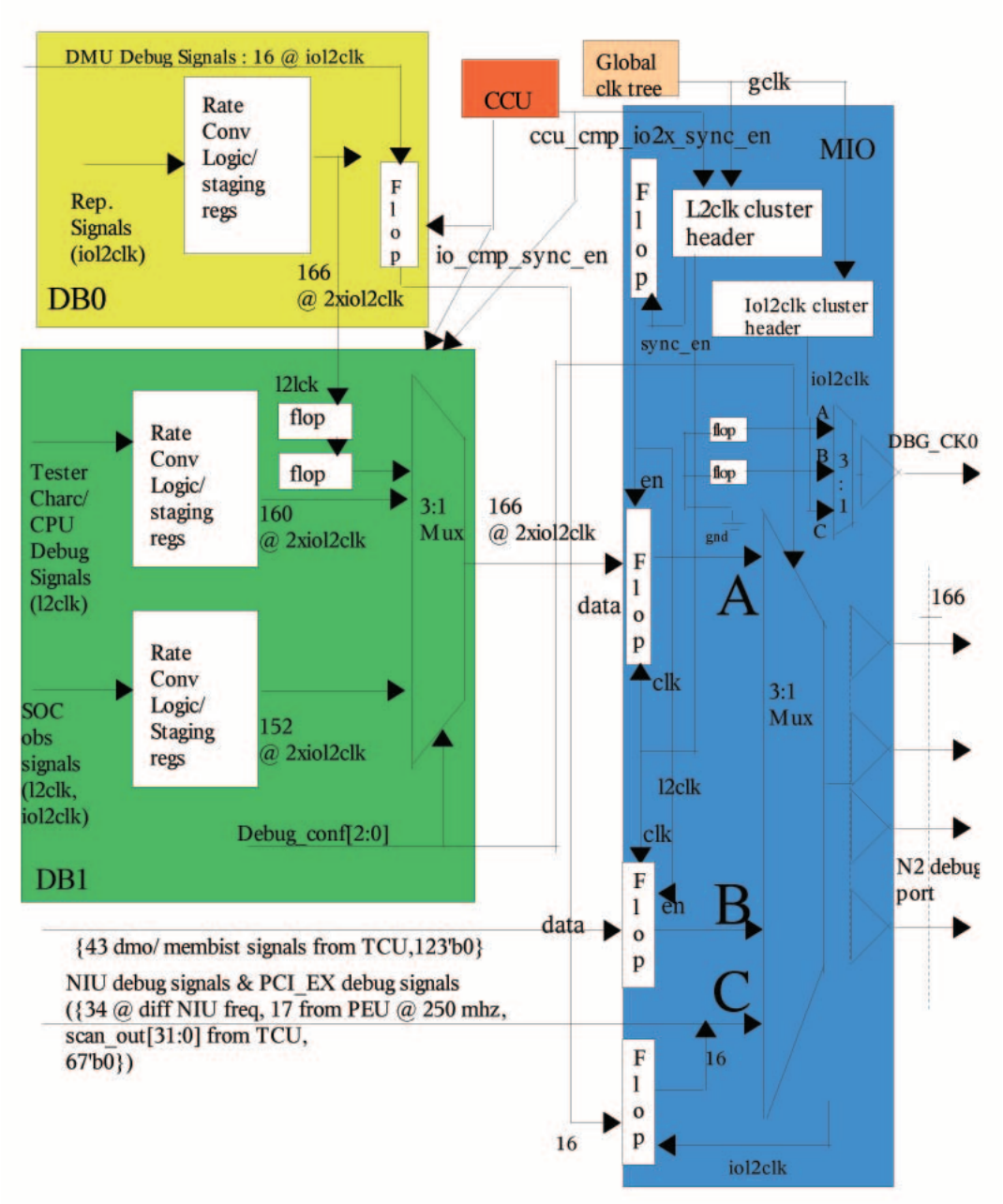


FIGURE 10-9 Rate Conversion from iol2clk to io2xclk

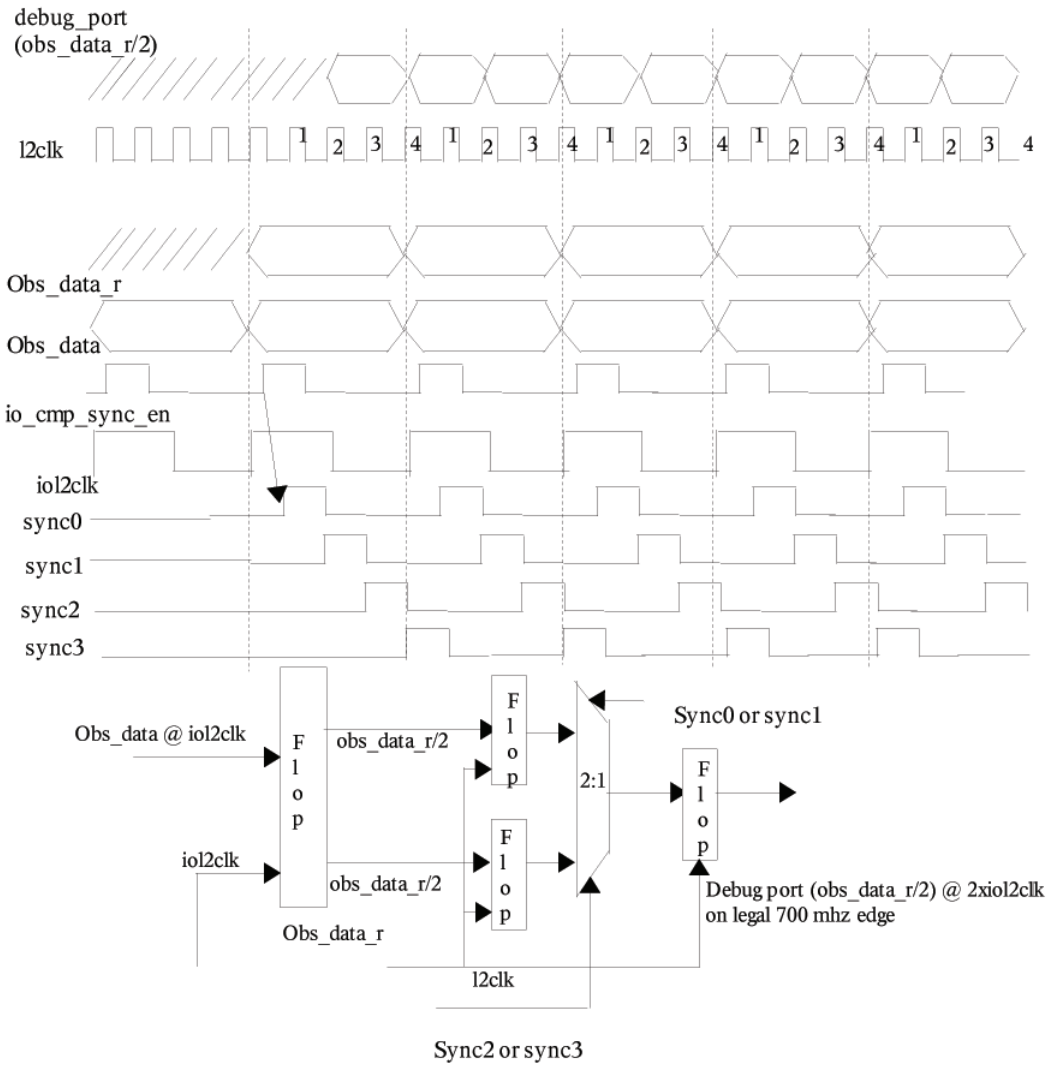
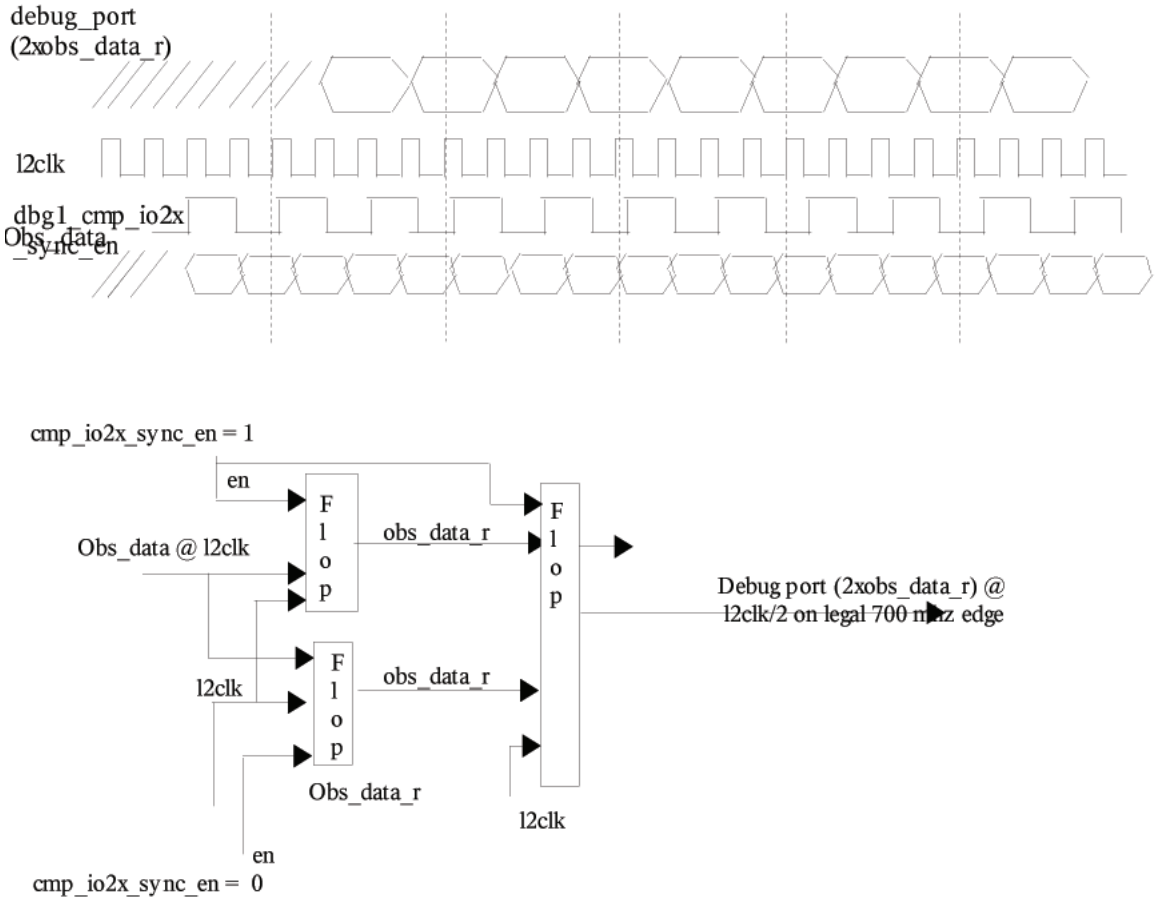


FIGURE 10-10 Rate Conversion from l2clk to io2xclk



10.6.2 CSR Block in debug.v

The CSR block in debug.v will host the OpenSPARC T2 Debug port Config register,the OpenSPARC T2 I/O Quiesce Control register and the SOC DECR register . These registers are all defined in the Appendix section of this document. This module will be operating at iol2clk frequency and will communicate with NCU ,TCU,DMU and NIU.

It will have a 4 bit standard UCB interface with NCU similar to the UCB interface between NCU and RST and NCU and TCU. It will be able to respond to CSR read/write requests on this UCB interface from the NCU initiated either by the SPARCs or JTAG (CREG access from TCU).

W.r.t. TCU, it will have a a pair of signals : dbg_tcu_soc_hard_stop and dbg_tcu_soc_asrt_trigout to request hard stop and TRIGOUT pulsing respectively due to occurrence of some SOC debug event. The SOC debug event sampling logic will be working at iol2clk frequency . So all debug events that arrive at l2clk frequency as pulses(e.g L2 PA matches) will need to be synchronized to a iol2clk pulse before being sampled @ iol2clk (first 0 to 1 transition) . All debug events that come as levels either at iol2clk or l2clk will be sampled @ iol2clk (first 0 to 1 transition) . All debug events that come as pulses in iol2clk domain will also be sampled @ iol2clk (first 0 to 1 transition) . Then the result of the sampling logic for all the respective debug events will get Ored and based on what is programmed in SOC DECR register , will pulse either dbg_tcu_soc_hard_stop or dbg_tcu_soc_asrt_trigout for one iol2clk cycle . If there is a request for hard stop and TRIGOUT assertion both in the same iol2clk cycle, both wires will be pulsed simultaneously to TCU for one iol2clk cycle.

W.r.t. NIU and DMU it will support separate interfaces to control I/O quiescing of NIU and DMU individually to complement checkpoint replay. The details of this protocol and SW-HW handshake has already been described in 3.2.2.

10.7 APPENDIX

10.7.1 Checkpoint Sequence (SW-HW interaction)

prior to booting OS:

==========

reserve at least half of system DRAM for checkpoint code/dump

enable timer tick interrupts on all threads

to take checkpoint:

==========

tick interrupt jumps into hypervisor code on each thread (this will happen at approx. the same time on all threads as ticks are synchronized).

each thread does following sparcv9 state dump:

dump ARF/FRF

dump trap/pstate regs

dump hpriv state regs

dump global regs

dump MMU config regs

dump scratch regs

dump interrupt pending register

write local regs into scratch regs so we can reuse %l's

one thread from each core dumps the ITLB and DTLB

** master thread stalls IO DMA

all threads jump into spin loop waiting for others to arrive

**wait for pending DMA to complete

master thread dumps all active pages of dram (can make this multiple threads to save time). Active pages are tagged using software tricks to minimize how much dumping is required.

do debug init sequence - see below

**enable DMA

restore local regs

restore scratch regs

all threads jump into spin loop waiting for others to arrive

program tick compare to time of next checkpoint

retry back into normal execution

the following is required to get all the flops in the core blocks in a known state. Careful alignment of code and reset handler is required to ensure allocation in caches is predictable. All code from reset vector to dram refresh should hit in i$, to avoid repeatability problems. If we plan to reset the MCU and use self refresh mode, we'll make sure the whole reboot sequence is in the l2/l1$ before the reset.

debug init sequence:

==========

halt all threads except master

put l2$ into direct mapped mode

clear VUAD bits.

flush l2$ (implies l1$ flush too)

**dump NCU interrupt state to memory.

Wait 1 microsec for all pending MCU transactions to complete to memory (The worst case time to flush 16 writes in each MCU if no reads are present is about 660 ns)

**initiate debug reset - assume short enough to not drop excessive DRAM

refreshes - or use self refresh if we reset the MCU

** ..reboot out of dram/l2...

** write MCU refresh counters

go back to 16 way l2$ mode

** ..restore hpriv regs from saved area

** ..execute done to get back to where we were

** reload regs from dumped state that we cleared by reset

** reload NCU regs.

** Probe all NIU interrupt sources and poke interrupt into NCU or cores for all dropped interrupts.

debug port info:

==========

debug port dumps NIU and PCI-ex traffic to pins

sync point is deemed to be the end of the debug reset. thus we need to be able to

observe the end of the debug reset on an external pin somehow.

clock alignment:

==========

OpenSPARC T1 debug init ensures a known clock alignment. Need to prove OpenSPARC T2 reset scheme will do the same, unless clocks alignment is always the same for given ratio

10.7.2 SW Visible State Lost on Debug Reset

The following table shows all the SW visible registers in the synchronous portion of OpenSPARC T2 (excluding PCI-EX and NIU blocks) that will maintain their value over "debug_reset".

TABLE 10-9 State that Loses Value over debug_reset (excluding NIU and PCI_EX)

Name	Fields	POR	WMR/DBR
PSTATE	TCT	0 (Trap on control transfer)	0 (Trap on control transfer)
PSTATE	MM	0 (TSO)	0 (TSO)
PSTATE	RED	0 (RED_state bit is in HPSTATE register)	0 (RED_state bit is in HPSTATE register)
PSTATE	PEF	1 (FPU on)	1 (FPU on)
PSTATE	AM	0 (Full 64-bit addresses)	0 (Full 64-bit addresses)
PSTATE	PRIV	0 (Hyperpriviledged mode)	0 (Hyperpriviledged mode)
PSTATE	IE	0 (Disable interrupts)	0 (Disable interrupts)
PSTATE	AG	0 (Alternate globals always 0)	0 (Alternate globals always 0)
PSTATE	CLE	0 (Current not little endian)	0 (Current not little endian)
PSTATE	TLE	0 (Trap not little endian)	0 (Trap not little endian)
PSTATE	IG	0 (Interrupt globals always 0)	0 (Interrupt globals always 0)
PSTATE	MG	0 (MMU globals always 0)	0 (MMU globals always 0)
HPSTATE	IBE	0 (Instruction breakpoint disabled)	0 (Instruction breakpoint disabled)
HPSTATE	RED	1 (RED_state)	1 (RED_state)
HPSTATE	HPRIV	1 (Hyperprivileged mode)	1 (Hyperprivileged mode)
HPSTATE	TLZ	0 (TLZ traps disabled)	0 (TLZ traps disabled)

TABLE 10-9 State that Loses Value over debug_reset (excluding NIU and PCI_EX) *(Continued)*

TT[TL	TT[TL	1	1
TPC[TL] TnPC[TL]	TPC[TL] TnPC[TL]	Unknown Unknown	PC nPC
TL	TL	MAXTL	MAXTL
GL	GL	MAXGL	MAXGL
TSTATE[TL	GL	Unknown	Unknown
TSTATE[TL	CCR	Unknown	Unknown
TSTATE[TL]	ASI	Unknown	Unknown
TSTATE[TL	PSTATE	Unknown	Unknown
TSTATE[TL	CWP	Unknown	Unknown
HTSTATE[TL]	IBE	Unknown	Unknown
HTSTATE[TL]	RED	Unknown	Unknown
HTSTATE[TL]	HPRIV	Unknown	Unknown
HTSTATE[TL]	TLZ	Unknown	Unknown
TICK	NPT	1	1
TICK	Counter	Unknown	Count
PERF_CONTROL (PCR)	all	0 (off)	0 (off)
PERF_COUNTER (PIC)		0	0
ASI_INST_MASK_REG		0	0
ASI_LSU_DIAG_REG		0	0
ASI_ERROR_INJECT_REG		0	0
ASI_LSU_CONTROL_REG		0	0
ASI_DECR		0	0
ASI_CERER		0	0
ASI_CETER		0	0
ASI_SPARC_PWR_MGMT		0	0

TABLE 10-9 State that Loses Value over debug_reset (excluding NIU and PCI_EX) *(Continued)*

ASI_IMMU_TAG_TARGET		0	0
ASI_IMMU_SFSR		0	0
ASI_IMMU_TAG_ACCESS		0	0
L2 Error Injection Reg		0	0
L2 Error En Reg		0	0
DRAM Error Injection Reg		0	0
SSI Timeout Reg		0x800000	0x800000
L2 Control Reg		0x1	0x1
L2 Diag Data		X	X
L2 Diag Tag		X	X
L2 Diag VD		X	X
L2 Diag UA		X	X
L2 Bist control reg		0	0
SPARC Bist Control Reg		0	0
NCU Core running RW Reg		0	0
NCU L2 Bank Enable Reg		Bank_avail	Bank_avail
NCU L2 Index Hash Enable		0	0
NCU PCIE LinkA Mem32 Addr Offset Base		0	0
NCU PCIE LinkA Mem32 Addr Offset Mask		0	0
NCU PCIE LinkA Mem64 Domain Addr Base		0	0
NCU PCIE LinkA Mem64 Domain Addr Mask		0	0

TABLE 10-9 State that Loses Value over debug_reset (excluding NIU and PCI_EX) *(Continued)*

NCU PCIE LinkA IOConfig Domain Addr Base		0	0
NCU PCIE LinkA IOConfig Domain Addr Mask		0	0
NCU PCIE Link A Flush		0	0

10.7.3 Registers to Support Debug

10.7.3.1 Debug Port Configuration Register

This register is used to enable and configure the debug port in any one of 6 modes . It is located in debug.v module at location 0x86-0000-0000. The format of this register is shown in :TABLE 10-11

TABLE 10-10 Debug Port Configuration Register

Field	Bit Position	POR Value	R/W	Description
IMP_CTRL	63:62	0x0 Preserved on WMR/DBR	R/W	MIO Driver Impedance Control. 11 : Strong Driver 10 : Nominal Driver 01 : Weak Driver 00 : Low Power Driver
IMPED_MON_EN	61	0 Preserved on WMR/DBR	R/W	Impedence monitoring on/off for IMPED_MON_PU, IMPED_MON_PD pins in OpenSPARC T2. 1 : on 0 : off
RSVD	60:10	0x0	RO	Reserved, Read as 0.
NIU_DBG_SEL	9:5	0x0 Preserved on WMR/DBR	R/W	NIU debug select bits ,sent out on dbg1_niu_dbg_sel[4:0] wires

TABLE 10-10 Debug Port Configuration Register *(Continued)*

Debug_Train	4	0x0 Preserved on WMR/DBR	R/W	When set to 1, enables Training for Debug port in modes 000,001, 010 and 011
Debug_Conf	3:1	0 Preserved on WMR/DBR	R/W	Debug Port Configuration 000 : SOC Observability 001 : Tester Charac/CPU debug 010 : Repeatability 011 : CORE_SOC debug 100 : NIU Debug 101 : PCI_EX Debug 110 – 111 : Reserved
Debug_En	0	0 Preserved on WMR/DBR	R/W	When set to 1, enables debug port drivers

10.7.3.2 RESET_GEN Register

The reset generation register, shown below , is provided to allow software to generate XIR resets to all processors specified in the ASI_XIR_STEERING register or a chipwide warm or debug reset.

TABLE 10-11 Reset Generation Register RESET_GEN (0x89-0000-0808)

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:4	0	RO	Reserved
DBR_GEN	3	0	R/W	Set to one to generate Debug Reset. Value is automatically cleared once the DBR is complete.
RSVD1	2	0	RO	Reserved (was POR_GEN on Fire).
XIR_GEN	1	0	R/W	Set to one to generate a XIR. Value is automatically cleared once the XIR is complete.
WMR_GEN	0	0	R/W	Set to one to generate a WMR. Value is automatically cleared once the WMR is complete.

10.7.3.3 RESET_SOURCE Register

The reset source register, shown below allows software to identify the source of a reset. The bits in this register are write-one to clear.

TABLE 10-12 Reset Source Register RESET_SOURCE (0x89-0000-0818)

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:8	0	RO	Reserved
DBR_GEN	7	0	R/W1C	Software wrote a 1 to the DBR_GEN field of the RESET_GEN register.
FATAL	6	0	R/W1C	The L2 cache detected a fatal error.
PB_XIR	5	0	R/W1C	The user asserted the BUTTON_XIR_ input pin.
PB_RST	4	0	R/W1C	The user asserted the PB_RST_ input pin.
POR	3	1	R/W1C	The system processor asserted the POR_ input pin
RSVD1	2	0	RO	Reserved (was POR_GEN on Fire).
XIR_GEN	1	0	R/W1C	Software wrote a 1 to the XIR_GEN field of the RESET_GEN register.
WMR_GEN	0	0	R/W1C	Software wrote a 1 to the WMR_GEN field of the RESET_GEN register.

10.7.3.4 ASI_WMR_VEC_MASK Register

All physical cores share a hyperprivileged, read/write ASI_WMR_VEC_MASK register located as ASI 0x45, VA 0x18. Reserved bits read as zero and are ignored on write. The contents of this register are preserved across warm reset and debug reset . This register will be physically located in the NCU block (ncu.sv). The format of the register is as follows :

TABLE 10-13 ASI_WMR_VEC_MASK Reg Format

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:1	0	RO	Reserved
VEC_MASK	0	0	R/W	If `1', trap to 0x0000000000000020 instead of 0xFFFFFFFF0000020. Value preserved across warm reset and debug reset.

10.7.3.5 MCU Channel Read Latency Register

This register is at location 0x84_0000_08B8 . The format is as follows :

TABLE 10-14 MCU Channel Read latency Register Format

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:32	0	RO	Reserved
LATENCY1	31:16	0xFFFF	RW	Read Latency For Channel 1. Determined during polling state.
LATENCY0	15:0	0xFFFF	RW	Read Latency For Channel 0. Determined during polling state.

10.7.3.6 MCU Sync Frame Frequency Register

This register is at location 0x84_0000_08B0. The format is as follows.

TABLE 10-15 MCU Sync Frame Frequency Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:6	0	RO	Reserved
FREQ	5:0	0x2A	RW	Frequency at which Sync frames are issued on the FBDIMM channels.

10.7.3.7 Subsystem Reset Register

The subsystem reset generation register, is provided to allow software to reset selected IO subsystems. This register is located at (0x89-0000-0838).

TABLE 10-16 Subsystem Reset Register

Field	Bit Position	Initial Value	R/W	Description
RSVD1	63:5	0	RO	Reserved

TABLE 10-16 Subsystem Reset Register *(Continued)*

RSVD0	3:2	0	RO	Reserved
DMU_LINK_ TRAIN	1	0	R/W	Set to one to have the DMU cause a link reset training sequence. Value is automatically cleared once the XIR is complete.
NIU	0	0	R/W	Set to one to generate a warm reset to the Ethernet subsystem, both ingress and egress. Value is automatically cleared once the WMR is complete.

10.7.3.8 I/O Quiesce Control Register

This register is used by SW to quiesce I/O to SII and NCU blocks in OpenSPARC T2 from NIU and PCI_EX blocks. It is located in debug.v module at location 0x86-0000-0008. The format of this register is shown in TABLE 10-18:

TABLE 10-17 I/O Quiesce Control Register Format

Field	Bit Position	POR Value	R/W	Description
RSVD	63:4	0x0	RO	Reserved
NIU_STALL_ DONE	3	X	RO	Status bit set to 1 when NIU stall complete. Cleared by hardware when NIU_STALL cleared from 1 to 0 by SW.
DMU_STALL_DON E	2	X	RO	Status bit set to 1 when DMU stall complete. Cleared by hardware when DMU_STALL cleared from 1 to 0 by SW.
NIU_STALL	1	0 Preserved across WMR/DBR	R/W	When set to 1, causes NIU traffic to stall . When cleared to 0 from 1, causes NIU traffic to resume.
DMU_STALL	0	0 Preserved across WMR/DBR	R/W	When set to 1, causes DMU traffic to stall. When cleared to 0 from 1, causes DMU traffic to resume.

10.7.3.9 Core DECR Register

All strands of a physical OpenSPARC T2 core share a hyperprivileged, read/write, Debug Event Control Register located at ASI 0x45, VA 0x8. The DECR controls the stop type (hard or soft) or a trigger pin for an associated event if that event occurs. The format of the Core DECR is described in the following table.

TABLE 10-18 ASI_DECR Format

Data Bits	Field name	Remarks
63:62	IWA_DE	Instruction breakpoint match debug event enable
61:60	IVA_DE	Instruction virtual address match debug event enable
59:58	DVA_DE	Data virtual address match debug event enable
57:56	DPA_DE	Data physical address match debug event enable
55:54	TCT_DE	Trap on Control Transfer debug event enable
53:52	PE_DE	Precise error event (an event which will be recorded in the I-SFSR or D-SFSR) debug event enable
51:50	DE_DE	Disrupting error event (an event which will be recorded in the DESR) debug event enable
49:48	DF_DE	Deferred error event (an event which will be recorded in the DFESR) debug event enable
47:46	PM_DE	Performance monitor event which causes a performance counter to wrap debug event enable
45:0	-	Reserved

Bits 63:62 control what type of stop occurs if an instruction watchpoint occurs on any strand. (Each strand has independent control over instruction breakpoints via it's HPSTATE.IBE register). Remaining bit pairs in the table similarly control their associated event.

There are two bits in the DECR for each event type. Each pair of bits in the DECR encode the type of stop for that event as follows.

TABLE 10-19 ASI_DECR bit-pair settings to achieve Debug

DECR event enable bit pair settings, bit i+1:i	Response if debug event occurs
00	Debug event disabled
01	Soft-stop
10	Hard-stop
11	Pulse trigger pin

10.7.3.10 *SOC DECR Register*

All SOC Debug events will share a read/write, SOC Debug Event Control Register located at address 0x86-0000-0010. The SOC DECR controls hard stop or a trigger pin assertion for an associated event if that event occurs. The format of the SOC DECR is described in the following table. This register will be physically located in the Debug block (debug.v).

TABLE 10-20 SOC_DECR Format

Data Bits	Field name	Remarks
63:22	-	Reserved
21:20	SE_DE	SOC Error (SII,SIO,NCU,DMU,PEU)Debug Event Enable
19:18	ME_DE	MCU Error Debug Event Enable
17:16	L2E_DE	L2 Error Debug Event Enable
15:14	L2B7_DE	L2 PA Match Bank 7 Debug Event Enable
13:12	L2B6_DE	L2 PA Match Bank 6 Debug Event Enable
11:10	L2B5_DE	L2 PA Match Bank 5 Debug Event Enable
9:8	L2B4_DE	L2 PA Match Bank 4 Debug Event Enable
7:6	L2B3_DE	L2 PA Match Bank 3 Debug Event Enable
5:4	L2B2_DE	L2 PA Match Bank 2 Debug Event Enable
3:2	L2B1_DE	L2 PA Match Bank 1 Debug Event Enable
1:0	L2B0_DE	L2 PA Match Bank 0 Debug Event Enable

Thus there are two bits in the SOC_DECR for each event type. Each pair of bits in the SOC_DECR encode the type of stop for that event as follows.

TABLE 10-21 ASI_DECR bit-pair settings to achieve Debug

DECR event enable bit pair settings, bit i+1:i	Response if debug event occurs
00	Debug event disabled
01	Debug event disabled
10	Hard-stop
11	Pulse trigger pin

10.7.3.11 *L2 Mask Register*

This register will be located at address 0xAF-0000-0000 within l2t.sv. The format is as shown in TABLE 10-23.

TABLE 10-22 L2 Mask reg Format

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:52	0	RO	Read as Zero
TTYPE[3:0]	51:48	Preserved	R/W	Transaction Type
RSVD1	47:46	0	RO	Read as Zero
VCID[5:0]	45:40	Preserved	R/W	Virtual Core ID.
RSVD2	39:34	0	RO	Read as Zero
ADDR[33:2]	33:2	Preserved	R/W	Corresponds to addr[33:2]
RSVD4	1:0	0	RO	Read as Zero

10.7.3.12 L2 Compare Register

This register will be located at 0xBF-0000-0000 within l2t.sv. The format is as shown inTABLE 10-24.

TABLE 10-23 L2 Compare Reg Format

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:52	0	RO	Read as Zero
TTYPE[3:0]	51:48	Preserved	R/W	Transaction Type
RSVD1	47:46	0	RO	Read as Zero
VCID[5:0]	45:40	Preserved	R/W	Virtual Core ID.
RSVD2	39:34	0	RO	Read as Zero
ADDR[33:2]	33:2	Preserved	R/W	Corresponds to addr[33:2]
RSVD4	1:0	0	RO	Read as Zero

10.7.3.13 DMC Core and Block Interrupt Enable Register

This register is at address (0x00631800 / 0x0). It will host the Debug_trig_en bit for DMU and PEU errors.

TABLE 10-24 DMC Core and Block Interrupt Enable register Format

Field	Bit Position	Initial Value	R/W	Description
DMC	63	0x0	R/W	The enable bit to enable all operations from the DMC which will cause an interrupt via mondo 62. 1 = Core Level interrupt is enabled, 0 Core Level interrupt is disabled
DEBUG_TRIG_EN	62	0x0	R/W	DEBUG_TRIG_EN for PCI_EX Errors .
Reserved	61:2	-	RO	Reserved
MMU	1	0x0	R/W	The enable bit to enable all operations from the MMU which will cause an interrupt via mondo 62. 1 = Block Level interrupt is enabled, 0 = Block Level interrupt is disabled.
IMU	0	0x0	R/W	The enable bit to enable all operations from the IMU which will cause an interrupt via mondo 62. 1 = Block Level interrupt is enabled, 0 = Block Level interrupt is disabled.

10.7.3.14 DRAM Debug Trigger Enable Register

Each DRAM controller has a register that contains the Debug_Trig_En for all the errors detected by that DRAM controller (Esc,mecc and fbdimm channel errors) .The register is located at address (0x97-0000-0230) in mcu.sv . The format of this register is as follows :

TABLE 10-25 DRAM Debug Trigger Enable Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:3	0x0	R/W	Reserved

TABLE 10-25 DRAM Debug Trigger Enable Register *(Continued)*

DEBUG_TRIG_EN	2	0x0	R/W	DEBUG_TRIG_EN for DRAM Controller Errors
MASK_ERR	1	0x0 on POR, preserved on WMR/DBR	R/W	If set to 1, MCU mask all the errors it normally detects on LFSR mismatches on ALERT frame patterns coming in from AMB .
KP_LNK_UP	0	0x0 on POR, preserved on WMR/DBR	R/W	When written to 1'b1 : (i) Keeps the Southbound Links enabled during the duration of the Debug reset to send out the sync pulses. (ii) selects the output of the sync pulse gen logic in the new MCU control module to generate sync pulses. When written to 1'b0 : (i) selects the output of the regular sync pulse gen logic in MCU (ii) clears the counter for the regular sync pulse gen logic in MCU. (iii) takes MCU fbdimm interface state machine to L0 state, where it is ready to dispatch new read/write requests to the DIMMs.

10.7.3.15 NCU Debug Trigger Enable Register

The NCU has a register to contain the Debug_Trig_en for all the SOC errors logged in SOC Error Status Register in NCU (ncu.sv). This register is located at address 0x80_0000_4000 . The format of this register is as follows :

TABLE 10-26 NCU Debug Trigger Enable Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:1	0x0	RO	Reserved
DEBUG_TRIG_EN	0	0x0	R/W	DEBUG_TRIG_EN for SOC Error Status Register Errors

10.7.3.16 L2 Error Enable Register

This register contains the DEBUG_TRIG_EN bit for L2 errors. In addition it also contains the trigger enable for PA & VCID match . It is located at address 0xAA-0000-0000 or 0xBA-0000-0000 in l2t.sv and the format is as follows :

TABLE 10-27 L2 Error Enable Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:3	X	RO	Reserved
DEBUG_TRIG_EN_ ERR	2	0	RW	DEBUG_TRIG_EN for L2 Errors
NCEEN	1	0	RW	If set to 1, report uncorrectable errors.
CEEN	0	0	RW	If set to 1, report correctable errors.

10.7.3.17 ASI_OVERLAP_MODE Register

All physical cores share a hyperprivileged ASI_OVERLAP_MODE register located at ASI 45, VA 0x10. The contents of the ASI_OVERLAP_MODE register are described below. Reserved bits read as all zeroes and are ignored on write. Bits 15:0 is set to '0' on POR.

TABLE 10-28 ASI_OVERLAP_MODE Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:16	0	RO	Reserved
OVLP_7	15:14	0	R/W	Overlap control for physical core 7 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step
OVLP_6	13:12	0	R/W	Overlap control for physical core 6 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step
OVLP_5	11:10	0	R/W	Overlap control for physical core 5 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step

TABLE 10-28 ASI_OVERLAP_MODE Register *(Continued)*

OVLP_4	9:8	0	R/W	Overlap control for physical core 4 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step
OVLP_3	7:6	0	R/W	Overlap control for physical core 3 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step
OVLP_2	5:4	0	R/W	Overlap control for physical core 2 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step
OVLP_1	3:2	0	R/W	Overlap control for physical core 1 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step
OVLP_0	1:0	0	R/W	Overlap control for physical core 0 as follows: 0x - Normal operation 10 - Disable overlap 11 - Single-step

10.7.3.18 PEU Debug Select A Register

The PEU debug select register selects the output on PEU debug bus A .

TABLE 10-29 PEU Debug Select A Register (0x000683000/0x0)

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:9	0x0	RO	Reserved
BLOCK	8:6	0x0	R/W	Block select in core 000b - Constant zero 001b - Training Sequence Selection 010b - ETL block 011b - ITL block 100b - PMC block 101b - RSB block 110b - CTB block 111b - CXPL core
MODULE	5:3	0x0	R/W	Module select in block
SIGNAL	2:0	0x0	R/W	Signal select in sub-block

10.7.3.19 PEU Debug Select B Register

The PEU debug select register selects the output on PEU debug bus B.

TABLE 10-30 PEU Debug Select B Register (0x000683008/0x0)

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:9	0x0	RO	Reserved
BLOCK	8:6	0x0	R/W	Block select in core 000b - Constant zero 001b - Training Sequence Selection 010b - ETL block 011b - ITL block 100b - PMC block 101b - RSB block 110b - CTB block 111b - CXPL core
MODULE	5:3	0x0	R/W	Module select in block
SIGNAL	2:0	0x0	R/W	Signal select in module

10.7.3.20 DMU Debug Select Register for DMU Debug Bus A

The DMU debug select register A selects the output on DMU debug bus A.

TABLE 10-31 DMU Debug Select A Register (0x000653000/0x0)

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:10	0x0	RO	Reserved
BLOCK	9:6	0x0	R/W	DMU Block Debug Selects for DMU Debug Bus A 0000 – All Zeroes 0001 – CLU Block Selects (cache Line Unit) 0010 – CMU Block Selects (Context Manager Unit) 0011 – CRU Block Selects (CSR Request Unit) 0100 – DSN Block Selects 0101 – Training Sequence Select 0110 – ILU Block Selects (Interface Layer Unit) 0111 – All Zeroes 1000 – All Zeroes 1001 – IMU Block Selects (Interrupt Messager Unit) 1010 – MMU Block Selects 1011 – PMU Block Selects 1100 – PSB Block Selects (Packet Scoreboard unit) 1101 – RMU Block Selects (Record Manager Unit) 1110 – TMU Block Selects (Transaction Manager Unit) 1111 – TSB Block Selects (Transaction Scoreboard Unit)
SUB_SEL	5:3	0x0	R/W	Select the sub-block for DMU Debug Bus A
SIGNAL_SEL	2:0	0x0	R/W	Select the signals for DMU Debug Bus A

10.7.3.21 DMU Debug Select Register for DMU Debug Bus B

The DMU debug select register selects the output on DMU debug bus B.

TABLE 10-32 DMU Debug Select B Register (0x000653008/0x0)

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:10	0x0	RO	Reserved
BLOCK	9:6	0x0	R/W	DMU Block Debug Selects for DMU Debug Bus B 0000 – All Zeroes 0001 – CLU Block Selects (cache Line Unit) 0010 – CMU Block Selects (Context Manager Unit) 0011 – CRU Block Selects (CSR Request Unit) 0100 – DSN Block Selects 0101 – Training Sequence Select 0110 – ILU Block Selects (Interface Layer Unit) 0111 – All Zeroes 1000 – All Zeroes 1001 – IMU Block Selects (Interrupt Messager Unit) 1010 – MMU Block Selects 1011 – PMU Block Selects 1100 – PSB Block Selects (Packet Scoreboard unit) 1101 – RMU Block Selects (Record Manager Unit) 1110 – TMU Block Selects (Transaction Manager Unit) 1111 – TSB Block Selects (Transaction Scoreboard Unit)
SUB_SEL	5:3	0x0	R/W	Select the sub-block for DMU Debug Bus B
SIGNAL_SEL	2:0	0x0	R/W	Select the signals for DMU Debug Bus B

CHAPTER 11

Electronic Fuse Unit (EFU)

This chapter contains the following sections:

11.1 Overview

The Efuse (electronic fuse) unit (EFU) contains an Efuse array macro (EFA), TCU interface and an Efuse controller(FCT). In a broad sense, the Efuse array is a non-volatile memory used to store information that needs to be programmed at the factory and used in the field.

On OpenSPARCT2, EFA contains the following die specific information :

Redundant array repair information for the SRAMs

Serial ID of the chip

Working processor core IDs (core available information)

Working L2 bank information (bank available information)

SERDES bits

DMU delay calibration

The Efuse array is a 64 deep and 32 bit wide array. Each cell in the Efuse array consists of poly fuses that replace traditional laser fuses. They can be programmed to store any value by blowing them with an electrical pulse. The Efuse controller has the logic to read and transfer data from EFA to on and off chip components. The TCU interface consists of logic to handle all the TCK clock domain generated signals.

The Efuse unit has only limited knowledge of ways to interpret the data stored in the array. Most of the time the payload data is just read and passed along with little interpretation. This document will attempt to describe some of the data uses, to aid users.

After the power on reset sequence, a state machine in FCT reads all the 64 entries of the EFA (one at a time). If a valid (refer table 2) SRAM repair row is found, it is shifted to the destination register. If no information is programmed into the EFA, no information will be shifted to the destination registers. Read access to the Efuse array is available at any point (other than during power up sequence) through TCU(TAP controller). EFA can be programmed and read via private TAP instructions. Moreover, after power on reset, data can be shifted through TCU to any destination register overriding either the default value of the destination register or the previously programmed value.

Main features of the Efuse unit:

The Efuse array is organized as 32 bits wide 64 entry array.

It supports a maximum of 59 SRAM repairs.

It stores 3 entries of chip ID information, 1 entry for core valid information, and 1 entry for L2 bank valid information.

It interfaces with TCU : TCU can program the EFA, read any entry in EFA, and configure EFU in bypass mode to overwrite the destination register.

It interfaces with NCU to provide serial ID, core and L2 bank available information.

It interfaces with L1 cache (instruction and data) and L2 cache (tag and data). EFU provides information to swap defective SRAM rows and columns with redundant spares.

Access (programming and reading) to EFA is supported at various stages :

1. Before bump : through JTAG with Vpp laser pad.
2. At wafer level : through JTAG with Vpp bump.
3. At package : through JTAG with Vpp pins.

Writing to destination registers is done only through the JTAG port. Software running on OpenSPARC T2 cannot program the destination registers (in SRAM or NCU) or access EFU.

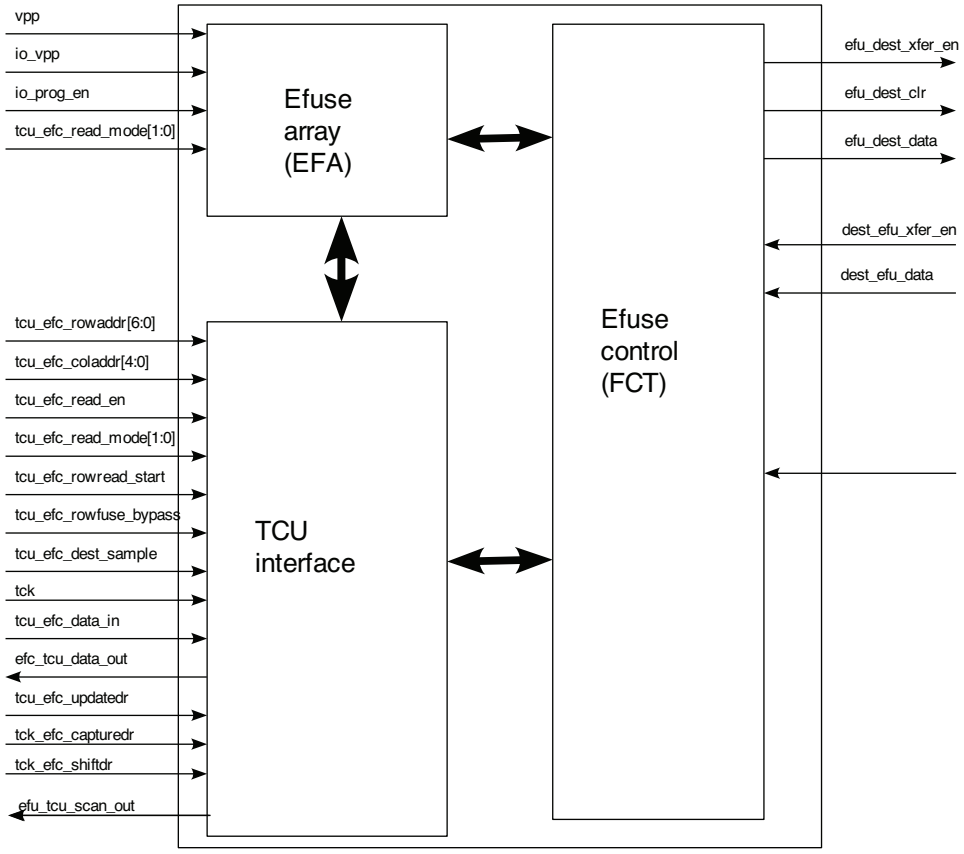
11.1.1 Definitions of Terms Used

TABLE 11-1 Terms

Abbreviations	Expansions
EFU	Efuse unit
EFA	Efuse array
FCT	Efuse controller
ICD	L1 Instruction cache
DCD	L1 Data cache
RID	Logical sub bank ID in the SRAM
RV	Repair value.

11.2 EFU Block Diagram

FIGURE 11-1 EFU Top Level Diagram



11.2.1 Unit Functional Description of EFU

1. The Efuse unit (EFU) contains :
2. Efuse array (EFA)

3. Efuse controller (FCT)
4. TCU interface

11.2.1.1 Efuse Array (EFA)

The Efuse array is a 64 deep and 32 bit wide array. Each cell in the Efuse array consists of poly-fuse that replace a traditional laser fuse. Each cell can be programmed to store any value by blowing the fuse with an electrical pulse.

Fuse data interpretation :

We assume that EFA consists of 64 rows with 32 bits each. Each entry consists of the following fields.

TABLE 11-2 Fields in the Efuse Array Data[31:0]

Bit position	Number of bits	Description of fields
31:29	3	Valid bits
28	1	Parity bit
27:22	6	Block ID of destination register
21:0	22	Data

The Efuse controller will use EFA bits[31:22]. EFA bits [21:0] will be interpreted and used by hardware in the cluster associated with the destination register.

Truth Table of EFA Programmed Data

# of valid bits at logic "1"	Computed parity = ^efa bits [28:0]	Row valid	Row error	Action on read	Interpretation
2 or 3	Logic 0	Y	N	Shift value to destination register	Programmed row with valid data
0	X	N	N	Ignore	A row which was not programmed correctly or not used
1	X	N	Y	Log error (NCU) and ignore data	Programmed row where 1 or 2 valid bits have flipped from their intended values.
2 or 3	Logic 1	N	Y	Log error (NCU) and ignore data	Programmed row where a EFA bit in position [27:0] has flipped from it's intended value.

Note:If any of bits[28:0] cannot be programmed successfully, the valid bits are left un-programmed and the desired data is programmed at another row address.

11.2.1.2 Efuse Controller (FCT)

An Efuse controller reads from the Efuse array and transfers data to on and/or off-chip components. The Efuse controller hosts the following main sub blocks :

1. Clock generator

IO clock (iol2clk) is distributed within the Efuse unit. A pair of two phase (fuse_clk1 and fuse_clk2), non-overlapping IO clock(nominally 375 MHz) divided by 4 signals are used to shift values into the destination registers. This block generates shift clocks for each destination register.

The shift clocks are active only when data is shifting to any destination register. Qualification with either ashift or dshift (refer to figure 1) is necessary to ensure that only the correct destination register is being addressed.

2. Address sequencer

The address sequencer is a 0 to 63 counter. It counts through all the entries of the Efuse array. Its initial state consists of all zeroes (reset by POR).

TCU asserts tcu_efu_read_start which starts the counter operation. The counter increments when the shift_done signal from the shift register is asserted. When a new address is generated, it asserts the new_addr signal to the shift register. Upon reaching the count of 63, the next shift_done signal results in asserting the addr_done signal to power down the EFA.

3. Shift register

This block contains a parallel in/serial out register. Loading of the shift register can happen from EFA or TCU (Depending on the mode EFU is configured in(refer section 4.1)). Unloading of the shift registers can be to any legal block ID (refer table 4) or TCU.

Reads from EFA are loaded in parallel to a shift register called read_data_ff[31:0]. The shift register block checks the valid bits and parity of the data (refer table 2). If the entry is valid, and no parity errors are detected, the shift register shifts bits [21:0] to the destination register specified by the block ID bits[27:22]. Signals ashift and dshift are asserted to qualify shifted data fields repair ID and repair value respectively. If a parity error or invalid row is encountered in a row “n”, the corresponding parity_status_reg[63:0] bit gets set. The shift states are still counted internally (no ashift or dshift signals are asserted) and the address sequencer increments. At the end of the sequence (after EFA entry 63's data has been determined invalid or shifted to the appropriate destination register), the error report (parity_status_reg[63:0]) is shifted to the NCU cluster. The parity status

register in the NCU enables software to figure out whether a parity error or row error occurred in EFA readout. Software can then decide whether to fail out or continue (if for instance the device ID is potentially incorrect). Additionally, the register can be loaded serially by TCU via tcu_efu_data_in. Control signals used by TCU to load are tcu_efu_shiftdr, tcu_efu_updatedr and tcu_efu_capturedr.

The serial output of the shift register is read_data_ff[31]. The serial output (read_data_ff[31]) is shifted to the destination register defined by block ID field bits[27:22] (refer table 4) as efu_<destination>_fuse_data or to TCU as efu_tcu_data_out.

Destination registers are organized as repair chains. Repair chains for the L2 caches are 22 bits long and enables are held high for 22 cycles when shifting data. Each SPARC cluster gets two (22 bit long) repair chains, one for I and the other for D cache.

11.2.1.3 TCU Interface

All the signals coming from the TCU clock are generated in TCK clock domain. The TCU interface of EFU synchronizes almost all incoming TCU signals to the iol2clk domain. (refer section 10 multi clock domains for details).

EFU places tcu_efu_data_in from TCU at bit 0 of the tck 32 bit register. As a result the first bit that is shifted in from TCU will end up at bit 31 of the tck 32 bit register. MSB of the data should be shifted in first.

When EFU shifts the data back to TCU it loads bit 31 on to efu_tcu_data_out. As a result bit 31 will go back first followed by bit 30, bit 29, and so on. MSB of the readback data is shifted out first.

Following table lists all the commands, which are used by TCU to program EFU behavior.

TABLE 11-3 TAP Private Instructions for Fuse Functionality

Command	Encoding	Functionality
TAP_FUSE_READ	6h'28	Issue Read Command and shift out the result to destination registers
TAP_FUSE_BYPASS_DATA	6h'29	Issue Bypass command and shift in 32 bit value from TCU
TAP_FUSE_BYPASS	6'h2a	Command initiates shifting of data to receiver from FCT block
TAP_FUSE_ROW_ADDR	6h'2b	Shift in 7 bit Row Address for EFA access

TABLE 11-3 TAP Private Instructions for Fuse Functionality *(Continued)*

TAP_FUSE_COL_ADDR	6h'2c	Shift in 5 bit Column address (only for programming) for EFA access
TAP_FUSE_READ_MODE	6h'2d	Shift in 2 bit Read Mode for EFA access
TAP_FUSE_DEST_SAMPLE	6'h2e	Tell efu to get the data and return it to tcu

For more instruction details, please refer to TCU specification.

11.3 EFU Logical Implementation

11.3.1 Efuse Modes of Operations

TCU can configure Efuse in 5 different modes. The following are the various modes of operation of EFU :

11.3.1.1 Power On Reset Read Mode

In this mode, all the valid entries in EFA are shifted to the destination register.

At some point after POR_, TCU signals EFU to start shifting all valid entries in the EFA. The sequence of events are :

TCU asserts tcu_efu_read_start valid for one TCK clock cycle. tcu_efu_read_start is synchronized to iol2clk as local_read_start (refer figure 1 group A).

local_read_start triggers a counter addr_cnt_ff[5:0]. This counter is used to compute row address for reading EFA. fct_efa_read_en is asserted for a predetermined number of clocks to read an entry in EFA. The EFA read data efa_fct_data[31:0] is then parallel loaded to a shift register read_data_ff[31:0] (refer figure 1 group B).

EFU determines if the row is valid and error free (refer table 2). If an error is encountered then the corresponding bit is set in the rslt_status_ff[63:0].

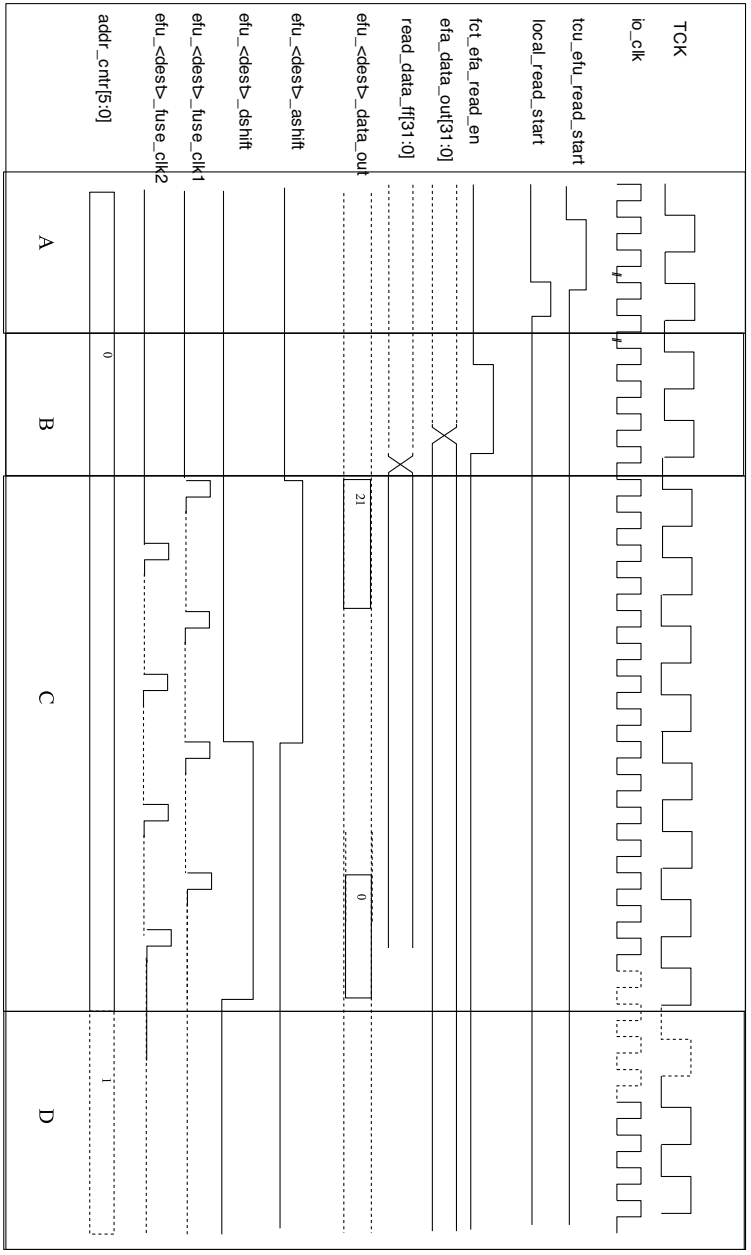
For a valid row, EFU interprets block ID (read_data_ff[27:22]) to determine the destination register for the row. EFU asserts a pair of non overlapping clocks (iol2clk/4 clocks) efu_<dest>_fuse_clk1 and efu_<dest>_fuse_clk2 for the duration of transfer. Only the read_data_ff[21:0] is shifted to the destination register. read_data_ff[21:0] consists of RID and RV information. Higher order bits are shifted first. efu_<dest>_fuse_ashift is asserted and RID information (bits[21:12]) and wren

are shifted to the destination register. A unit that doesn't use the higher order bits allow (unwanted) data to overflow. The RV (bits read_data_ff[11:0]) are shifted to the destination register, by asserting efu_<dest>_fuse_dshift. (refer FIGURE 11-2 group C)

Upon completion of processing a row, addr_cnt_ff[5:0] is incremented (refer FIGURE 11-2 group D). This process is repeated until the last row is processed.

EFU will shift out the rslt_status_ff[63:0] to NCU. Software will interpret this information and decide to failout or continue. (refer to NCU interface protocol in Section 11.3.2.3, “EFU to NCU Interface :” on page 11-19.

FIGURE 11-2 Timing Diagram showing Power On Reset Read Mode



11.3.1.2 JTAG Read Access

In this mode, a row in EFA can be read via the TAP controller in TCU. To read an EFA row, read mode (tcu_efa_read_mode[1:0]), row address (tcu_efu_rowaddr[6:0]), and read enable are required. The sequence of JTAG instructions to program the TAP controller to read the fuse array are as follows :

1. TAP_FUSE_READ_MODE
2. TAP_FUSE_ROW_ADDR
3. TAP_FUSE_READ

TAP_FUSE_READ_MODE instruction is programmed to the TAP controller through the JTAG port (this instruction configures EFU to read EFA in a particular mode). TCU will decode the instruction and drive a 2 bit encoded tcu_efa_read_mode[1:0] signal to EFU. tcu_efa_read_mode[1:0] gets registered and is driven to EFA as fct_efa_margin0_rd and fct_efa_margin1_rd respectively. (refer figure 2 group A) Bit 2 of tcu_efa_read_mode bus is a power down enable mode bit. When the internal state machine finishes all the transfers and there is no pending transfer the EFU state machine will activate the power-down signal to EFA. When bit 2 is low EFA is in the normal mode. When it is high EFA will power down after the current operation finishes.

EFA can be read in 4 different modes. EFA decodes fct_efa_margin0_rd and fct_efa_margin1_rd as 00=normal mode, 01=margin0 mode, 10=margin1A mode and 11=margin1B and configures it's sense amplifier circuit. In different modes, EFA sense amplifiers are supplied with different reference voltages to detect logic 1 and logic 0.

TAP_FUSE_ROW_ADDR instruction provides EFA with a read address. TCU provides EFU with the row address as tcu_efu_rowaddr[6:0]. tcu_efu_rowaddr[6:0] generated in TCK domain is synchronized to the iol2clk domain and driven to EFA.(refer figure 2 group B) Bit[6] (when it is high; the other bits will be ignored) of the tcu_efu_rowaddr[6:0] is to read back the stage of the power supply. The format of the read is as follow from the efa fuse:

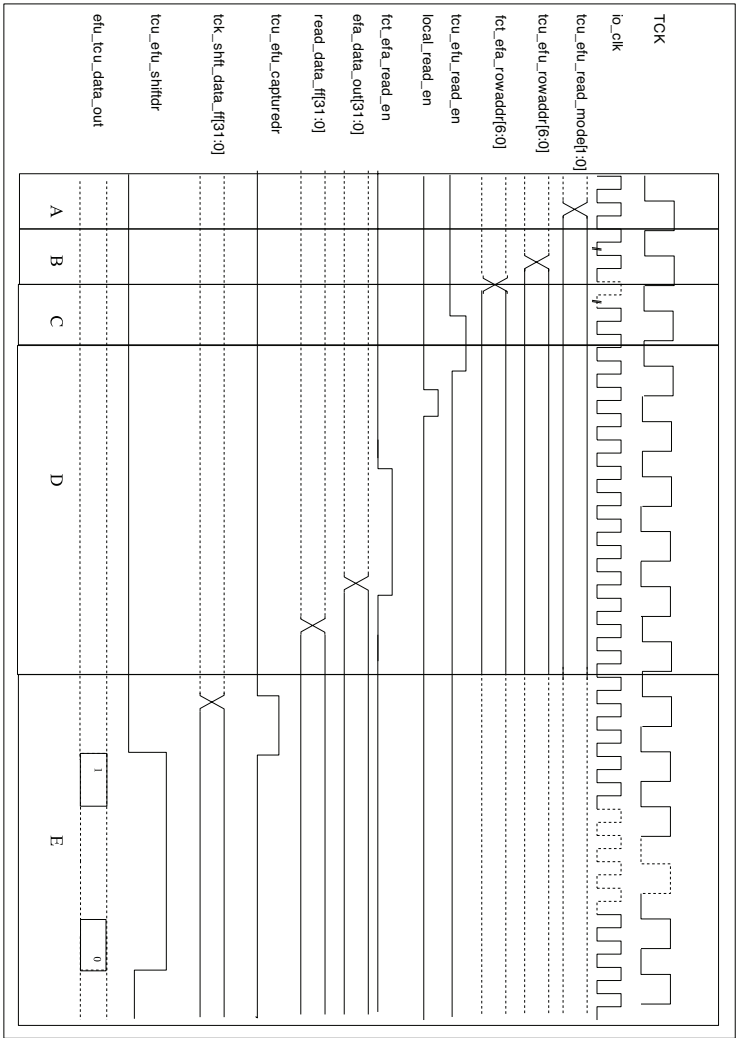
efa_fuseout[31:0] = {29'b0,vddc_ok,vddo_ok,vpp_ok

TAP_FUSE_READ instruction requests a read to be performed. TCU decodes this instruction and generates a one cycle tcu_efu_read_en pulse. tcu_efu_read_en is synchronized into iol2clk domain as local_read_en. (refer FIGURE 11-3 group C)

The logic in EFU will load a counter with read latency and assert fct_efa_read_en. After the counter is counted down to zero, the EFA output is parallel loaded into a shift register read_data_ff[31:0]. (refer FIGURE 11-3 group D)

TCU will wait for a predetermined period (EFA read is a multicycle operation; the current wait time is 30 tck cycles) and issue one TCK cycle valid tcu_efu_capturedr pulse. Shift register read_data_ff[31:0] contents are loaded to another shift register called tck_shft_data_ff[31:0] (read_data_ff shift register is in iol2clk domain and tck_shft_data_ff is in TCK clock domain). TCU asserts tcu_efu_shiftdr for 32 clocks causing the shift of tcu_shft_data_ff[31] onto efu_tcu_data_out. (refer FIGURE 11-3 group E) The readback data shifted out first is bit 31 to tcu, followed by the subsequent lower significant bits

FIGURE 11-3 JTAG Read Access Timing Diagram.



11.3.1.3 Fuse Programming Mode

In this mode, EFA is programmed one bit at a time (EFA does not support multiple bit programming). To program a bit, row address[6:0], column address[4:0] and fct_efa_prog_en should be valid. In order to program a fuse bit, actions required are :

1. TAP_FUSE_ROW_ADDR
2. TAP_FUSE_COL_ADDR

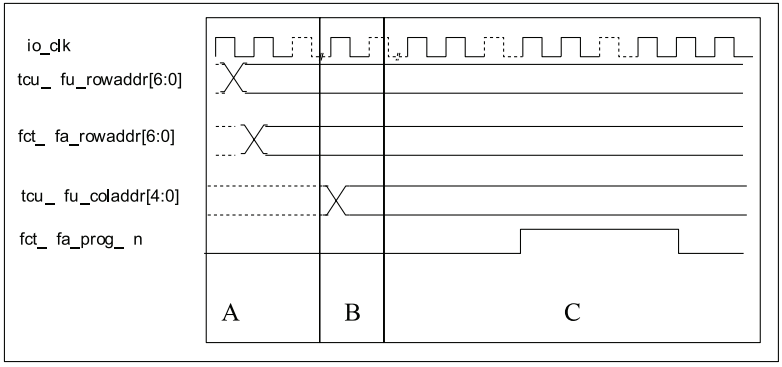
TAP_FUSE_ROW_ADDR instruction gets issued to the TAP controller through the JTAG port. TCU decodes and supplies the row address as efu_tcu_rowaddr[6:0]. The row address is synchronized from TCK to iol2clk domain as fct_efa_row_addr[6:0]. (refer FIGURE 11-5 group A)

TAP_FUSE_COL_ADDR instruction gets issued to the TAP controller through the JTAG port. TCU decodes and supplies the column address as tcu_efa_coladdr[6:0]. (refer FIGURE 11-5 group B).

After the row and column address is supplied to EFA, fct_efa_prog_en is asserted for as long as deemed necessary by TI. The fuse value is then read back (refer section 4.1.2) to ensure that the bit was programmed correctly. (refer FIGURE 11-5 group C).

fct_efa_prog_en chip pin must be available to all test environments, so we need a probe pad, C4 and a package pin. From the top level, the of fct_efa_prog_en signal is fed directly to the row and column decoders as well as the supply enable

FIGURE 11-4 Fuse Programming Mode Timing Diagram.



11.3.1.4 JTAG Fuse Bypass Mode

Fuse bypass mode is to enable bring up if there is a problem in the Efuse functionality. In this mode, EFA is bypassed. The sequence of operations to program the TAP controller to bypass the fuse array are as follows :

1. TAP_FUSE_BYPASS_DATA
2. TAP_FUSE_BYPASS

TAP_FUSE_BYPASS_DATA instruction gets issued to the TAP controller through the JTAG port. TCU decodes and programs a shift register tck_shft_data_ff[31:0] serially with efu_tcu_data_in by asserting tcu_efu_shiftdr for 32 clocks. tcu_efu_updatedr,

valid for one TCK clock will parallel load tck_shft_data_ff[31:0] to read_data_ff[31:0]. As mention previously tcu_efu_data_in is placed at bit 0 of the 32 bit tck register, tck_shft_data_ff[31:0]. MSB of the data should be shifted in first.

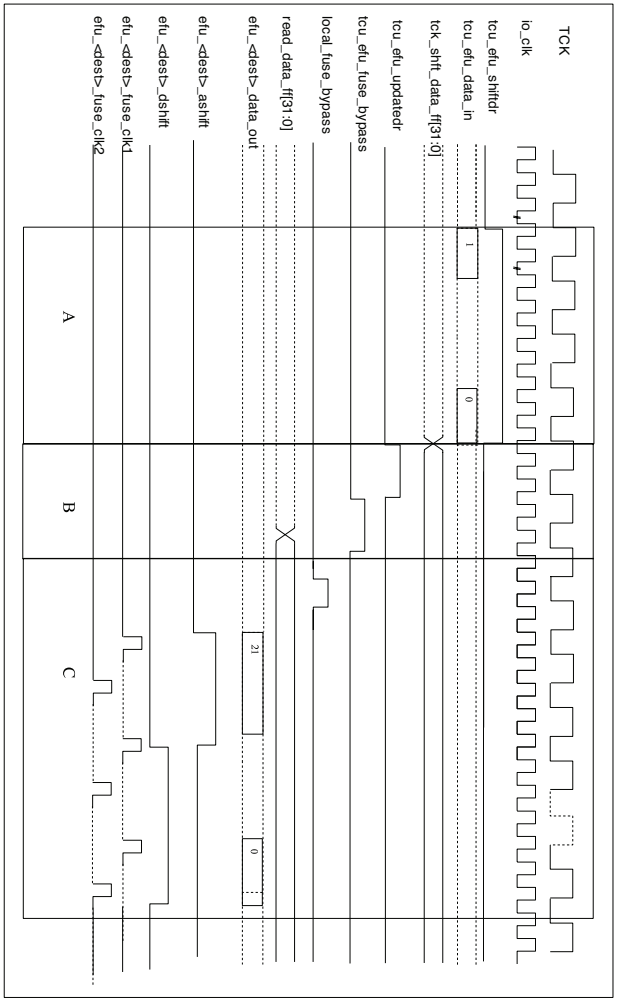
TAP_FUSE_BYPASS instruction gets issued to the TAP controller through the JTAG port. TCU decodes and asserts tcu_efu_fuse_bypass valid for one TCK clock. tcu_efu_fuse_bypass is synchronized into iol2clk domain as local_fuse_bypass.

local_fuse_bypass triggers the shift of the contents of shift register (read_data_ff) to destination register. EFU decodes the block ID (read_data_ff[27:22]) to determine the destination register and determines if the row is valid and error free (refer TABLE 11-3).

For a valid row, EFU interprets block ID (read_data_ff[27:22]) to determine the destination register for the row. It asserts a pair of non overlapping clocks (iol2clk/4 clocks) efu_<dest>_fuse_clk1 and efu_<dest>_fuse_clk2 for the duration of transfer. Only the read_data_ff[21:0] is shifted to the destination register. Higher order bits are shifted first.

efu_<dest>_fuse_ashift is asserted and RID information (bits[21:12]) and wren are shifted to the destination register. A unit that doesn't use the higher order bits allow (unwanted) data to overflow. The RV (bits read_data_ff[11:0]) are shifted to the destination register, by asserting efu_<dest>_fuse_dshift. (refer FIGURE 11-6 group C)

FIGURE 11-5 JTAG Fuse Bypass Mode



11.3.1.5 Fuse Sample Mode

In this mode the destination redundancy value (RV) is read and transferred to TCU. In order to read the destination register the following commands need to be executed.

1. TAP_EFU_BYPASS_DATA
2. TAP_EFU_DEST_SAMPLE
3. TAP_CAPTUREDR

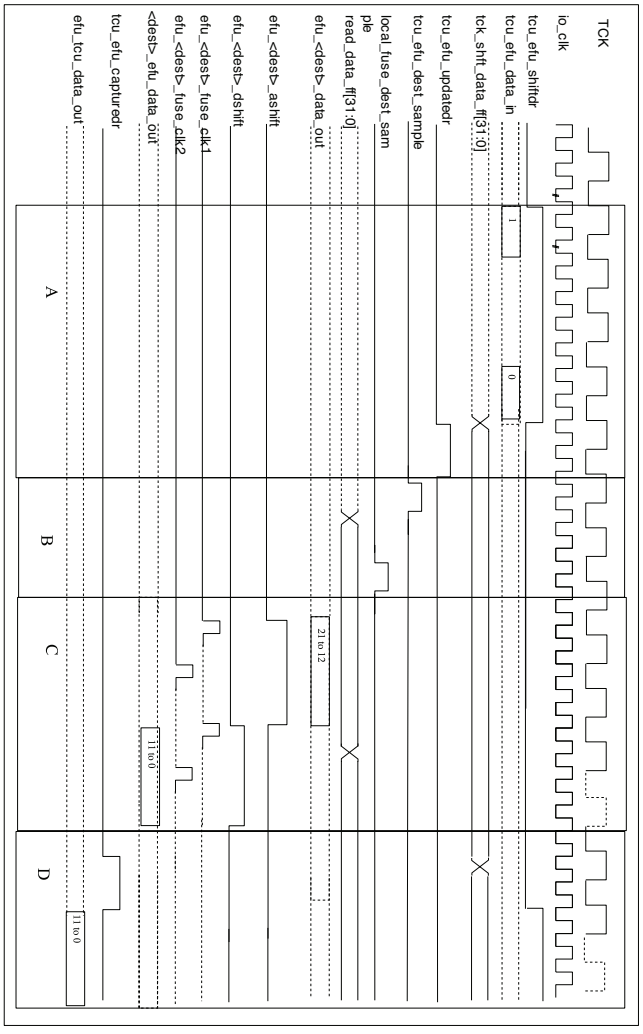
4. TAP_SHIFTDR

TAP_FUSE_BYPASS_DATA instruction gets issued to the TAP controller through the JTAG port. TCU decodes and programs a shift register tck_shft_data_ff[31:0] serially with efu_tcu_data_in by asserting tcu_efu_shiftdr for 32 clocks. TCU asserts tcu_efu_updatedr valid for one clock. tcu_efu_updatedr parallel loads tck_shft_data_ff[31:0] to read_data_ff[31:0].

When TAP_EFU_DEST_SAMP is issued read_data_ff shift register bits[21:0] where bit[21] is the read_en, are shifted to the destination register. efu_<dest>_xfer_en is asserted for the duration of transfer. The redundancy registers are organized as chains. During the efu_<dest>_fuse_xfer_en, the data is collected and forwarded to the SRAM header after all the data has been shifted in. <dest>_efu_fuse_xfer_en is then asserted to read the correct data into read_data_ff[31:0] shift register.

TCU will wait for a predetermined period and issue one TCK cycle valid tcu_efu_capturedr. Shift register read_data_ff[31:0] contents are parallel loaded into tck_shft_data_ff[31:0] by the valid tcu_efu_capturedr. tcu_efu_shiftdr asserted by TCU for 32 clocks shifts tcu_shft_data_ff[31] onto efu_tcu_data_out (refer FIGURE 11-6 group D). Bit 31 of the tck 32 bit register, tck_shft_data_ff[31:0], will be shifted out first.

FIGURE 11-6 Destination Sample Mode Timing Diagram



11.3.2 Interface with NCU, SRAM Header Flops and TCU Destinations

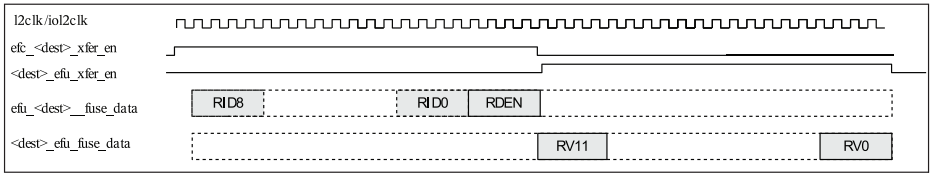
11.3.2.1 EFU to SRAM Header Flops

Data from EFU is transferred serially to SRAM destination header from read_data_ff[31:0]. EFU asserts efu_<dest>_xfer_en for the duration of transfer. MSB is shifted first as efu_<dest>_fuse_data. See the timing diagram below.

11.3.2.2 SRAM to EFU Interface :

The redundancy registers are organized as chains. <dest>_efu_xfer_en is asserted and held valid to read the correct data. The valid clocks correspond to the the appropriate SRAMs with the correct sync_en signals.

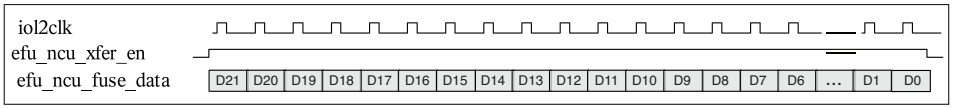
FIGURE 11-7 SRAM to EFU Data Transfer Timing Diagram



11.3.2.3 EFU to NCU Interface :

EFU transfers serial number, core available, bank available information, and fuse state information (rslt_shft_ff[63:0]) to NCU. The protocol for transfer is similar to SRAM header. EFU asserts efu_ncu_<info>_dshift where <info> indicates serial number, core available, bank available information, and fuse state information. efu_ncu_fuse_clk1 is active for the duration of transfer. efu_ncu_fuse_data is transferred.

FIGURE 11-8 EFU to NCU Interface Timing Diagram



11.3.2.4 TCU to EFU Transfers

The protocol for transfering data from TCU to EFU is as follows :

TCU asserts tcu_efu_shiftdr in order to initiate a transfer and keeps it asserted for the duration of the transfer. EFU configures a shift register tck_shft_data_reg[31:0] to accept data from TCU. This shift register receives the data in TCK clock domain. TCU then asserts tcu_efu_updatedr. EFU transfers the data from tck_shift_data_reg[31:0] in tck domain to read_data_ff in iol2clk domain.

11.3.2.5 EFU to TCU :

TCU asserts tcu_efu_capturedr. EFU transfers the contents from read_data_ff in iol2clk domain to tck_shift_data_reg[31:0] in TCK domain. TCU asserts tcu_efu_shiftdr. EFU shifts the data out from tck_shift_data_reg as efu_tcu_data_out (tck_shift_data_reg[31]).

11.3.3 Register Formats

11.3.3.1 RV REGISTER CLEAR ID

The following are the 7 bit rv register clear ID. When bit 7 is high the clear function is enable. When bit 7 is low the clear function is disable. When all 7 bits are high all rv clear signals are active.

TABLE 11-4 6 Bit Block ID for Memories

DestinationID	Clear ID	Description
Core0 I$	1000000	Clear all bits in the rv registers
Core0 D$	1000001	Clear all bits in the rv registers
Core1 I$	1000010	Clear all bits in the rv registers

TABLE 11-4 6 Bit Block ID for Memories *(Continued)*

Core1 D$	1000011	Clear all bits in the rv registers
Core2 I$	1000100	Clear all bits in the rv registers
Core2 D$	1000101	Clear all bits in the rv registers
Core3 I$	1000110	Clear all bits in the rv registers
Core3 D$	1000111	Clear all bits in the rv registers
Core4 I$	1001000	Clear all bits in the rv registers
Core4 D$	1001001	Clear all bits in the rv registers
Core5 I$	1001010	Clear all bits in the rv registers
Core5 D$	1001011	Clear all bits in the rv registers
Core6 I$	1001100	Clear all bits in the rv registers
Core6 D$	1001101	Clear all bits in the rv registers
Core7 I$	1001110	Clear all bits in the rv registers
Core7 D$	1001111	Clear all bits in the rv registers
l2t0	1010000	Clear all bits in the rv registers
l2t1	1010001	Clear all bits in the rv registers
l2t2	1010010	Clear all bits in the rv registers
l2t3	1010011	Clear all bits in the rv registers
l2t4	1010100	Clear all bits in the rv registers
l2t5	1010101	Clear all bits in the rv registers
l2t6	1010110	Clear all bits in the rv registers
l2t7	1010111	Clear all bits in the rv registers
l2d0	1011000	Clear all bits in the rv registers
l2d1	1011001	Clear all bits in the rv registers
l2d2	1011010	Clear all bits in the rv registers
l2d3	1011011	Clear all bits in the rv registers
l2d4	1011100	Clear all bits in the rv registers
l2d5	1011101	Clear all bits in the rv registers
l2d6	1011110	Clear all bits in the rv registers
l2d7	1011111	Clear all bits in the rv registers
niu_4k_clr	1100000	Clear all bits in the rv registers (RTX VLAN)
niu_ram_clr	1100001	Clear all bits in the rv registers (TDS TDMC)

TABLE 11-4 6 Bit Block ID for Memories *(Continued)*

niu_ram0_clr	1100010	Clear all bits in the rv registers (RDP RDMC0)
niu_ram1_clr	1100011	Clear all bits in the rv registers (RDP RDMC1)
niu_cfifo1_clr	1100100	Clear all bits in the rv registers (RTX ZCP1)
niu_cfifo0_clr	1100101	Clear all bits in the rv registers (RTX ZCP0)
niu_mac1_sf_clr	1100110	Clear all bits in the rv registers (RTX TXE1)
niu_mac1_ro_clr	1100111	Clear all bits in the rv registers (RTX TXE1)
niu_mac0_sf_clr	1101000	Clear all bits in the rv registers (RTX TXE0)
niu_mac0_ro_clr	1101001	Clear all bits in the rv registers (RTX TXE0)
niu_ipp1_clr	1101010	Clear all bits in the rv registers (RTX IPP1)
niu_ipp0_clr	1101011	Clear all bits in the rv registers (RTX IPP0)
dmu_clr	1101100	Set the bits to 4'b0010
mcu_fclrz	1110000	Clear all bits in the rv registers
psr_fclrz	1110001	Clear all bits in the rv registers
niu_fclrz	1110010	Clear all bits in the rv registers
All rv clear active	1111111	Clear all bits in the rv registers

11.3.3.2 Block ID

The following are the 6 bit block ID's for destination :

TABLE 11-5 6 Bit Block ID for Memories

#	Destination ID	Block ID	Description
1	Core0 I$	000000	Sparc core 0 Icache repair information
2	Core0 D$	000001	Sparc core 0 Dcache repair information
3	Core1 I$	000010	Sparc core 1 Icache repair information
4	Core1 D$	000011	Sparc core 1 Dcache repair information
5	Core2 I$	000100	Sparc core 2 Icache repair information
6	Core2 D$	000101	Sparc core 2 Dcache repair information
7	Core3 I$	000110	Sparc core 3 Icache repair information
8	Core3 D$	000111	Sparc core 3 Dcache repair information
9	Core4 I$	001000	Sparc core 4 Icache repair information
10	Core4 D$	001001	Sparc core 4 Dcache repair information

TABLE 11-5 6 Bit Block ID for Memories *(Continued)*

11	Core5 I$	001010	Sparc core 5 Icache repair information
12	Core5 D$	001011	Sparc core 5 Dcache repair information
13	Core6 I$	001100	Sparc core 6 Icache repair information
14	Core6 D$	001101	Sparc core 6 Dcache repair information
15	Core7 I$	001110	Sparc core 7 Icache repair information
16	Core7 D$	001111	Sparc core 7 Dcache repair information
17	l2t0	010000	L2 bank 0 tag array repair information
18	l2t1	010001	L2 bank 1 tag array repair information
19	l2t2	010010	L2 bank 2 tag array repair information
20	l2t3	010011	L2 bank 3 tag array repair information
21	l2t4	010100	L2 bank 4 tag array repair information
22	l2t5	010101	L2 bank 5 tag array repair information
23	l2t6	010110	L2 bank 6 tag array repair information
24	l2t7	010111	L2 bank 7 tag array repair information
25	l2b0	011000	L2 bank 0 data array repair information
26	l2b1	011001	L2 bank 1 data array repair information
27	l2b2	011010	L2 bank 2 data array repair information
28	l2b3	011011	L2 bank 3 data array repair information
29	l2b4	011100	L2 bank 4 data array repair information
30	l2b5	011101	L2 bank 5 data array repair information
31	l2b6	011110	L2 bank 6 data array repair information
32	l2b7	011111	L2 bank 7 data array repair information
33	coreavail	100000	NCU Sparc Core available
34	L2 bank avail	100001	NCU L2 bank available
35	sernum0	100010	NCU Serial number row0
36	Sernum1	100011	NCU Serial number row1
37	Sernum2	100100	NCU Serial number row2
45	DMU	101100	DMU delay calibration
46		101101	
47		101110	
48		101111	

TABLE 11-5 6 Bit Block ID for Memories *(Continued)*

49		110000	
50		110001	
51		110010	
52		110011	
53		110100	
54		110101	
55		110110	
56		110111	
57		111000	
58		111001	
59		111010	
60		111011	
61		111100	
62		111101	
63		111110	
64		111111	

11.3.3.3 SRAM Redundancy Register Formats :

There are 4 different storage formats in Efuse for SRAM. They are :

- L2 data array
- L2 tag array
- L1 data array
- L1 tag array
- Core Available
- L2 bank available
- SERDES
- DMU data
- SERNUM0, SERNUM1, SERNUM2

The Efuse unit will read the EFA and interpret bits [31:22] and shift out bits [21:0] into the cluster containing the destination register. Not all of the bits of the RID and RV will be used for all arrays.

11.3.3.4 L2 Data Array EFA Entry Definition

For the L2 Data array, the EFA entry is stored in the following format :

TABLE 11-6 L2 Data Array Entry Description

Bits	Size	Description
[21]	1	DO NOT BLOW THIS BIT Read enable: 1-read, 0-write (used in the bypass mode; must be 0 in the fuse)
[20:18]	3	Unused
[17:11]	7	RID[6:5] Selects one of the four quads RID[4:3] Selects one of the four 32KB in the quad RID[2:0] Selects one of 8 registers in the 32KB.
[11]	1	E1 (Enable1 -Both Enable1 and Enable0 must be asserted or the repair value is ignored)
[10:9]	2	Unused RV (These bits are shifted out of the EFU and off the end of the redundancy register)
[8:1]	8	RV (Repair Value – the row(needs all the 8bits) or column(needs only 6bits) to be repaired)
[0]	1	E0 (Enable0- Both Enable1 and Enable 0 must be asserted or the repair value is ignored)

TABLE 11-7 Readback

Bits	Size	Description
[10]	1	Unused
[9:8]	2	Valid: always 2'b11 on the readback data
[7:0]	8	RV data

0.5.4 L2 Tag Array EFA Entry Definition

For the L2 Tag array, the RID/RV fields are defined as follows for row repairs:

TABLE 11-8 L2 Tag Array RID/RV Field dDescription

Bits	Size	Description
[21]	1	DONOT BLOW THIS BIT Read enable: 1-read, 0-write (used in the bypass mode; must be 0 in the fuse)
[20:15]	5	Unused
[14:11]	4	RID[3:0] (Logical subbank ID. Values 0-15 are valid.)
[11]	1	E1(Enable1- Both Enable1 and Enable 0 must be asserted or the repair value is ignored.)

TABLE 11-8 L2 Tag Array RID/RV Field dDescription

[10:6]	5	Unused RV
[5:1]	5	RV (Repair Value—The row/column to be repaired)
[0]	1	E0 (Enable0- Both Enable1 and Enable 0 must be asserted or the repair value is ignored.)

For the L2 Tag array, the RID/RV fields are defined as follows for column repairs :

TABLE 11-9 L2 Tag Array RID/RV Field Description

Bits	Size	Description
[21]	1	DO NOT BLOW THIS BIT Read enable: 1-read, 0-write (used in the bypass mode; must be 0 in the fuse)
[20:15]	5	Unused
[14:11]	4	RID[3:0] (Logical sub bank ID. Values 0-15 are valid.)
[11]	1	E1(Enable1- Both Enable1 and Enable 0 must be asserted or the repair value is ignored.)
[10:6]	5	Unused RV
[5:1]	5	RV (Repair Value—The row/column to be repaired)
[0]	1	E0 (Enable0- Both Enable1 and Enable 0 must be asserted or the repair value is ignored.)

TABLE 11-10 Readback

Bits	Size	Description
[10:6]	5	Unused
[5:1]	5	RV value
[0]	1	Valid: 1'b1 always on the readback unless there is a problem

11.3.3.5 L1 INSTRUCTION CACHE (ICD) EFA Entry Definition

For the L1 ICD, the RID/RV fields are defined as follows for column repairs:

TABLE 11-11 L1 ICD RID/RV Field Descriptions

Bits	Size	Description
[21]	1	DO NOT BLOW THIS BIT Read enable: 1-read, 0-write (used in the bypass mode; must be 0 in the fuse)
[20:15]	6	Unused
[14:11]	4	RID select value
[11]	1	E1 (Enable1- Both Enable1 and Enable 0 must be asserted or the repair value is ignored.)
[10:6]	5	Unused RV
[5:1]	5	RV (Repair Value—The row to be repaired): 5 bits of rv and 1 bit of row/column repair select
[0]	1	Enable: tie it to both enable pins of the SRAM

11.3.3.6 L1 data cache array redundancy register (DCD) definition

For the L1 DCD, the RID/RV fields are defined as follows for column repairs:

TABLE 11-12 L1 DCD RID/RV Field Descriptions for Column Repair

Bits	Size	Description
[21]	1	DO NOT BLOW THIS BIT Read enable: 1-read, 0-write (used in the bypass mode; must be 0 in the fuse)
[20:13]	8	Unused
[12:11]	2	RID[1:0] (register select. Values 0-3 are valid)
[11]	1	E1(Enable1- Both Enable1 and Enable 0 must be asserted or the repair value is ignored)
[10:7]	4	Unused RV
[6:1]	6	RV (Repair Value—The column to be repaired)
[0]	1	E0 (Enable0- Both Enable1 and Enable 0 must be asserted or the repair value is ignored)

11.3.3.7 Core Available

TABLE 11-13 Core Available

Bits	Size	Description
[21:8]	16	Reserved (not used)
[7:0]	8	Core available : 1 = core available; 0 = core not available (NCU initializes its core-available register to all 1's. Upon the completion of the EFUSE dump, the register will pick up the value of this fuse.)

11.3.3.8 L2 Bank Available

TABLE 11-14 L2 Bank Available

Bits	Size	Description
[21:8]	16	Reserved (not used)
[7:0]	8	L2 bank available : 1 = L2 bank available; 0 = L2 bank not available (NCU initializes its bank-available register to all 1's. Upon the completion of the EFUSE dump, the register will pick up the value of this fuse.)

11.3.3.9 FSR SERDES Trimming Registers

Each time the data is written in the internal data is sent back to EFU from the output of the last chains. There is no direct readback from the serdes registers. RTRIM[0] is closest to fdo. Any reprogramming of the FSR SERDES macros also requires three EFA rows. The three row addresses are also important:

Address Row0 EFA.ROW[6:0] is arbitrary

Address Row1 EFA.ROW[6:0] > Row0 EFA.ROW[6:0]

Address Row2 EFA.ROW[6:0] > Row1 EFA.ROW[6:0]

```
Start: efu_mcu_fdi (out from EFU)
fsr_left.fsr0_b8_1.FDI
fsr_left.fsr0_a8.FDI
fsr_left.fsr0_b8_0.FDI
fsr_left.fsr1_b8_1.FDI
fsr_left.fsr1_a8.FDI
fsr_left.fsr1_b8_0.FDI
fsr_left.fsr2_b8_1.FDI
fsr_left.fsr2_a8.FDI
fsr_left.fsr2_b8_0.FDI
fsr_left.fsr3_b8_1.FDI
```

```
fsr_left.fsr3_a8.FDI
fsr_left.fsr3_b8_0.FDI
fsr_right.fsr4_b8_1.FDI
fsr_right.fsr4_a8.FDI
fsr_right.fsr4_b8_0.FDI
fsr_right.fsr5_b8_1.FDI
fsr_right.fsr5_a8.FDI
fsr_right.fsr5_b8_0.FDI
fsr_right.fsr6_b8_1.FDI
fsr_right.fsr6_a8.FDI
fsr_right.fsr6_b8_0.FDI
fsr_bottom.fsr7_b8_1.FDI
fsr_bottom.fsr7_a8.FDI
fsr_bottom.fsr7_b8_0.FDI
End: mcu_efu_fdo (back to efu)
>>-Aaron
```

11.3.3.10 DMU DATA Registers

TABLE 11-15 DMU WRITE DATA FORMAT

Bits	Size	Description
[21]	1	read_enable: 1-read, 0-write (used in the bypass mode; must be 0 in the fuse)
[20:12]	9	Reserved (not used)
[11]	1	VALID bit; must be 1'b1 to write fuse data
[10:5]	6	Reserved (not used)
[4:1]	4	Fuse data, where bit 4 is the bit identified pt 3 in the iommu spec, and bit 2 is the default bit to be on, this vector being one-hot note: bits[4:1] correspond to fuse[3:0] and bits[2] == fuse[1] (default bit to be on)
[0]	1	VALID bit; must be 1'b1 to write fuse data

TABLE 11-16 DMU READ DATA FORMAT

Bits	Size	Description
[21:4]	18	ignored
[3:0]	4	Readback fuse data; when clear the data is 4'b0010

11.3.3.11 SER_NUM Programming

SERNUM Format (SER_NUM reg)

SER_NUM[63:60] => DeltaVdd

SER_NUM[59:50] => DeltaT

SER_NUM[49] => TESTINFO-RESERVED

SER_NUM[48:46] => Fab

SER_NUM[45:41] => TESTINFO-RESERVED

SER_NUM[40] => Bin

SER_NUM[39:16] => Lot

SER_NUM[15:10] => Wafer

SER_NUM[9:5] => Column

SER_NUM[4:0] => Row

The SERNUM id has 64 bits. It uses three fuse rows, SERNUM0, SERNUM1, and SERNUM2. The SERNUM id in each row can have 22 bits of data. As the result there are possible 66 bits for SERNUM_ID in the fuse array. However the NCU register only keep 64 LSB. The upper 2 bits are shifted out of the SERNUM2 register. Those bits can be read directly from the fuse array. According to the SERNUM format above SERNUM0, SERNUM1, and SERNUM2 formats are as follow:

TABLE 11-17 Efuse Row SERNUM0 Format

<31:29>	<28>	<27:22>	<21:16>	<15:10>	<9:5>	<4:0>
Valid	Parity	Block id	Lot[5:0]	Wafer	Column	Row

TABLE 11-18 Efuse Row SERNUM1 Format

<31:29>	<28>	<27:22>	<21:19>	<18>	<17:0>
Valid	Parity	Block id	TESTINFO-RESERVED	Bin	Lot[23:6]

The most significant bit—DeltaVdd[3]—should be interpreted as a sign, and the bits DeltaVdd[2:0] define 8 positive (negative) increments (decrements) of the vdd.

DeltaVdd[3] = 0 means that the delta is an increment to the nominal vdd

DeltaVdd[3] = 1 means a decrement to the nominal vdd.

Each increment is a fixed value for the product typically in the order of 25mV. Not all steps need be used. Initially it is expected that only 2 or 3 decrements of Vdd will be allowed. Extra bits are allowed in case we need them on future products.

In an Efuse array row, this information would take the following format:

TABLE 11-19 Proposed Efuse Row SERNUM2 Format

<31:29>	<28>	<27:22>	<21:20>	<19:16>	<15:6>	<5>	<4:2>	<1:0>
Valid	Parity	Block id	TESTINFO-RESERVED	DeltaVdd [3:0]	DeltaT [9:0]	TESTINFO-RESERVED	FAB	TESTINFO-RESERVED

When the test flow determines that a change from the nominal Vdd is necessary to optimize yield, a new SERNUM2 row will be programmed into the Efuse array at a higher row address than the previous one. Thus it overwrites the previous DeltaVdd value.

Each time SERNUM2 is reprogrammed at least one additional row out of the Efuse array will be consumed.

11.4 Unit-Level Interface Signals

TABLE 11-20 Unit-Level Interface Signal s

Signal name	Direction	Size	Description
io_vpp	Input	1	Programming voltage
gclk	Input	1	L2 Input clock
tcu_aclk	Input	1	Test clock
tcu_bclk	Input	1	Test clock
tcu_pce_ov	Input	1	Scan - Override
tcu_clk_stop	Input	1	Scan stop
tcu_scan_en	Input	1	Scan enable
scan_in	Input	1	Scan input
scan_out	Output	1	Scan output
io_pgrm_en	Input	1	Program Enable
ccu_io_out	Input	1	
io_cmp_clk_sync_en	Input	1	IO to CMP clock sync enable
cmp_io_clk_sync_en	Input	1	CMP to IO clock sync enable
rst_por_	Input	1	POR reset active low
TCU to EFU			
tcu_efu_rowaddr	Input	7	Efuse row address for read/write
tcu_efu_coladdr	Input	5	Efuse column address for write
tcu_efu_read_en	Input	1	Read enable
tcu_efu_read_mode	Input	3	00=normal; 01=margin0, 10=margin1A; 11=margin1B
tcu_efu_read_start	Input	1	Start SM for scanning bits out
tcu_efu_fuse_bypass	Input	1	Shift data from TCU
tcu_efu_dest_sample	Input	1	Destination sample from TCU
TCU EFU shift interface			
tcu_efu_data_in	Input	1	Serial scan in from TCU
tcu_efu_updatedr	Input	1	Read reg update from shift register
tcu_efu_shiftdr	Input	1	Shift data register

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

tcu_efu_capturedr	Input	1	Shift data register captures read register value
tck	Input	1	Shift dr data in/out from TCU
tcu_red_reg_clr	Input	7	Redundancy register clear
efu_tcu_data_out	Output	1	Serial scan out to TCU
EFU to outside logic in the chip			
EFU and SPC interface			
efu_spc1357_fuse_data	Output	1	Efuse data to SPARC cores 1,3,5 and 7
efu_spc0246_fuse_data	Output	1	Efuse data to SPARC cores 2,4,6 and 8
efu_spc7_fuse_iclr	Output	1	SPARC core 7 I$ clear
efu_spc7_fuse_ixfer_en	Output	1	SPARC core 7 I$ transfer enable
efu_spc7_fuse_dclr	Output	1	SPARC core 7 D$ clear
efu_spc7_fuse_dxfer_en	Output	1	SPARC core 7 D$ transfer enable
spc7_efu_fuse_idata	Input	1	SPARC core 7 I$ read header data return
spc7_efu_fuse_ixfer_en	Input	1	SPARC core 7 I$ read transfer enable
spc7_efu_fuse_ddata	Input	1	SPARC core 7 D$ read header data return
spc7_efu_fuse_dxfer_en	Input	1	SPARC core 7 D$ read transfer enable
efu_spc6_fuse_iclr	Output	1	SPARC core 6 I$ clear
efu_spc6_fuse_ixfer_en	Output	1	SPARC core 6 I$ transfer enable
efu_spc6_fuse_dclr	Output	1	SPARC core 6 D$ clear
efu_spc6_fuse_dxfer_en	Output	1	SPARC core 6 D$ transfer enable
spc6_efu_fuse_idata	Input	1	SPARC core 6 I$ read header data return
spc6_efu_fuse_ixfer_en	Input	1	SPARC core 6 I$ read transfer enable
spc6_efu_fuse_ddata	Input	1	SPARC core 6 D$ read header data return
spc6_efu_fuse_dxfer_en	Input	1	SPARC core 6 D$ read transfer enable
efu_spc5_fuse_iclr	Output	1	SPARC core 5 I$ clear
efu_spc5_fuse_ixfer_en	Output	1	SPARC core 5 I$ transfer enable
efu_spc5_fuse_dclr	Output	1	SPARC core 5 D$ clear
efu_spc5_fuse_dxfer_en	Output	1	SPARC core 5 D$ transfer enable
spc5_efu_fuse_idata	Input	1	SPARC core 5 I$ read header data return
spc5_efu_fuse_ixfer_en	Input	1	SPARC core 5 I$ read transfer enable
spc5_efu_fuse_ddata	Input	1	SPARC core 5 D$ read header data return

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

spc5_efu_fuse_dxfer_en	Input	1	SPARC core 5 D$ read transfer enable
efu_spc4_fuse_iclr	Output	1	SPARC core 4 I$ clear
efu_spc4_fuse_ixfer_en	Output	1	SPARC core 4 I$ transfer enable
efu_spc4_fuse_dclr	Output	1	SPARC core 4 D$ clear
efu_spc4_fuse_dxfer_en	Output	1	SPARC core 4 D$ transfer enable
spc4_efu_fuse_idata	Input	1	SPARC core 4 I$ read header data return
spc4_efu_fuse_ixfer_en	Input	1	SPARC core 4 I$ read transfer enable
spc4_efu_fuse_ddata	Input	1	SPARC core 4 D$ read header data return
spc4_efu_fuse_dxfer_en	Input	1	SPARC core 4 D$ read transfer enable
efu_spc3_fuse_iclr	Output	1	SPARC core 3 I$ clear
efu_spc3_fuse_ixfer_en	Output	1	SPARC core 3 I$ transfer enable
efu_spc3_fuse_dclr	Output	1	SPARC core 3 D$ clear
efu_spc3_fuse_dxfer_en	Output	1	SPARC core 3 D$ transfer enable
spc3_efu_fuse_idata	Input	1	SPARC core 3 I$ read header data return
spc3_efu_fuse_ixfer_en	Input	1	SPARC core 3 I$ read transfer enable
spc3_efu_fuse_ddata	Input	1	SPARC core 3 D$ read header data return
spc3_efu_fuse_dxfer_en	Input	1	SPARC core 3 D$ read transfer enable
efu_spc2_fuse_iclr	Output	1	SPARC core 2 I$ clear
efu_spc2_fuse_ixfer_en	Output	1	SPARC core 2 I$ transfer enable
efu_spc2_fuse_dclr	Output	1	SPARC core 2 D$ clear
efu_spc2_fuse_dxfer_en	Output	1	SPARC core 2 D$ transfer enable
spc2_efu_fuse_idata	Input	1	SPARC core 2 I$ read header data return
spc2_efu_fuse_ixfer_en	Input	1	SPARC core 2 I$ read transfer enable
spc2_efu_fuse_ddata	Input	1	SPARC core 2 D$ read header data return
spc2_efu_fuse_dxfer_en	Input	1	SPARC core 2 D$ read transfer enable
efu_spc1_fuse_iclr	Output	1	SPARC core 1 I$ clear
efu_spc1_fuse_ixfer_en	Output	1	SPARC core 1 I$ transfer enable
efu_spc1_fuse_dclr	Output	1	SPARC core 1 D$ clear
efu_spc1_fuse_dxfer_en	Output	1	SPARC core 1 D$ transfer enable
spc1_efu_fuse_idata	Input	1	SPARC core 1 I$ read header data return
spc1_efu_fuse_ixfer_en	Input	1	SPARC core 1 I$ read transfer enable

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

spc1_efu_fuse_ddata	Input	1	SPARC core 1 D$ read header data return
spc1_efu_fuse_dxfer_en	Input	1	SPARC core 1 D$ read transfer enable
efu_spc0_fuse_iclr	Output	1	SPARC core 0 I$ clear
efu_spc0_fuse_ixfer_en	Output	1	SPARC core 0 I$ transfer enable
efu_spc0_fuse_dclr	Output	1	SPARC core 0 D$ clear
efu_spc0_fuse_dxfer_en	Output	1	SPARC core 0 D$ transfer enable
spc0_efu_fuse_idata	Input	1	SPARC core 0 I$ read header data return
spc0_efu_fuse_ixfer_en	Input	1	SPARC core 0 I$ read transfer enable
spc0_efu_fuse_ddata	Input	1	SPARC core 0 D$ read header data return
spc0_efu_fuse_dxfer_en	Input	1	SPARC core 0 D$ read transfer enable
L2 and EFU shift interface			
efu_l2t0246_fuse_data	Output	1	Efuse data to l2t banks 0,2,4 and 6
efu_l2t1357_fuse_data	Output	1	Efuse data to l2t banks 1,3,5 and 7
efu_l2b0246_fuse_data	Output	1	Efuse data to l2b banks 0,2,4 and 6
efu_l2b1357_fuse_data	Output	1	Efuse data to l2b banks 1,3,5 and 7
efu_l2t0_fuse_clr	Output	1	l2t bank 0 fuse data clear
efu_l2t0_fuse_xfer_en	Output	1	l2t bank 0 fuse data transfer enable
l2t0_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 0
l2t0_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 0
efu_l2b0_fuse_clr	Output	1	L2b bank 0 fuse data clear
efu_l2b0_fuse_xfer_en	Output	1	L2b bank 0 fuse data transfer enable
l2b0_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 0
l2b0_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 0
efu_l2t1_fuse_clr	Output	1	l2t bank 1 fuse data clear
efu_l2t1_fuse_xfer_en	Output	1	l2t bank 1 fuse data transfer enable
l2t1_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 1
l2t1_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 1
efu_l2b1_fuse_clr	Output	1	L2b bank 1 fuse data clear
efu_l2b1_fuse_xfer_en	Output	1	L2b bank 1 fuse data transfer enable
l2b1_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 1
l2b1_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 1

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

efu_l2t2_fuse_clr	Output	1	l2t bank 2 fuse data clear
efu_l2t2_fuse_xfer_en	Output	1	l2t bank 2 fuse data transfer enable
l2t2_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 2
l2t2_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 2
efu_l2b2_fuse_clr	Output	1	L2b bank 2 fuse data clear
efu_l2b2_fuse_xfer_en	Output	1	L2b bank 2 fuse data transfer enable
l2b2_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 2
l2b2_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 2
efu_l2t3_fuse_clr	Output	1	l2t bank 3 fuse data clear
efu_l2t3_fuse_xfer_en	Output	1	l2t bank 3 fuse data transfer enable
l2t3_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 3
l2t3_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 3
efu_l2b3_fuse_clr	Output	1	L2b bank 3 fuse data clear
efu_l2b3_fuse_xfer_en	Output	1	L2b bank 3 fuse data transfer enable
l2b3_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 3
l2b3_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 3
efu_l2t4_fuse_clr	Output	1	l2t bank 4 fuse data clear
efu_l2t4_fuse_xfer_en	Output	1	l2t bank 4 fuse data transfer enable
l2t4_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 4
l2t4_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 4
efu_l2b4_fuse_clr	Output	1	L2b bank 4 fuse data clear
efu_l2b4_fuse_xfer_en	Output	1	L2b bank 4 fuse data transfer enable
l2b4_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 4
l2b4_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 4
efu_l2t5_fuse_clr	Output	1	l2t bank 5 fuse data clear
efu_l2t5_fuse_xfer_en	Output	1	l2t bank 5 fuse data transfer enable
l2t5_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 5
l2t5_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 5
efu_l2b5_fuse_clr	Output	1	L2b bank 5 fuse data clear
efu_l2b5_fuse_xfer_en	Output	1	L2b bank 5 fuse data transfer enable
l2b5_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 5

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

l2b5_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 5
efu_l2t6_fuse_clr	Output	1	l2t bank 6 fuse data clear
efu_l2t6_fuse_xfer_en	Output	1	l2t bank 6 fuse data transfer enable
l2t6_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 6
l2t6_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 6
efu_l2b6_fuse_clr	Output	1	L2b bank 6 fuse data clear
efu_l2b6_fuse_xfer_en	Output	1	L2b bank 6 fuse data transfer enable
l2b6_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 6
l2b6_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 6
efu_l2t7_fuse_clr	Output	1	l2t bank 7 fuse data clear
efu_l2t7_fuse_xfer_en	Output	1	l2t bank 7 fuse data transfer enable
l2t7_efu_fuse_data	Input	1	Fuse read data shift from l2t bank 7
l2t7_efu_fuse_xfer_en	Input	1	Fuse read shift enable for bank 7
efu_l2b7_fuse_clr	Output	1	L2b bank 7fuse data clear
efu_l2b7_fuse_xfer_en	Output	1	L2b bank 7 fuse data transfer enable
l2b7_efu_fuse_data	Input	1	Fuse read data shift from l2b bank 7
l2b7_efu_fuse_xfer_en	Input	1	Fuse read shift enable for l2b bank 7
NCU and EFU shift interface			
efu_ncu_fuse_data	Output	1	Efuse NCU data
efu_ncu_srlnum0_xfer_en	Output	1	Efuse NCU serial number 0 transfer enable
efu_ncu_srlnum1_xfer_en	Output	1	Efuse NCU serial number 1 transfer enable
efu_ncu_srlnum2_xfer_en	Output	1	Efuse NCU serial number 2 transfer enable
efu_ncu_fusestat_xfer_en	Output	1	Efuse NCU fuse status transfer enable
efu_ncu_coreavl_xfer_en	Output	1	Efuse NCU core available transfer enable
efu_ncu_bankavl_xfer_en	Output	1	Efuse NCU bank available transfer enable
NIU and EFU shift interface			
NIU SRAM 2			
niu_efu_4k_data	Input	1	Niu to efu data
niu_efu_4k_xfer_en	Input	1	Niu to efu xfer enable

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

efu_niu_4k_clr	Output	1	Efu to niu clear
efu_niu_4k_data	Output	1	Efu to niu data
efu_niu_4k_xfer_en	Output	1	Efu to niu xfer enable
NIU SRAM 1			
niu_efu_cfifo0_data	Input	1	Niu cfifo0 data to efu
niu_efu_cfifo0_xfer_en	Input	1	Niu cfifo0 xfer enable to efu
efu_niu_cfifo0_clr	Output	1	Efu to niu cfifo0 clear
efu_niu_cfifo0_xfer_en	Output	1	Efu to niu cfifo0 xfer enable
niu_efu_cfifo1_data	Input	1	Niu cfifo1 data to efu
niu_efu_cfifo1_xfer_en	Input	1	Niu cfifo1 xfer enable to efu
efu_niu_cfifo1_clr	Output	1	Efu to niu cfifo1 clear
efu_niu_cfifo1_xfer_en	Output	1	Efu to niu cfifo1 xfer enable
efu_niu_cfifo_data	Output	1	Share data from efu to niu for this group of RAMs
NIU SRAM 0			
niu_efu_ipp0_data	Input	1	Niu ipp0 data to efu
niu_efu_ipp0_xfer_en	Input	1	Niu ipp0 xfer enable to efu
efu_niu_ipp0_clr	Output	1	Efu to niu ipp0 clear
efu_niu_ipp0_xfer_en	Output	1	Efu to niu ipp0 xfer enable
niu_efu_ipp1_data	Input	1	Niu ipp1 data to efu
niu_efu_ipp1_xfer_en	Input	1	Niu ipp1 xfer enable to efu
efu_niu_ipp1_clr	Output	1	Efu to niu ipp1 clear
efu_niu_ipp1_xfer_en	Output	1	Efu to niu ipp1 xfer enable
niu_efu_mac0_ro_data	Input	1	Niu mac0 ro data to efu
niu_efu_mac0_ro_xfer_en	Input	1	Niu mac0 ro xfer enable to efu
niu_efu_mac1_ro_data	Input	1	Niu mac1 ro data to efu
niu_efu_mac1_ro_xfer_en	Input	1	Niu mac1 ro xfer enable to efu
niu_efu_mac0_sf_data	Input	1	Niu mac0 sf data to efu
niu_efu_mac0_sf_xfer_en	Input	1	Niu mac0 sf xfer enable to efu
efu_niu_mac0_ro_clr	Output	1	Efu to niu mac0 ro clear
efu_niu_mac0_ro_xfer_en	Output	1	Efu to niu mac0 ro xfer enable
efu_niu_mac0_sf_clr	Output	1	Efu to niu mac0 sf clear

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

efu_niu_mac0_sf_xfer_en	Output	1	Efu to niu mac0 sf xfer enable
niu_efu_mac1_sf_data	Input	1	Niu mac1 sf data to efu
niu_efu_mac1_sf_xfer_en	Input	1	Niu mac1 sf xfer enable to efu
efu_niu_mac1_ro_clr	Output	1	Efu to niu mac1 ro clear
efu_niu_mac1_ro_xfer_en	Output	1	Efu to niu mac1 ro xfer enable
efu_niu_mac1_sf_clr	Output	1	Efu to niu mac1 sf clear
efu_niu_mac1_sf_xfer_en	Output	1	Efu to niu mac1 sf xfer enable
efu_niu_mac01_sfro_data	Output	1	Efu to niu mac01 sfro data
NIU SRAM3			
niu_efu_ram0_data	Input	1	Niu ram0 data to efu
niu_efu_ram0_xfer_en	Input	1	Niu ram0 data xfer enable to efu
efu_niu_ram0_clr	Output	1	Efu to niu ram0 clear
efu_niu_ram0_xfer_en	Output	1	Efu to niu ram0 xfer enable
niu_efu_ram1_data	Input	1	Niu ram1 data to efu
niu_efu_ram1_xfer_en	Input	1	Niu ram1 xfer enable to efu
efu_niu_ram1_clr	Output	1	Efu to niu ram1 clear
efu_niu_ram1_xfer_en	Output	1	Efu to niu ram1 xfer enable
niu_efu_ram_data	Input	1	Niu ram data to efu
niu_efu_ram_xfer_en	Input	1	Niu ram xfer enable to efu
efu_niu_ram_clr	Output	1	Efu to niu ram clear
efu_niu_ram_xfer_en	Output	1	Efu to niu ram xfer enable
efu_niu_ram_data	Output	1	Efu to niu ram, ram0, ram1 data in
NIU SERDES i/f			
niu_efu_fdo	Input	1	Niu to efu data
efu_niu_fclk	Output	1	Efu to niu fclk (100MHz)
efu_niu_fclrz	Output	1	Efu to niu clear
efu_niu_fdi	Output	1	Efu to niu data in
PEU and EFU shift interface			
psr_efu_fdo	Input	1	Psr to efu data
efu_psr_fclk	Output	1	Efu to psr clock
efu_psr_fclrz	Output	1	Efu to psr clear

TABLE 11-20 Unit-Level Interface Signal *(Continued)*s

efu_psr_fdi	Output	1	Efu to psr data
MCU and EFU shift interface			
mcu_efu_fdo	Input	1	Mcu to efu data
efu_mcu_fclk	Output	1	Efu to mcu clock
efu_mcu_fclrz	Output	1	Efu to mcu clear
efu_mcu_fdi	Output	1	Efu to mcu data
DMU and EFU shift interface			
dmu_efu_data	Input	1	Dmu to efu data
dmu_efu_xfer_en	Input	1	Dmu to efu xfer enable
efu_dmu_clr	Output	1	Efu to dmu clear
efu_dmu_data	Output	1	Efu to dmu data
efu_dmu_xfer_en	Output	1	Efu to dmu xfer enable

11.5 Misc/Multiple Clock Domains

The following signals coming from the TCU will have to be synchronized to the io clock domain before use since they are generated on the tck (JTAG) clock.

```
tcu_efu_read_start
tcu_efu_read_en
tcu_efu_fuse_bypass
tcu_efu_updatedr
tcu_efu_rowaddr[6:0]
```

A special synchronizer library cell will be used to synchronize the above signals. tcu_efu_rowaddr[6:0] is fed through the same synchronizer. This signal is assumed to be stable before being used and hence is not qualified with any valid signal.

The following signals are not synchronized. They are used only on the tester and hence are assumed to transition and settle well before they are used. They do not need explicit synchronization.

```
io_pgrm_en
tcu_efu_coladdr[4:0]
tcu_efu_read_mode[1:0]
```

The following signals are generated and used in the tck clock domain:

```
tcu_efu_data_in
efu_tcu_data_out
tcu_efu_shiftdr
tcu_efu_capturedr
tck
```

11.6 Efuse Array Specification

11.6.1 Efuse Array Organization

In a broad sense, the Efuse array is a non-volatile memory used to store information that needs to be programmed at the factory and used in the field. On OpenSPARC T2, it contains the following die specific information :

- Redundant array repair information for the SRAMs
- Serial ID of the chip
- Working processor core IDs (core available information)
- Working L2 bank information (bank available information)

The Efuse array is a 64 deep and 32 bit wide array. Each cell in the Efuse array consists of a poly fuse that replace traditional laser fuse. They can be programmed to store any value by blowing them with an electrical pulse.

11.6.2 Efuse Array Functions

Supports the following 2 funcitons :

1. Read access : There can be 2 types of read access.

   a. EFA row read : Contents of an entry in the array specified by fct_efa_word_addr[5:0] are read out as fct_efa_data_out[31:0].

   b. Supply detect read : efa_sup_det_rd is asserted indicating a request for supply detect read. EFA will read out voltage levels and sense amplifier power levels (vpp_detect, vdd_detect, vddo_detect, and sense amplifier power level detect) as efa_sbc_data[3:0].

2. Program access : EFA array is programmed one bit at a time. The fct_efa_prog_en needs to be asserted requesting a program access. The vpp bump pads needs to be supplied with special voltage before fct_efa_prog_en is valid. fct_efa_word_addr[5:0] and fct_efa_bit_addr[4:0] needs to be valid. The bit in entry specified by fct_efa_word_addr[5:0] and fct_efa_bit_addr[4:0] is programmed. After a entry is programmed, the entry is read back. If the desired value is not obtained, the mismatched bits if possible are reprogrammed. The process is repeated until desired value is read. Valid bits are then programmed when the entry is programmed with valid data.

11.6.3 Timing Diagrams

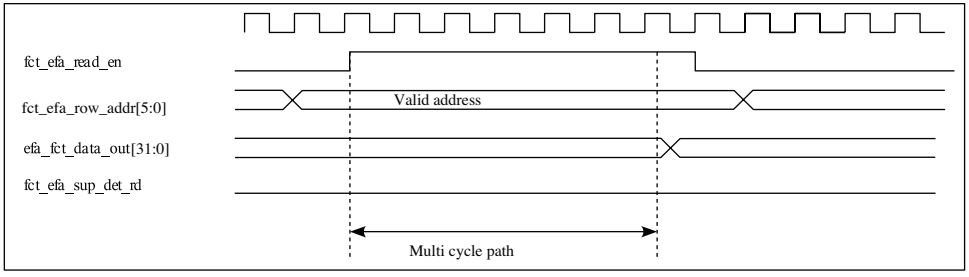
Read access :

EFA row read :

The Efuse controller (FCT) will request a normal read operation by asserting fct_efa_read_en along with fct_efa_row_addr[5:0].

FCT will assert fct_efa_read_en for a predetermined number of clocks (as per the requirements of the EFA) for the read data efa_fct_data_out[31:0] to be ready.

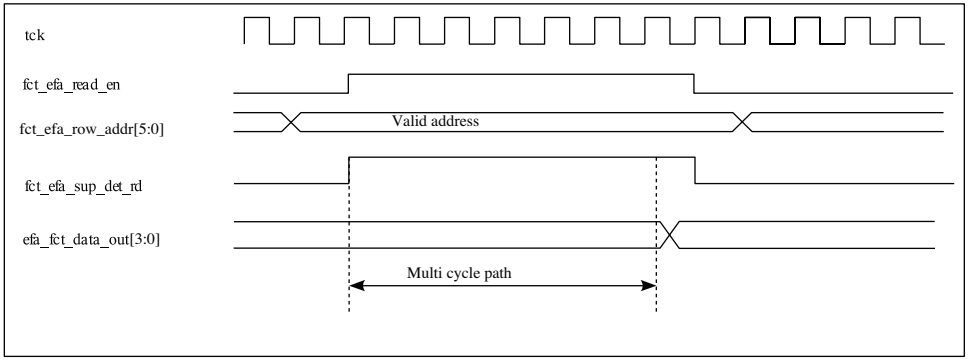
FIGURE 11-9 EFA Row Read Access



Supply detect read :

FCT can perform a supply read detect by asserting fct_efa_sup_det_rd. In this case EFA will read out various voltage levels (vpp_detect, vdd_detect, vddo_detect, and sense amplifier power level detect) as efa_fct_data_out[3:0].

FIGURE 11-10 EFA Supply Detect Access



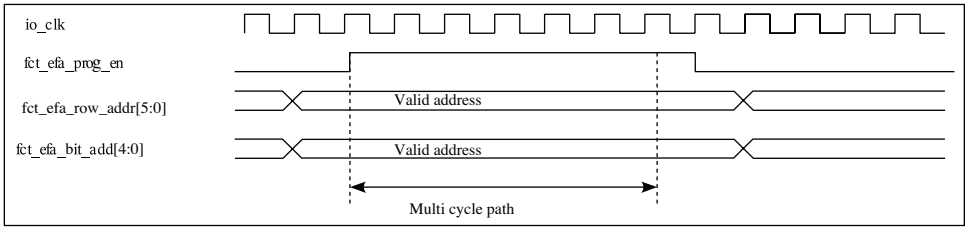
Program access :

Programming happens one bit at a time.

EFA will request a program access by asserting fct_efa_row_addr[5:0], fct_efa_bit_addr[4:0] and fct_efa_prog_en.

EFA will zap the fuse in the bit cell identified by the row address (fct_efa_row_addr[5:0]) and the bit address (fct_efa_bit_addr[4:0]). (The zapped bit will be read as a zero)

FIGURE 11-11 EFA Program Access



11.6.4 Interface Table

TABLE 11-21 Interface Table for EFA

Signal name	I/O	Width	Description
Vpp	I	1	VPP input from I/O
fct_efa_prog_en	I	1	Efuse array program enable
fct_efa_read_en	I	1	Efuse array read enable
fct_efa_word_addr	I	6	Efuse array word address from TCU
fct_efa_bit_addr	I	5	Efuse array bit address
fct_efa_sup_det_rd	I	1	Efuse array supply detect read
fct_efa_power_down	I	1	Efuse array power down signal from SBC
scan_in	I	1	Scan input
scan_en	I	1	Scan enable
clk	I	1	Clock
scan_out	O	1	Scan output
efa_fct_data	O	32	Data from Efuse array to SBC
fct_efa_margin0_rd	I	1	Efuse array margin0 read
fct_efa_margin1_rd	I	1	Efuse array margin1 read

CHAPTER 12

Reset Unit Specification

This chapter contains the following sections:

12.1 OpenSPARC T1 and OpenSPARC T2 Partitioning

Except for the system controller, OpenSPARC T2 integrates all motherboard system circuitry on a chip. While OpenSPARC T1 writes to a register on an external IO Bridge chip to assert WMR_RST, OpenSPARC T2 writes to the on-chip RESET_GEN register.

TABLE 12-1 OpenSPARC Partitioning

OpenSPARC T1		OpenSPARC T2	
Abbr.	**Unit**	**Abbr.**	**Unit**
IOB	(Internal) IO Bridge Unit	NCU	Non-Cacheable Unit
n.a.	External IO Bridge chip		
CTU	Control and Test Unit	TCU	Test Control Unit
		CCU	Clock Control Unit
		RST	Reset Unit
		CMP	Chip MultiProcessor Unit (may be part of NCU)
	JBus System Interface	DMU	Data Management Unit
n.a.	n.a.	PEU	PCI Express Unit
n.a.	n.a.	NIU	Network Interface Unit

12.2 Reset Overview

12.2.1 Goals

The Reset Unit asserts signals that cause other units to immediately revert to the initial state defined by the Programmer's Reference Manual.

The OpenSPARC T2 team has endeavored to keep OpenSPARC T2 as much the same as OpenSPARC T1 as possible. One major difference is that OpenSPARC T2 conforms to the CMP Programming Model.

12.2.2 Nomenclature

In the specifications relevant to OpenSPARC T2, the term *reset* is used in many ways. The table in the following section lists all of them except one. The exception is that the two-bit TYPE field of the OpenSPARC T1 INT_VEC_DIS register can take on a value named reset, described below in the OpenSPARC T1 Thread Suspension section, but that register field named reset differs from signals named reset in the sense used here.

Exercise caution in referring to the various documents, as a single reset can have multiple names. Power-on reset has several names: POR, cold power-on, flush, scan flush, and hard reset. In fact, the chip-wide warm reset also goes by at least six other names: chip, CR, full-CMP, soft, software induced warm, and system reset.

Conversely, resets that are different can have names that are similar (soft or software induced warm differ from Software-Initiated), or have identical abbreviations (chip-wide WMR differs from OpenSPARC T1 thread WMR). The CMP Programming Model considers POR to be a special case of system reset.

OpenSPARC T2 retains the reset concepts and names used in OpenSPARC T1. TABLE 12-2 expands upon Table 5-4 in the Fire ASIC spec.

TABLE 12-2 Reset Actions

Function	Sun/Fire ASIC	PCI-Express Spec	OpenSPARC T1	OpenSPARC T2
Reset of all chip state including errors	Hard	Cold	Power-On	
Reset of all non-error chip state	Soft	Warm	Warm	

12.2.3 Priority

The *OpenSPARC T1 Programmer's Reference Manual* and the *OpenSPARC T2 Programmer's Reference Manual* give the trap types of the resets in TABLE 12-3. Priority 1 traps, which are resets other than POR, are processed in the following order (with the trap type in parentheses):
XIR(3) > WDR(2) > SIR(4) > RED(5).

Reset priorities from highest to lowest are:

TABLE 12-3 Trap Types

Trap Type	Abbr.	Reset Name	Priority	Cause	Scope
-	TRST_	Test Reset	Assert with POR	TRST_	TCU only
1	POR	Power-On, cold power-on, flush, scan flush, hard	Highest	PWRON_RST_L	Chip-wide except TCU
1	WMR	Chip-wide warm, chip, CR, full-CMP, soft, software induced warm, system	WMR < POR	PB_RST_L (1st pri.), or Fatal Error (2nd pri.), or write Gen_Reset reg (ctu40) or RESET_GEN reg (ctu39) (3rd pri.)	Chip-wide, except for WMR-protected flops. See PRM, Table 11-13.
-	DBG_INIT	debug_init_	Same as WMR	rst_wmr_, or PIO read to DBG_INIT reg (ctu56, n1prm369, n2prm386)	OpenSPARC T1 only. Replaced by DBR in OpenSPARC T2.
-	DBR	Debug	DBR< WMR	Write to DBR_GEN bit of Reset_Gen reg	Full chip, except NIU and DMU-PEU
1*	WMR trap	Warm Reset trap	WMR < POR	Write INT_VEC_DIS reg	Per thread
3*	XIR	Externally-Initiated	XIR < WMR	BUTTON_XIR_L (1st pri.), or set bit[1] in Reset_Gen reg (tprm190)	Virtual cores set in ASI_XIR_STEERING
2*	WDR	Watchdog	WDR < XIR	Write INT_VEC_DIS reg	Per thread
4*	SIR	Software-Initiated	Lowest	Issue SIR (SIGM) instr in priv mode, or write INT_VEC_DIS reg	Per thread
-	NIU	NIU	-	Write to NIU bit of SSYS_RESET reg	NIU

*Note: WMR trap, XIR, WDR, and SIR do not cause other units to immediately revert to the initial state defined by the Programmer's Reference Manual. They are interrupt traps. "Software can distinguish a chip-wide Warm Reset from a Warm Reset [trap] by the RSET_STAT register."

Software can distinguish a POR from a WMR by the RSET_STAT register, as follows:
Reset from WMR_RSTpin sets WMR bit of RSET_STAT.
Reset from PWRON_RST_pin sets PWRON_RST bit of RSET_STAT.

TABLE 12-4 Preemption

TRST	The FPGA or tester can assert TRST_ at any time, and it will reset the JTag Test Access Port.
POR	The FPGA or tester can assert PWRON_RST_L at any time, and it will reset OpenSPARC T2 immediately.
WMR, DBR	If the Reset Unit is in the engaged in servicing a prior POR, WMR, or DBR, it will defer processing a WMR, a DBR, or an XIR until it finishes the prior one. If, at that time, it finds more than one reset pending, it will choose the highest priority, according to the table above.
XIR	If the Reset Unit is in the engaged in servicing a Externally-Initiated Reset, it will allow any other reset to preempt the XIR.

12.2.4 OpenSPARC T2 Structures that Hold State

OpenSPARC T2 holds state in three types of structure:

1. Latches.
2. Flip-flops. These may be:
   a. Synchronously-resettable.
      The Asic clusters use synchronous reset.
   b. Asynchronously-resettable.
      Only the CCU and the cluster headers contain asynchronously-resettable flops.
   c. Resettable by flush reset.
      The SPARC core clusters use flush reset.
   d. Resettable by having a known value shift down a pipeline.
   e. Non-resettable.
3. Array cores.
   a. E-Fuse Array.
   b. SRAM array cores.

An SRAM may use each of the three kinds of structure to hold state:

1. Latches. There are three types:

a. SRAM redundant array Repair Value latches. (Not shown.) An SRAM may hold its Repair Values in flops instead, depending upon its area requirements. See next section.

b. Other SRAM latches.

c. Latches in the path of the clock pre-grid drivers. These latches remain, even though multiple drivers, each controlled by its own latch, are shorted together through the grid. describes how we avoid clock contention in Asic cluster SRAMs without resetting these latches. SunV cluster SRAMS are held in flush reset until gclk starts, and flush reset asserts SE, which is how we avoid clock contention in these SRAMs.

2. Flip Flops:

   a. SRAM redundant array Repair Value flops. (Not shown.) An SRAM may hold its Repair Values in latches instead, depending upon its area requirements. See next section.

   b. SRAM input flops. The L2T CAM only has a latch at the input. All other SRAMs have input flops.

   c. SRAM output flops. Approximately 35 percent of SRAMs have an output flop. The remaining 65 percent have a latch, instead.

3. SRAM array core contents. A special case of SRAM contents is the valid bits in the L2 directory of L1 tags.

12.2.5 E-Fuse destination Flops and Latches

"EFUSE OpenSPARC T2 Micro-Architecture Specification" lists the destinations of information from the EFU, as shown in TABLE 12-5:

TABLE 12-5 Destination of Information from the EFU

Block ID	Destination	Information
00-15	SPC	I-cache and D-cache Repair Values (RV)
16-31	L2T,L2D	SRAM RV
32	NCU	SPARC core available
33	NCU	L2 bank available
34, 35, 36	NCU	Serial number
37-40	NIU	SRAM RV
41	PSR	PSR Serdes termination resistor trimming

TABLE 12-5 Destination of Information from the EFU *(Continued)*

42	MCU	FSR Serdes termination resistor trimming
43	NIU	ESR Serdes termination resistor trimming
44-62	DMU/IOMMU	DEVTSB RAM delay chain calibration

Only the destination flops in the NCU are affected by flush reset, and since they are Warm Reset-protected, they are only reset during POR1 and POR2. Since they are reset by POR2, the Power-On Reset sequence includes EFU2.

All of the other destination flops and latches are only set to their initial values by the TCU via the E-Fuse Unit, before the transfer starts, using flash (synchronous) reset. An SRAM may hold its Repair Values in latches or flip-flops, depending upon its area requirements. If it holds its RVs in flops, they are not scannable, so that they are protected from flush reset. The initial value for all SRAM RVs is 0. The initial value for the Serdes's termination resistor trimming may be other than 0. The DMU/IOMMU DEVTSB RAM delay chain calibration will initialize to 4'b0010.

12.2.6 Latches

TABLE 12-6 lists each kind of latch, the agent that sets it to its initial value, and how it does it

TABLE 12-6 Latch Kind

Latch	Agent that initializes value	Method of initializing
SRAM redundant array Repair Value valid bit latches	TCU via E-Fuse Unit, before the transfer starts	Flash (synchronous) reset
Other SRAM redundant array Repair Value latches	E-Fuse Unit	Write to latch. (These bits have no effect if the valid bit is cleared.)
Other SRAM latches	(Not resettable.)	(See SRAM tables, below.)
Valid bits in the L2T directory of L1 tags, implemented as latches (a special case of SRAM core array contents.)	Reset Unit	Flash (synchronous) reset (See Section 5.3)

12.2.7 Flip-flops Outside of SRAMs

Flip-flops may be found in special clusters, in flop stations, and in SPARC core and ASIC clusters. Within SPARC core and ASIC clusters, they may be found in cluster headers, in SRAMs, and in the rest of the cluster. TABLE 12-7 lists each kind of flip-flop, except for SRAM input and output flops:

TABLE 12-7 Flip-Flop Kinds

Flip-flops outside of SRAMs	Agent that initializes value	Method of initializing
7-bit counter in Process Control Monitor (PCM)	Raw PWRON_RST_ input pin (not synchronized)	Reset of unknown type.
In Test Access Port (TAP)	TAP or TRST input pin	Asynchronous and synchronous reset
Boundary scan flops	(Not resettable.)	JTAG
In Reset Unit rst_fsm_ctl module	PWRON_RST_ input pin, after synchronized to ccu_rst_sys_clk	Synchronous reset
In Reset Unit rst_ucbflow_ctl module	Reset Unit	Synchronous reset
In Reset Unit rst_cmp_ctl and rst_io_ctl modules	Some by Reset Unit, and some by resettable.	Synchronous reset, or known value shifts down pipeline.
In global distribution flop stations (approximately 500 flops)	(Not resettable, and not scannable.)	Known value shifts down pipeline in ~5 cycles.
In SPARC core cluster headers	(Not resettable, and not scannable.)	Known value shifts down pipeline in ~5 cycles.
In ASIC cluster headers (CCU, DMU, PEU, and NIU)	(Not resettable, and not scannable.)	Known value shifts down pipeline in ~5 cycles.
In non-SRAM, non-cluster header portions of ASIC clusters (DMU, PEU, NIU, and parts of CCU)	Reset Unit	Synchronous reset
In CCU	Reset Unit	Synchronous and asynchronous reset
Some flops in MAC cluster of NIU	(Not resettable.)	Known value shifts down pipeline.

TABLE 12-7 Flip-Flop Kinds *(Continued)*

FSR SERDES	MCU via config. bus	LFSR has ckt to prevent lockout value. String of 1s flushes out bubble in the middle.
PSR SERDES	PEU via config. bus	
ESR SERDES	NIU vis config. bus	
ESR SERDES	Software resettable from SCR MAC.	Synchronous reset
Shadow-scan flops in non-cluster header portions of SunV clusters	(Not resettable.)	Acquires value of master flop after first clock cycle.
In non-cluster header portions of SPARC core clusters, other than the Reset Unit	Reset Unit	Flush reset

Notice that the Reset Unit only resets flops in clusters, and it does not affect flops in the following:

1. The PCM (partially in the MIO).
2. The TAP (in the TCU).
3. The boundary scan flops don't really need to be flush reset, since they are bypassed in functional mode and must be specifically selected by JTAG to be active. When they are selected, the chip is no longer in a functional mode.
4. Serdes clusters. The Reset Unit affects these indirectly, since for each Serdes, it resets its configuration register in the cluster that controls it. MCU controls FSR, PEU controls PSR, and NIU controls ESR.
5. The Reset Unit itself, except as part of normal logic operation.
6. Global distribution flop stations.
7. Some flops in some SRAMs (see next two sections).

Besides the TAP, which the Reset Unit does not affect, there are two blocks that contain asynchronously-resettable flops

1. CCU,reset by rst_ccu_ and rst_ccu_pll_, and
2. Cluster header,reset by cluster_arst_.

The Reset Unit will suppress its rst_ccu_, rst_ccu_pll_, and cluster_rst_l output ports when it is being scanned.

12.2.8 SRAM Input flops

TABLE 12-8 lists each kind of SRAM input flop:

TABLE 12-8 SRAM Input Flops

SRAM Input Flops	Agent that initializes value	Method of initializing
In L2T CAM	(Latch, not input flop.)	(Latch, not input flop.)
In DMU, PEU, and NIU	(Not resettable.)	Known value shifts in from upstream logic.
In SunV clusters	Reset Unit	Flush reset

12.2.9 SRAM Output Flops

TABLE 12-9 lists each kind of SRAM output flop:

TABLE 12-9 SRAM Output Flops

SRAM Output Flops	Agent that initializes value	Method of initializing
In 65 percent of SRAMs	(Latch, not output flop.)	Pre-MBISI value shifts in from core array.
In DMU and PEU	(No output flops.)	(No output flops.)
In NIU, 15 instances of 4 types of compiled SRAMs that are shared with other clusters	(Not resettable.)	Pre-MBISI value shifts in from core array.
In NIU, custom (latches) SRAMs that are unique to the NIU	Reset Unit	Synchronous reset
In SPARC core clusters	Reset Unit	Flush reset

MBisi now performs a read after competing all writes, for the purpose of initializing SRAM output flops. This requires twice as much time to complete MBisi, but we also now have the JTag POR instruction to abort MBisi, if desired.

12.2.10 Core Array Contents

TABLE 12-10 lists each kind of core array contents:

TABLE 12-10 Core Array Contents

Core array contents	Agent that initializes value	Method of initializing
E-Fuse Array contents	Factory	Fuse blow
E-Fuse Array block	Reset Unit via EFU	(See note at end of this section.)
SRAM core array contents	(Not resettable.)	MBISI or MBIST writes to SRAM.
Valid bits in the L2T directory of L1 tags, implemented as latches (a special case of SRAM core array contents.)	Reset Unit	Flash reset, a kind of synchronous reset. (See Section 5.3)

There is one reset signal (por_n) which comes to n2_efa_sp_256b_cust. The function of the por_n in the Efuse array is to ensure the following:

1. The output of the efa-array is reset to zero at powerup.
2. The readpath of the efuse-array is disabled at powerup, sustained to be in the disable state by primary inputs.
3. The efuse-array is precharged during the powerup with por_n and sustained to be in the precharge state by primary inputs.

The por_n is used to reset the flops inside the efuse-array because the efuse-array is NOT on the scan chain.

12.2.11 NIU, DMU-PEU, RST, and TAP Reset Implementations Differ

The flip-flops in OpenSPARC T2 's library have no reset input. Instead, each flip-flop is reset in one of two ways:

1. Flush reset. The Reset Unit resets most flip-flops by flush reset.
2. Synchronous reset. The two IO clusters, NIU and DMU-PEU, as well as the Reset Unit [and the TAP], use synchronous reset. Each flip-flop in these clusters has a mux feeding its D input. The reset signal directs the mux to select either (1) an initial value or (2) an operational value. The flip-flop loads this value at the next rising edge of the clock.

12.2.12 Eliminating Clock Contention

To eliminate clock contention in Asic cluster SRAMs, assert SE. (SPARC core cluster SRAMs are held in flush reset until gclk starts, and flush reset asserts SE, which is how we avoid clock contention in those clusters.)

To eliminate clock contention in the CCX cluster, assert cluster_arst_l.

Clock contention is only a problem at the beginning of POR1, before gclk has started running for the first time. During later resets, even if gclk stops, since gclk has already been running, every pair of flops and every pair of latches that are capable of causing contention have the same value.

12.2.12.1 Before gclk starts

1. To eliminate clock contention in Asic cluster SRAMs before gclk starts, assert SE. Also assert cluster_arst_l, for both the Asic cluster SRAMs and CCX.

   The l1clk header l1clk output has an Or gate, l1clk = (other signals) | SE, so SE = 1 will cause every l1clk header to drive l1clk = 1. No contention.

   Also, the cluster header l2clk output has an And gate, l2clk = (other signals) & cluster_arst_l, so cluster_arst_l = 0 will cause every cluster header to drive l2clk = 0.

   This makes the l1clk header latch transparent. Within each SRAM, the multiple l1clk headers have the same inputs. The transparent latch will cause the multiple l1clk headers to drive l1clk the same. No contention.

2. To eliminate clock contention in CCX cluster before gclk starts, assert cluster_arst_l.

   As described above, the cluster header l2clk output has an And gate, l2clk = (other signals) & cluster_arst_l, so cluster_arst_l = 0 will cause every cluster header to drive l2clk = 0. No contention.

12.2.12.2 After gclk starts, Asic SE deasserts, and Asic clk_ctop deasserts

1. In Asic cluster SRAMs: After about 5 gclk cycles, known values shifted down pipeline to both Asic and SPARC core cluster headers. Flops in both CCX cluster headers will then contain identical values. Can deassert multi-cycle cluster_arst_l and give it time to propagate. After deasserted cluster_arst_l arrives at cluster header, asserted Asic clk_stop continues to cause l2clk = 0.

After gclk starts, the l1clk header latch will operate. Within each SRAM, the multiple l1clk headers have the same inputs. The operating latch will cause the multiple l1clk headers to drive l1clk the same. No contention.

Deassert multi-cycle Asic SE and give it time to propagate. This releases l1clk from 1 and causes it to follow l2clk = 0. Deassert Asic stop_clk. This releases l2clk from 0 and allows it to follow gclk.

Continue to assert:

rst_dmu_peu_por_ = gl_dmu_por_ = gl_peu_por_. Allow at least 1 l1clk edge for Asic synchronous reset. Deassert:

rst_dmu_peu_por_ = gl_dmu_por_ = gl_peu_por_.

2. In CCX: After about 5 gclk cycles, known values shifted down pipeline to both Asic and SunV cluster headers. Flops in both CCX cluster headers contain identical values. No contention. Can now safely deassert multi-cycle cluster_arst_l and give it time to propagate.

12.2.12.3 Two Signals RequireAsynchronous Assert, Synchronous Deassert.

To eliminate clock contention, (1) rst_tcu_pwronrst_l and (2) cluster_arst_l require asynchronous assert and synchronous deassert.

1. The TCU asserts SE when the Reset Unit holds it in reset through rst_tcu_pwronrst_l. We assert this signal asynchronously, because we wish to eliminate clock contention even before sys_clk starts. We deassert this signal synchronously, because we wish the behavior of OpenSPARC T2 to be deterministic and repeatable. (Implementation note: we achieve this asynchronous assert and synchronous deassert by providing an And gate that bypasses a synchronization flop.)
2. The Reset Unit drives cluster_arst_l, so it drives this signal with asynchronous assert and synchronous deassert, just as it does rst_tcu_pwronrst_l, above.

12.3 Types of Reset

12.3.1 TRST_

TRST_ only involves the TCU, and not RST. The IEEE 1149.1 JTag Spec. requires five external pins:

TCK	Test Clock
TMS	Test Mode Select
TDIT	Test Data In
TDO	Test Data Out
TRST_	Test Reset

An alternate way to reset the JTag TAP is for the service processor to assert TMS for 5 clock cycles.

12.3.2 POR

Power-On Reset clears all flip-flops in OpenSPARC T2 clusters, except the JTag portion of TCU, which must be reset earlier by TRST. POR also clears the valid bits in the L2 cache directory of L1 tags.

Deassertion of PWRON_RST_L causes the Reset Unit to start the Power-On Reset sequence.

12.3.3 DBR

OpenSPARC T2 uses Debug Reset, DBR (instead of the DEBUG_INIT that OpenSPARC T1 uses). It is the same as WMR, but does not reset NIU nor DMU-PEU. PCI Express resets after 50 ms, so we want to do checkpoint and watchpoint and restore in 25 ms. Programming note: Be sure to configure MBIST not to run before triggering DBR.

12.3.4 WMR

By definition, Warm Reset occurs after the chip has already been running. It differs from POR in three ways:

1. It clears flip-flops in state machines, just as POR does, to ensure the chip will be able to run, but WMR attempts to maintain as much state as possible for error logging. After a WMR, this state is available to enable software to determine the cause of the reset. WMR must invalidate L2 cache to be coherent. There is no permanent state in the L1 caches since they are write-through, so WMR invalidates the L1 tags and parity, if WMR runs BISI. Section 12.4.2, "Reset Signals Asserted for each Kind of Reset" on page 12-23.
2. The EFU does not scan out the EFA again.
3. MCU continues to perform refresh cycles in order to preserve main memory contents. (It does this by placing the SDRams in self-refresh mode.) Software enables this by setting the MCU_SELFRSH bit in the SSYS_RESET register.

Three agents can cause a Warm Reset, as follows:

1. The user can press the Warm Reset pushbutton, or the external system processor can assert the PB_RST_L input pin.
2. Software can write to the WMR_GEN bit of the RESET_GEN register.
3. The L2 cache can detect a Fatal Error. (See Section 12.3.4.1, "A Fatal Error causes a WMR" on page 12-15.)

12.3.4.1 A Fatal Error causes a WMR

The two OpenSPARC T2 Fatal Errors are as follows:

1. LRU: L2 cache directory Uncorrectable parity error. "During directory scrub, parity is checked for the directory entry."
2. LVU: L2 cache VAD array Uncorrectable parity error. "On every L2 access, parity is checked for all 12 VAD bits in the set. (The used bit of VUAD is not covered by parity since it only affects performance, not correctness.)"

"When an Uncorrectable parity error is detected, the error information is captured in the L2 Cache Error Status and L2 Cache Error Address registers. In addition, a fatal error indication is issued... to request a warm_reset trap to the entire chip."

When the L2 cache detects either of these errors, it asserts l2t_rst_fatal. [Actually 8 signals.]

Even though the Fire documents make reference to Fatal Error, that case differs from what this document calls Fatal Error. OpenSPARC T2 will handle that case via an interrupt. If the interrupt handler decides a Warm Reset is needed, it can then cause it. For example, “Fire can initiate a Fatal Error [warm] reset...“Note: A fatal error is an error which causes the chip to no longer function in the manner it was designed for. A fatal error requires a reset, and there is no way to recover from it without a reset."

Fatal Errors can be masked by a register in the NCU, Fatal Error Enable - FEE (0x3020). "Each error type may be programmed to cause a Fatal Error. This register enables an error to cause the signal ncu_rst_fatal_error to be asserted to the Reset Unit. If the respective "Fatal Error Enable" bit is set, and the corresponding error type is asserted, a fatal error will be dispatched to the Reset Unit."

We reserve the right to add a third Fatal Error, if we discover a way to detect that a transaction queue is wedged, or has a bad address or control. (We can confine bad data to one thread.) We wish to prevent bad data from getting off the chip.

12.3.4.2 Conflicting Demands placed on WMR

Warm reset serves two purposes:

1. Test
2. Fatal error

1. Test involves hundreds of functional vectors. For example, the table below shows the percent of time on the tester for various steps in testing OpenSPARC T2. Also, to make a test reproducible, it must start from a known state. This tends to demand resetting as much of the chip's state as possible.
2. Since a WMR can occur due to a Fatal Error, it attempts to maintain as much state as possible for error logging. After a WMR, this state is available to enable software to determine the cause of the reset This tends to demand resetting as little of the chip's state as possible.

To satisfy both of these demands, WMR keeps memory state, L2 cache, error logs, and most architecturally-visible registers. It discards: transactions in flight, store buffers, and FIFOs, puts each state machine into its idle state, and lets the pipeline register drain. For a table of core CPU state as a result of POR and WMR, see Section 11.9, “Machine State after Reset and in RED_State”.

TABLE 12-11 Chip Reset

Chip reset step	Jclks	Percent of diags
POR/PLL	34800	27
EFA	8000	6

TABLE 12-11 Chip Reset *(Continued)*

WRM	4000	3
BISI	2600	2
SSI	27800	22
Total reset steps	77200	60
Total for 145 diags	127200	100
Diag portion other than reset	50000	40

12.3.5 WMR Trap and SPARC-V9 POR Trap

The WMR trap generates a SPARC-V9 POR trap, which has a trap type of 1.

12.3.5.1 How OpenSPARC T1 Starts its Virtual Cores at Reset

To start its virtual cores at reset, OpenSPARC T1 uses the Warm Reset trap.

1. The IOB starts the first virtual core with an interrupt: "Cold Power-On Reset Sequence, Step 19: IOB sends interrupt to Thread 0 on first available core."
2. That virtual core then starts each of the others with an interrupt: "11.10.1 Assumed POR software Initialization Sequence, Step 19: Send interrupts to other available cores."

From the *OpenSPARC T1 Programmer's Reference Manual*:

11.5.1 Warm Reset [Trap] (WMR [Trap])

"A thread can be sent a WMR [Trap] via the INT_VEC_DIS register. The warm reset [trap] generates a SPARC-V9 POR [trap], which has a trap type of 001_{16} at physical address offset 20_{16}. Software can distinguish a thread warm reset [trap] from a chipwide warm reset by the RSET_STAT register. Since thread resets [traps] do not set any bits in this register, and software will clear the chipwide reset bits after the reset sequence has been completed, a RSET_STAT with all reset source bits cleared will signal to the thread that it received a thread warm reset [trap].

OpenSPARC T1 and T2 both have an Interrupt/Trap Vector Dispatch Register, INT_VEC_DIS. For OpenSPARC T1, INT_VEC_DIS is in the IOB unit. For OpenSPARC T2, it is in the NCU.

7.3.2 Interrupt/Trap Vector Dispatch Register

"A thread may write to the following register to trigger an interrupt to another thread. This is intended to support interrupts from the TAP during bring up. In addition, any thread may be sent a reset [trap interrupt] via this register.

12.3.5.2 How OpenSPARC T2 Starts its Virtual Cores at Reset

1. For OpenSPARC T2, the E-Fuse Cluster scans out the E-Fuse Array to set ASI_CORE_AVAILABLE. NCU uses ASI_CORE_AVAILABLE to set ASI_CORE_ENABLE and ASI_CORE_ENABLE_STATUS. When the TCU finishes BIST, NCU can then unpark one virtual core, by setting one bit of ASI_CORE_RUNNING.
2. That virtual core then starts each of the others by unparking them, by setting the other bits in ASI_CORE_RUNNING.

The first time each SPARC core sees its bit of ASI_CORE_RUNNING change to 1, it does a POR trap.

"IMPL. DEP. #114: The RED_state trap vector is located at an implementation-dependent address referred to as RSTVaddr.

"Implementation Note: The RED_state trap handlers should be located in trusted memory, for example, in ROM. The value of RSTVaddr may be hard-wired in an implementation, but it is suggested that it be externally settable, for instance by scan, or read from pins at power-on reset." OpenSPARC T2 does not implement RSTVaddr as a register, so it is not settable.

"The RED_state trap vector address (RSTVaddr) is 256 MB below the top of the virtual address space; this is, at virtual address FFFF FFFF F000 0000_{16}, which is passed through to physical address FF F000 000016 in RED_state."

“Following the state initialization process, the TCU [NCU] instructs the machine (via the Trap Unit) to begin fetching and executing instructions at the RSTVaddr || 0x20.... These values may be changed by the system controller, if present, during reset.” The system controller cannot change these values during reset.

12.3.6 XIR

OpenSPARC T2 accepts a signal on its external BUTTON_XIR_ pin, and sends a packet for each virtual core enabled by the ASI_XIR_STEERING register. (By contrast, OpenSPARC T1 received XIR from a write to the INT_VEC_DIS register and did not use the ASI_XIR_STEERING register.)

“Used to... gain control of a chip. This corresponds to the L1-A key combination on Sun machines, or Ctl-Alt-Delete on a PC”.

"7.1.3.1 Soft XIR

"By setting bit[1] in the Reset_Gen Register of Tomatillo, a processor can generate an XIR. The Reset_Source register logs the cause of an XIR so that the XIR trap handler can easily identify the source.

"Reset due to XIR does not initiate fetch of initialization code from Boot PROM, and the memory controller continues to perform refresh cycles in order to preserve main memory contents."

The trap handler may initiate a reset, so it is not a reset like the others. Thus, XIR only involves the CMP and SPG units, and not RST, except to the extent that RST may debounce and synchronize the signal from the external input pin, and Or it with the XIR_GEN bit in the RESET_GEN register.

The FPGA debounces BUTTON_XIR_, so OpenSPARC T2 does not need to.

1. OpenSPARC T2 implements the CMP Programming Model as defined. OpenSPARC T1 implemented XIR as a hypervisor function, whereas OpenSPARC T2 will do this in hardware as specified in the CMP Programming Model." [OpenSPARC T1 has an ASI register accessible from code and JTag which initiates XIR on a per thread basis.]
2. OpenSPARC T2 has a pin for XIR. (OpenSPARC T1 did not have one.)" [OpenSPARC T1 did not implement XIR via Tomatillo. Tomatillo can generate it but OpenSPARC T2 will ignore the resulting JBus transaction (this is different from all other JBus implementations). The only off-chip way to cause an XIR on OpenSPARC T1 is via JTag. The idea is that, as JTag has access to the on-chip CSRs, it can poke the XIR bit as if it was written to by a thread. OpenSPARC T2 could choose to do the same, but that may break the CMP Programming Model.]
3. The way the OpenSPARC T2 cores handle XIR as a trap allows restart.
4. The XIR CMP config. register is ASI and JTag accessible
5. Application note: If a debug engineer wants to use the feature 'XIR a particular thread', they will need to implement a debug JTag test setup which can dynamically modify the XIR steering register, and make sure their POST code handles this correctly.

12.3.6.1 JTag can cause XIR

OpenSPARC T2 has a scannable flip-flop that can cause XIR, so JTag can cause XIR. "A yet-to-be-specified JTag command could cause an XIR to be steered through the XIR_STEERING register. Since they are OR'ed the first to happen would cause the first XIR."

12.3.7 WDR

"Watchdog reset (WDR) is a V9-defined trap. WDR can be initiated via an event (such as taking a trap when TL == MAXTL) which causes an entry into the V9 error state - the processor immediately generates a watchdog reset trap to take the core to RED_state. On N2, a WDR also can result from a fatal error condition detected by on-chip error logic. A WDR only affects the strand which created it. When a WDR is recognized, instruction fetching begins at RSTVaddr || 0x40."

12.3.7.1 Tomatillo SouthBridge System_watchdog Timer Signal

The Tomatillo SouthBridge system_watchdog timer signal differs from the CMP watchdog reset, WDR.

From the Tomatillo Programmer's Reference Manual:

> "7.1.3.2 Button XIR
>
> "For bring-up purposes, the system supports a Button XIR. This reset is triggered through a push button which is connected to SouthBridge and is OR'ed with the system_watchdog. This button is physically located on a dongle which is attached to the motherboard through a header connector.
>
> "The Button XIR feature is designed to facilitate bring-up and to provide an easy way to get the system out of software hang through an XIR instead of a general system reset. This allows the system to preserve most of its state and in particular the contents of all registers in the I/O subsystem. It can prove to be useful in identifying problems when the system hangs on I/O transfers.
>
> "The Button XIR signal is 'OR'ed' with the SouthBridge watchdog timer signal (inside the southbridge), and the result is connected to Tomatillo s input. When either the Button XIR or the watchdog signal is active Tomatillo generates an XIR transaction to all processors. Bit[5] of the Reset Source register is set to one when a watchdog or Button XIR is generated. This allows the trap handler to identify the cause of the XIR."

12.3.7.2 CMP Watchdog Reset, WDR

From the CMP Programming Model:

"The only resets that are limited to a single virtual core are the resets internally generated by a virtual core.... for current SPARC processors, these are the Software Initiated Reset, SIR, and the watchdog reset, WDR. These types of resets are generated by an individual virtual core and are not propagated to the other virtual cores on a CMP."

12.3.8 XIR, WDR, and SIR Perform No Reset

WDR and SIR are internally generated by a virtual core and are limited to a single virtual core. They are thread-specific and not propagated to other cores or TCU . They are independent of RST.

In conclusion, of the concepts in the above table, only POR, DBG_INIT, WMR, and NIU involve the reset unit. “Other reset types [XIR, WDR, SIR] are called reset for historical reasons, but they do not actually perform a reset. Their actual behavior is that of a Non-Maskable Interrupt (Trap) with fetch from PROM, TL = 2.”

12.4 Machine State after Each Kind of Reset

TABLE 12-12 uses 0 as a shorthand to mean that each unit in this portion of the chip will revert to the initial state defined by the Programmer’s Reference Manual.

TABLE 12-12 Machine State

	JTag portion of TCU	WMR-protected portion (Note 2)	WMR-protected part of DMU, PEU	WMR-exposed part of DMU, PEU	NIU	Rest of chip
TRST_	0	0 (Note 1)	0 (Note 1)	0 (Note 1)	0 (Note 1)	0 (Note 1)
POR	Stable	0	0	0	0	0
WMR	Stable	Stable	0	0	0	0
DMU_PEU bit	Stable	Stable	Stable	0	Stable	Stable
NIU bit	Stable	Stable	Stable	Stable	0	Stable
DBG	Stable	Stable	Stable	Stable	Stable	0

Notes:

1. A table entry of 0 indicates that a unit outside of the JTag portion of TCU is reset by TRST_ implicitly, because of the requirement that “the system must assert both TRST_ and PWRON_RST_L to properly reset the part.”

2. The Programmer’s Reference Manual defines the subset of the chip unchanged by WMR, and includes: integer registers, floating-point registers, TBA, Y, PIL, CWP, CCR, ASI, CANSAVE, CANRESTORE, OTHERWIN, CLEANWIN, WSTATE, FSR,

FPRS, TICK_CMPR, VA_WATCHPOINT, I/D/L2 tags and data, L2 directory, iTLB/DTLB, Sparc Error Status, Sparc Error Address, L2 Error Status, L2 Error Address, MCU Error Status, MCU Error Address, and all IO Error registers.

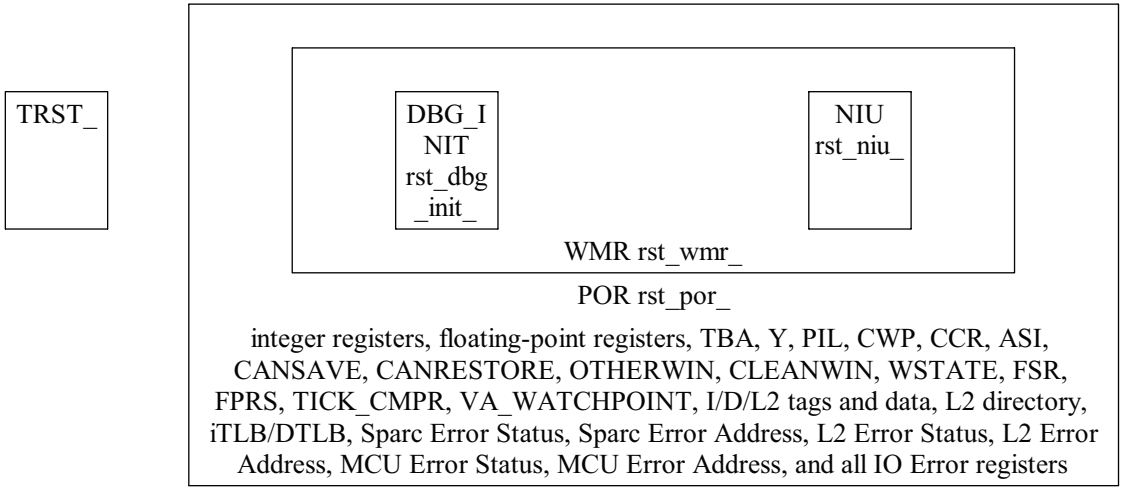
TABLE 12-13 Cleared Arrays

	Arrays cleared by WMR	Arrays cleared by BISI on WMR	Flops cleared on WMR	Flops cleared on DBG_INIT [change to DBG]
L2	Dcdir Icdir	VUAD-UA, VUAD-VD	All state machines. (CSRs not yet defined.)	None.
SIU	None.	None.	All. (No error logs in SIU.)	None.
DMU, PEU	None.	None.	All state machines. Some CSRs will be cleared, and some not.	None.
MCU	None.	None.	All except: MCU Error Status MCU Error Address	Refresh, scrub, & arbiter (1 bit now) state machines.
NCU	None.	None.	All except: ASI_CORE_AVAILABLE ASI_CORE_ENABLE ASI_CORE_ENABLE_STATUS ASI_XIR_STEERING	None.

12.4.1 Venn Diagram

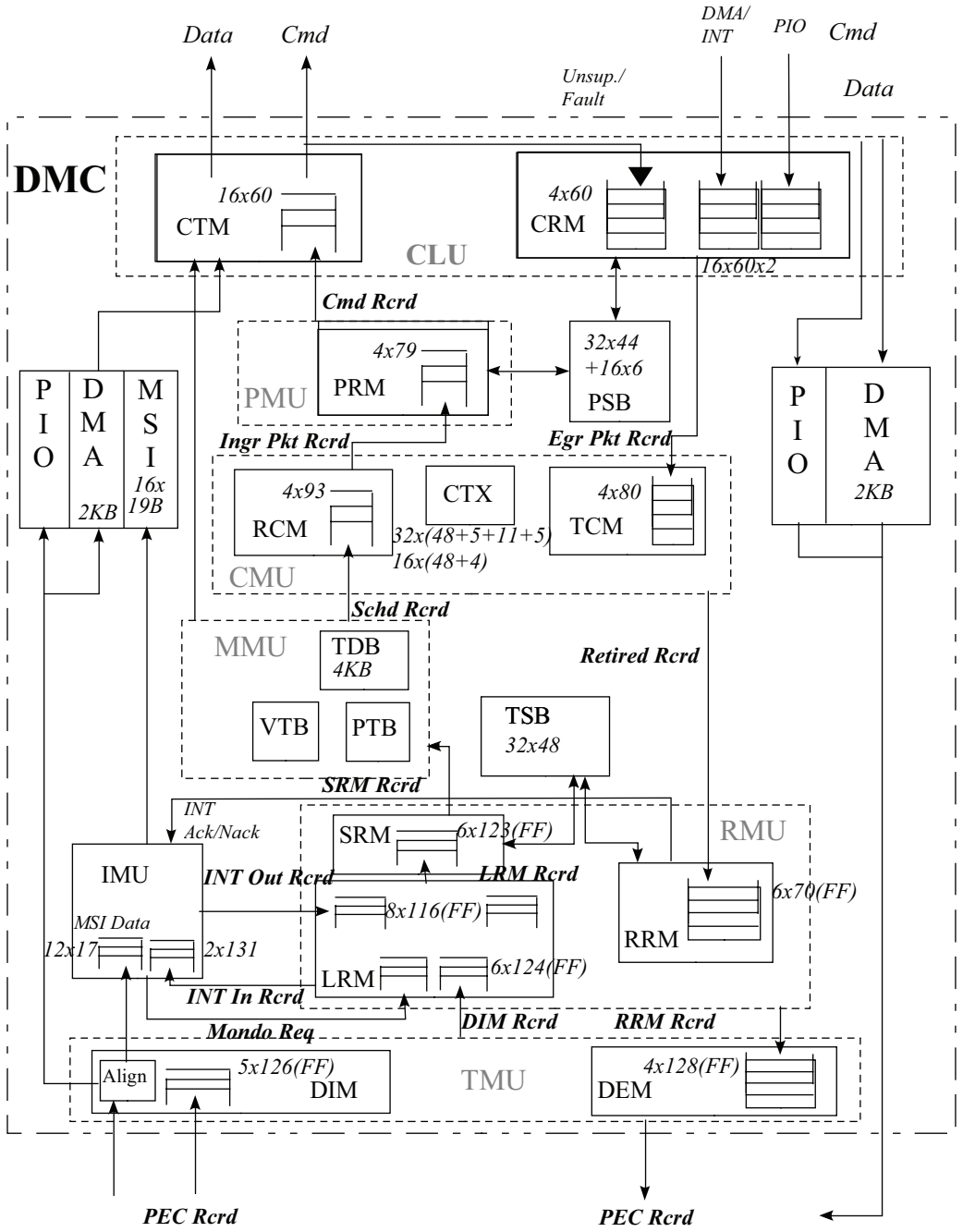
Rectangles in FIGURE 12-1 represent regions of the chip affected by each kind of reset. For example, POR resets all flops in the chip, except for those reset by TRST_. Parts of the chip affected by WMR are also reset by POR. Registers cleared by DBG_INIT are also cleared by WMR and POR.

FIGURE 12-1 Venn Diagram



12.4.2 Reset Signals Asserted for each Kind of Reset

FIGURE 12-2 Reset Signals



The Reset Unit resets the PLL within the CCU with rst_ccu_pll_ when it is locking.

(The PLL calls this signal pll_arst_l.) Thus, the Reset Unit resets it during POR1, and also during WMR1 if ccu_rst_change == 1. Once the PLL has locked to its new frequency, there is no need to reset it again during WMR2.

The Reset Unit asserts rst_ccu_ to reset the CCU only during POR1. This is one of the main differences between POR1 and POR2. It never resets the CCU during either WMR1 or WMR2.

Each of the next six signals is one of a pair, with a por version and a wmr version. Notice that in FIGURE 12-2, if the Reset Unit asserts the por member, it will also assert the wmr version, so that during POR, it resets both the WMR-exposed and the WMR-protected flops of a cluster.

The signals rst_l2_por_ and rst_l2_wmr_ differ from the others in that the L2 cache clusters are reset by flush reset, in which these two signals play no part. Rather, they are inputs to L2 cache intellectual property which had been reset by synchronous reset, and these two inputs remain.

The Reset Unit will reset the MAC, rst_niu_mac_, during POR1 and POR2. It will also reset it during WMR1 and WMR2 if ccu_rst_change == 1, but software can suppress this last event by setting to one the MAC_PROTECT bit in the SSYS_RESET register. FIGURE 12-2 shows two waveforms for rst_niu_mac_, one for MAC_PROTECT == 0, and another for MAC_PROTECT == 1. When the Reset Unit resets the MAC during POR1, it continues to assert rst_niu_mac_ for NIU_TIME after the TCU has deasserted the Asic clk_stop signals.

During Subsystem Reset, the Reset Unit will treat the NIU as if it were performing a Warm Reset when ccu_rst_change == 1. Thus it will reset the MAC by asserting rst_niu_mac_, by default. Software can suppress this by setting to one the MAC_PROTECT bit in the SSYS_RESET register.

To keep the link from going down while we apply reset to the NIU, software should do the following:

1. Program MAC tx_enable and rx_enable to zero. MAC will do a graceful shut-down, meaning it will stop transactions at a packet boundary.
2. Wait for some time to let the NIU enter a quiescent state.
3. Set to one the MAC_PROTECT bit and issue an NIU Subsystem Reset.

The Reset Unit will reset the other three NIU blocks, RTX, TDS, and RDP, by asserting rst_niu_wmr_ during POR, WMR, and NIU Subsystem Reset.

If rst_l2_wmr_ resets a flip-flop, then WMR will clear it, as will POR.

12.4.3 POR Clears the Valid Bits in the L2T Directory of L1 Tags CAM

To guarantee coherency and correct functionality, initialize the following before enabling the L2 cache:

TABLE 12-14 Initialize Arrays

Structure	Initialize	Approximate size
tag array	parity bits	28 kilobyte * 8
VUAD array	valid bits	140 bits * 32 * 8
directory CAM	valid bits	15 bits * 32 * 16 * 2
data array	no initialization	500 kilobyte * 8

BISI or ASIs are used to initialize the tag array with good parity.

BISI or ASIs are used to initialize the VUAD arrays by clearing all the valid bits.

Once we enable the L2 cache, it will generate parity for each directory entry written and check it when it reads it out, including the valid bit. However, directory hits are independent of parity. If there's a hit in a directory CAM, it sends a packet across the crossbar, even with the L2 cache disabled. To prevent such spurious hits and packets upon power-up, a signal at the time of Power-On Reset immediately clears all the directory valid bits. This leaves the parity bits uninitialized, but parity will be set later, by BIST, by ASIs, or by the L2 cache in operation after it is enabled. Should the L2 cache detect a parity error at any time, it logs the event and issues an interrupt.

L2 needs to be informed of three things:

1. L2 lines are invalid.
2. L1 lines are invalid (directory in L2).
3. L2 tag array parity and valid bits are cleared.

L2 lines are invalidated with BIST or BISI instructions issued to VUAD array.

L1 lines are invalidated using immediate reset. This is straightforward. There is already logic in the CAM which resets the valid bit when the corresponding entry is a hit. Hence this clearing of the valid bit is just a logic OR of the immediate reset input and the valid bit reset logic in the currently existing logic.

```
libs/n2sram/cams/
    n2_com_cm_64x64_cust_l/
    n2_com_cm_64x64_cust/rtl/
    n2_com_cm_64x64_cust.sv:
```

```
cam_hit0[63] = (wr_data[12:0]== addr_array_63[12:0]);
cam_hit1[63] = ((wr_data[13]== addr_array_4[63]) | force_hit);
cam_hit [63] = (cam_hit0[63] & cam_hit1[63]) & valid_bit[63];
```

The above mechanism prevents spurious packets dispatched to CCX. We wish to prevent such packet, even if the Sparc cores are all parked, because L2 will retry.

BIST or BISI sets the L2 tag array with good parity and valid bits.

The L2 directories are in l2t. The Reset Unit has to assert the clear pin (rst_l2_por_?) for 1 or 2 clock cycles. The clocks have to be running.

Each SRAM has a register at its input that is scanned and clocked at the rising edge of the clock. It is followed by a latch that is not scanned and clocked at the falling edge of the clock. The latch needs a clock edge to reset. The Reset Unit asserts rst_l2_por_ to reset the latches.

```
/l2sat_top/cpu/l2t0/ dc_row0/panel0  /array/valid_bit[63:0]
/l2sat_top/cpu/l2t7/             ic_row2/panel3
/array/valid_bit[63:0]
/l2sat_top/cpu/l2t[bank#]/[cache#]c_row[row#]
/panel[panel#]/array/valid_bit[63:0]
```

8 banks	data cache, instruction cache	2 rows	4 panels
l2t[bank#]	[cache#]c_	_row[row#]	panel[panel#]
0 bank# 7	dc cache# ic	0 row# 2	0 panel# 3
l2t0	dc_	row0	panel0
l2t1	ic_	row2	panel1
l2t2			panel2
l2t3			panel3
l2t4			
l2t5			
l2t6			
l2t7			

L2 initializations:
During POR_:
---

1) The directories should be initialized before L2 cache is enabled to guarantee coherency and correct functionality. The directory valid bits are cleared (flash clear) during POR_ [rst_l2_por_].

=> When the valid bits are cleared (not valid) then the entries are don't-care. Hence, the parity bits does not need to be initialized to good parity.

Clearing valid bits in the directory informs the L2 cache that there are no valid lines in L1.

BISI or ASI's are used to initialize:

1. The VUAD arrays by clearing all the VUAD bits and ecc associated with it. **This informs L2 cache that there are no valid lines in L2.**
2. The tag array with good parity. This eliminates the possibility of any error cases from happening. (False/true hits and misses)
3. The data array is initialized to good ecc+clean data eliminate any kind of false error detection.

Reverse directories valid bits will be clear up by synchronous rst_l2_por_. This ensures no pointers to L1 lines. L2 valid bits in VUAD array are reset by flush reset only. L2 LRU initialization is achieved by using rst_l2_por_ to set the all LRU entries to way 0.

In summary L2 uses a combination of flush reset and synchronous reset.

Before a core is turned off, all lines in the caches need to be cleared. If core enable or bank enable status is changed, then L1 and L2 caches need to be flushed by running BISI. If they are not changed, then you do not have to run BISI.

TABLE 12-15 shows the cache lines in Table 13-17, "CPU State After Reset and in RED_state", on page 179 of the OpenSPARC T2 Programmer's Reference Manual, Revision 1.2, April 3, 2006:

TABLE 12-15 CPU State after Reset

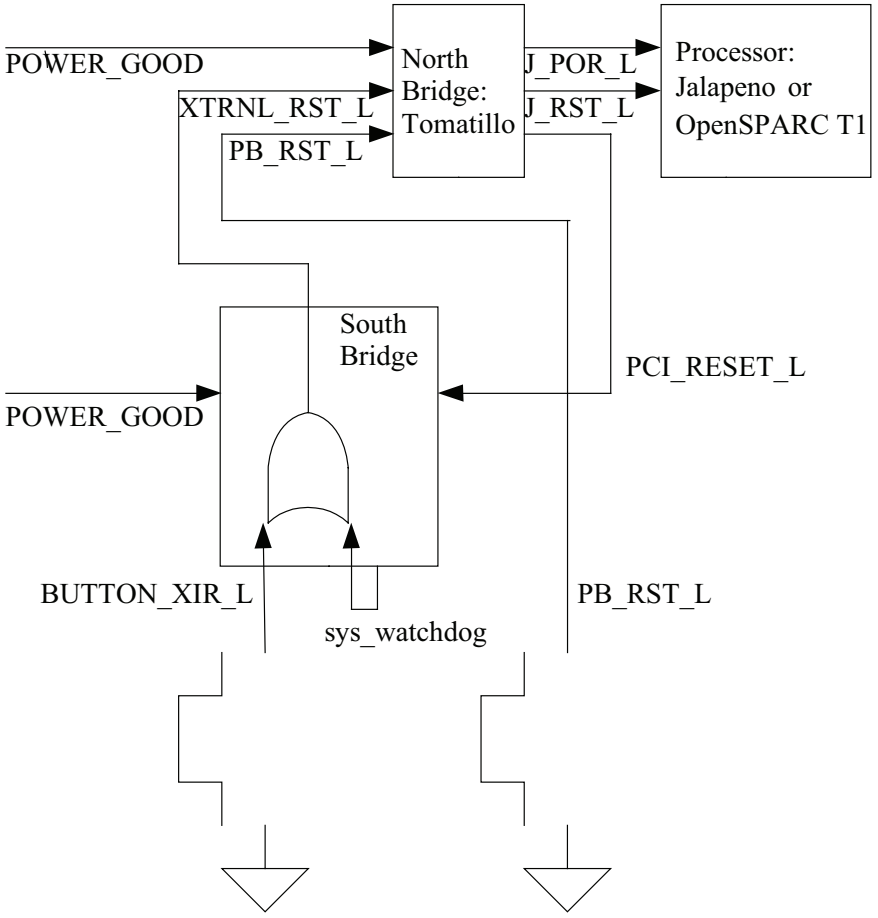
Structure that holds state	POR	WMR
I/D cache tags	All invalid	Unchanged if BISI not run, else invalid
L2 tags and data	Unknown	
L2 directory	All invalid	

12.5 OpenSPARC T2 is a System On a Chip

12.5.1 System On a Board

FIGURE 12-3 shows a possible configuration for a Jalapeno or a OpenSPARC T1 processor in a system on a board. An external NorthBridge chip such as Tomatillo supplies the processor with J_POR_L and J_RST_L.

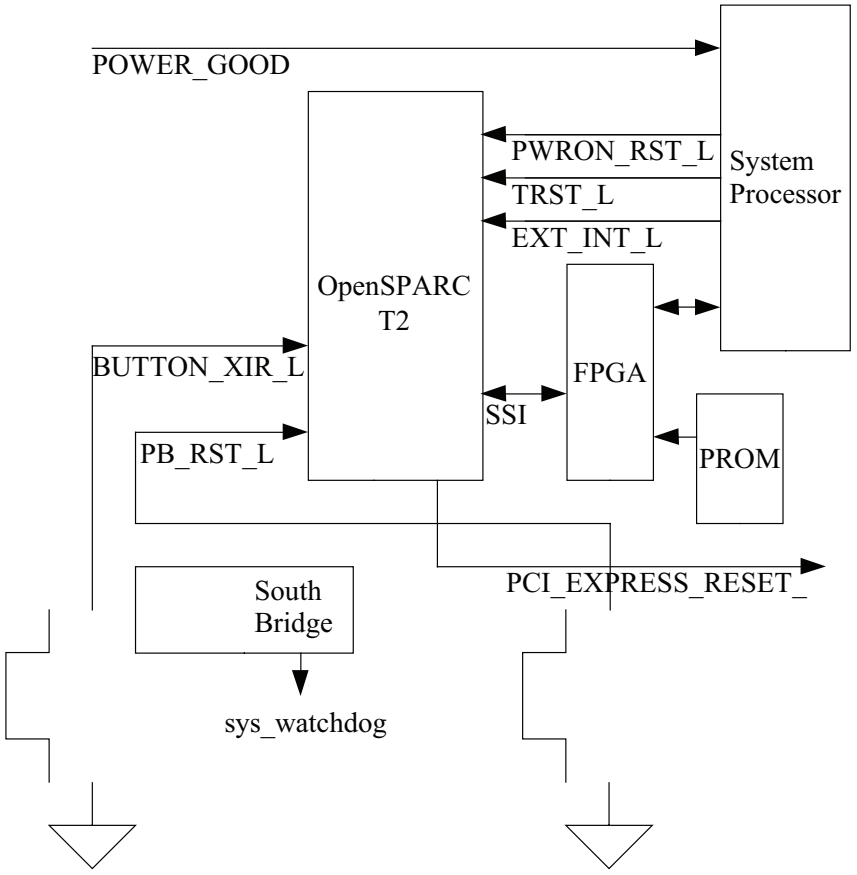
FIGURE 12-3 System On a Board



12.5.2 System On a Chip

FIGURE 12-4 shows the external reset connections for a OpenSPARC T2 system on a chip. The N-One initiative says there will be a service processor for any platform. "Addressable power devices that allow for software-based powering up or down of any device in the data center."

FIGURE 12-4 System On a Chip



12.5.3 Serial System Interface, SSI

"The Serial System Interface (SSI) is defined to allow microprocessors to access peripherals in a low-pin-count fashion. The OpenSPARC T2 chip will not directly interface to peripherals but instead will provide a interface that can be easily converted to peripheral protocols by an external Programmable Logic Device (PLD). Isolating the OpenSPARC T2 chip from these peripherals allows the devices to use higher voltage signalling and provides a mechanism for protocol conversion.

"... OpenSPARC T2 will always be the master of the bus."

"Addresses within the SSI address range (0xFF_F000_0000 to 0xFF_FFFF_FFFF) are issued to the off-chip SSI interface bus. The only transactions that are supported directly to the SSI interface are:

1. 1, 2, 4, 8 Byte aligned Reads

1. 1, 2, 4, 8 Byte aligned Writes

"SSI generates interrupts for two reasons: either the EXT_INT_L pin was asserted, or an error was detected. The external interrupt pin is intended to be used by the FPGA, and has NO ordering protection, meaning when EXT_INT_L is asserted, an interrupt is issued to the IOB, without checking any transactions in flight. The interrupt is delivered to the IOB using the SSI device ID, i.e. (device ID == 2)."

"Current implementation of the SSI interface for N2 has two issues:

1. "During reset (power_on or warm or debug), the SSI_SCK and SSI_MOSI wiggle over time and then settle to zero during flush. (SSI_SCK and SSI_MOSI are driven by NCU which gets flush reset). This causes the SSI CLK PLL in the FPGA in the system to see spurious transactions on SSI_MOSI and also an unstable SSI_SCK, eventually followed by the SSI_SCK to go to zero for several microsecs. Since the FPGA uses the SSI_SCK as one if its ref clocks, it loses lock with the SSI_SCK.

2. "When the SSI_SCK starts to run again after the flush, N2 sends out the first boot fetch after only a few cycles from the time of the unparking of the threads. This does not provide the FPGA enough time to lock against the SSI_SCK and hence the FPGA would not be able to service the request properly. Based on the datasheet from Xilinx, the FPGA PLL would require around 3 msec of time for the PLL to relock against the SSI_SCK.

"To solve these two issues, it has been agreed upon amongst system folks and N2 design team that N2 needs to indicate to the FPGA on a pin when it should ignore the SSI_SCK and SSI_MOSI outputs from N2 during reset, and instead hold the FPGA PLL in reset. The Reset unit would assert this new signal called SSI_SYNC_L on power-on, and keep asserting it until it unparks the threads to NCU. Then it would deassert it, indicating to the FPGA that it can deassert the reset to its SSI_CLK PLL and start locking against SSI_SCK coming from N2. By this point N2 would be driving the SSI_SCK properly and the PLL would get around 5 to 6 msec to lock before NCU would assert the first SSI_MOSI.

"Since we are short of functional pins, it has been agreed upon that the FATAL_ERR pin would be renamed to this SSI_SYNC_L pin. The Service Processor would extract fatal error information from the chip by reading on-chip registers if required."

Specific timing requirements for rst_mio_ssi_sync_l:

Deassert on power-up.

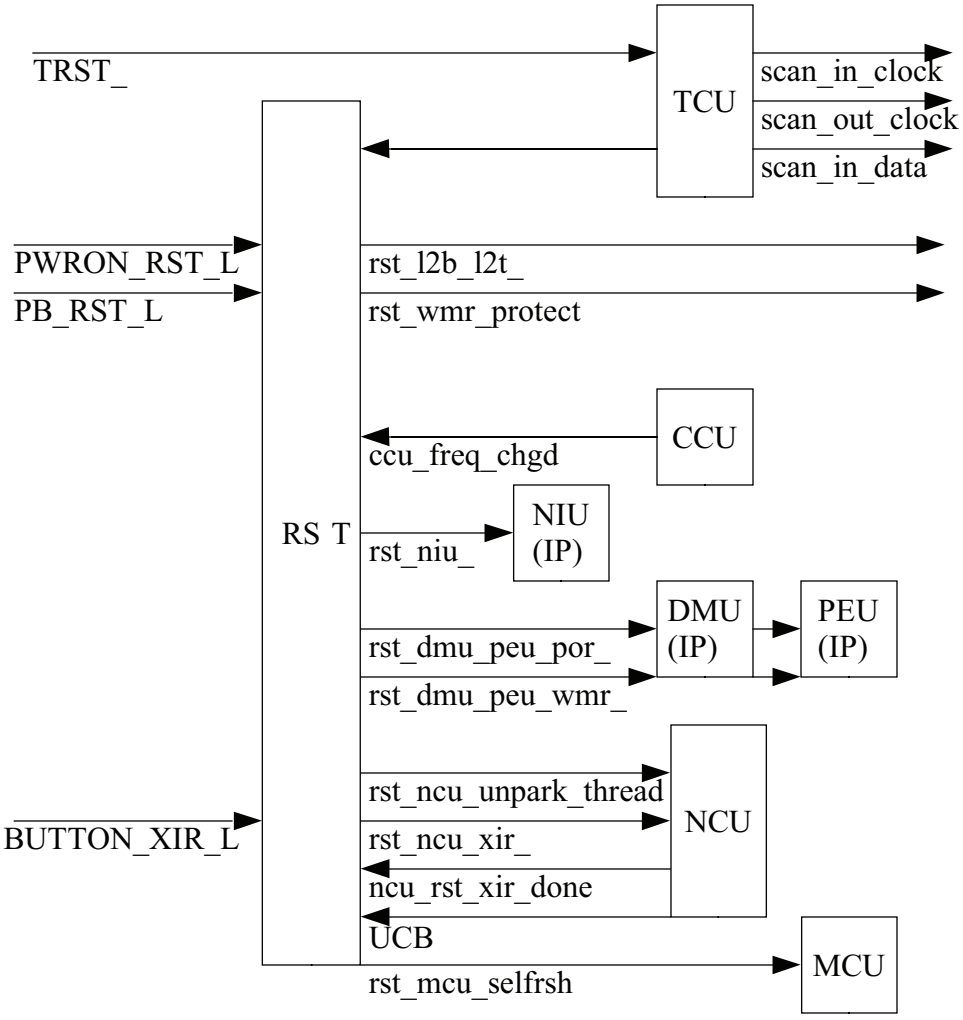
Assert after flush reset, but before rst_ncu_unpark_thread.

Deassert before flush reset of NCU.

12.5.4 Connections between RST and Other Clusters

FIGURE 12-5 shows some connections between RST and other clusters. See the Reset Unit Verification Test Plan for a more complete depiction.

FIGURE 12-5 Connections between RST and Other Clusters



12.6 Registers

12.6.1 (0x89-0000-0808) Reset Generation Register, RESET_GEN

This register allows software to generate resets. It is a copy of the Fire Reset Generation register, Reset_Gen, with one exception. Since the service processor drives PWRON_RST_L, the OpenSPARC T2 RESET_GEN register does not implement the POR_GEN bit that Fire has in bit position 2.

Write 1 to only one of the bits in this register at a time.

TABLE 12-16 Reset Generation Register

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:4	0	RO	Reserved
DBR_GEN	3	0	RW	Write 1 to cause a Debug Reset. This is the same as Warm Reset, except that PCI Express and NIU keep running. Enters Fig. 7 at WMR2. Set by software, cleared at completion of DBR.
RSVD1	2	0	RW	Reserved. (Was POR_GEN on fire, indicating that software wrote 1 to cause a Power-On Reset.)
XIR_GEN	1	0	RW	Write 1 to cause an eXternally-Initiated Reset. Set by software, cleared at completion of XIR.
WMR_GEN	0	0	RW	Write 1 to cause a Warm Reset. Enters Fig. 7 at WMR1. Set by software, cleared at completion of WMR.

Note that the *Fire Programmer's Reference Manual* calls Soft Reset a Power-On Reset. Section 1.4.2, entitled "Power-On Reset (Soft Reset)", says, "When power is already on, if the 'PB_RST_L' input to Fire gets asserted due to a push-button trigger in the system, Fire initiates a soft reset... When power is already stable and the processor detects a transition on its J_RST_L input pins, it takes a 'Soft Reset'. It is simply called a Power-On Reset in this document. It is similar to a 'Hard Reset' except that the on-chip memory controller continues to perform refresh cycles in order to preserve main memory contents, and clock ratio is unaffected." Thus, the *Fire Programmer's Reference Manual* calls bit 0 of Reset_Gen "PO_RST". This can cause confusion when speaking about both Fire in a OpenSPARC T1 or OpenSPARC T2 context.

The *Fire Programmer's Reference Manual* appears internally inconsistent, however. Section 1.4.5 is entitled "POR & Warm Reset Initialization", implying that POR and Warm Reset are two, different things.Also, the Fire Power-On, Reset and BIST document says, "18. OBP checks Reset_Source Register. a. if bit3 (Power_On)..."But the *Fire Programmer's Reference Manual* calls bit 3 "PU, Power_up (Low to High transition on Power_Good)." It's bit 0 that the *Fire Programmer's Reference Manual* calls "PO_RST". Reset Source Register, RESET_SOURCE

This register allows software to identify the origin of a reset. It is a copy of the Fire Reset_Source register.

12.6.2 (0x89-0000-0818) Reset Source Register, RESET_SOURCE

This register allows software to identify the origin of a reset. It is a copy of the Fire Reset Source register.

TABLE 12-17 Reset Source Register

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:16	0	RO	Reserved
L2T7_FATAL	15	0	RW1C	The L2T7 cache detected a fatal error, causing a WMR.
L2T6_FATAL	14	0	RW1C	The L2T6 cache detected a fatal error, causing a WMR.
L2T5_FATAL	13	0	RW1C	The L2T5 cache detected a fatal error, causing a WMR.
L2T4_FATAL	12	0	RW1C	The L2T4 cache detected a fatal error, causing a WMR.
L2T3_FATAL	11	0	RW1C	The L2T3 cache detected a fatal error, causing a WMR.
L2T2_FATAL	10	0	RW1C	The L2T2 cache detected a fatal error, causing a WMR.
L2T1_FATAL	9	0	RW1C	The L2T1 cache detected a fatal error, causing a WMR.
L2T0_FATAL	8	0	RW1C	The L2T0 cache detected a fatal error, causing a WMR.
NCU_FATAL	7	0	RW1C	One of the clusters feeding the NCU detected a fatal error, causing a WMR.
PB_XIR	6	0	RW1C	An external agent asserted the BUTTON_XIR_ input pin.
PB_RST	5	0	RW1C	WMR: An external agent asserted the PB_RST_L input pin, causing a WMR.
PWRON_RST	4	1	RW1C	The System Processor asserted the PWRON_RST_L input pin.
DBR_GEN	3	0	RW1C	Software wrote 1 to the DBR_GEN bit of the RESET_GEN register to cause a Warm Reset.
RSVD1	2	0	RW1C	Reserved. (Was POR_GEN on Fire, indicating that software wrote 1 to the POR_GEN bit of the RESET_GEN register to cause a Power-On Reset.
XIR_GEN	1	0	RW1C	Software wrote 1 to the XIR_GEN bit of the RESET_GEN register to cause an eXternally-Initiated Reset.
WMR_GEN	0	0	RW1C	Software wrote 1 to the WMR_GEN bit of the RESET_GEN register to cause a Warm Reset.

RW1C – Read, Write 1 to Clear: Writing a 0 to a bit in this field has no effect, but writing a 1 to a bit in this field will cause that bit to be set to 0.

The Reset Unit only recognizes an eXternally-Initiated Reset if it is processing no other reset, since XIR has the lowest priority of the resets that the Reset Unit handles. Thus, if an external agent asserts the BUTTON_XIR_L input pin, the Reset Unit will set the PB_XIR bit of the RESET_SOURCE register only when it completes any earlier reset. If software writes to the XIR_GEN bit of the RESET_GEN register, when the Reset Unit starts to process it, it will set the XIR_GEN bit of RESET_SOURCE.

The Reset Unit will clear a bit in the RESET_GEN register upon completion of the corresponding reset. In the RESET_SOURCE register, by contrast, software can clear a bit, but not the Reset Unit. It can only set a bit.

If software sets the XIR_GEN bit of the RESET_GEN register, and any other reset occurs while the Reset Unit is waiting for the NCU to finish processing the XIR, the Reset Unit will leave the XIR_GEN bit set.

12.6.3 (0x89-0000-0838) Subsystem Reset Register, SSYS_RESET

This register allows software to reset a particular subsystem.

For the NIU, the minimum reset width needs to cover TI serdes PLL lock up time (which is 3 s)plus some extra time for synchronous reset to propagate through various clock domain. A 10 us reset with should be good enough. (The NIU also has registers within it that allow software to reset portions of the NIU.)

TABLE 12-18 Subsystem Reset Register

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:7	0	RO	Reserved
MAC_ PROTECT	6	0	R/W	Set to one to suppress the assertion of rst_niu_mac_ that the Reset Unit would normally generate during a WMR with ccu_rst_change==1.
MCU_ SELFRSH	5	0	R/W	Set to one to have the MCUs put the DRAM info self-refresh. (Drives clspine_mcu_selfrsh to the MCU.)
RSVD1(Was: MCU_FBD_PRO TECT)	4	0	R/W	Reserved (Was: When 0, the FBDIMM interface logic in MCU will get reset as usual on Warm Reset and Debug Reset. When 1, this FBDIMM interface logic will not be reset on Warm Reset and Debug Reset and will continue functioning as normal.) (Now use self refresh.)

TABLE 12-18 Subsystem Reset Register

RSVD2	3:2	0	RO	Reserved
DMU_PEU	1	0	RW	Write 1 to send a warm reset to the PCI-Express subsystem (DMU and PEU), both ingress and egress, for at least 15 s.Cleared by hardware at completion.
NIU	0	0	RW	Write 1 to send a warm reset to the NIU for at least 4 s. Cleared by hardware at completion.

12.6.4 (0x89-0000-0810) Reset Status Register, RSET_STAT

"In order to enable or disable a functional unit's clocks, a number of L1 clock headers must be fed from the same enable signal. N2 SPG may use a "rolling enable", where possible, which follows the pipeline structure within the unit, which helps with I/dt noise on the power supply".

"Reset Status Register, RSET_STAT

"RegisterBaseAddress 1 IOBMAN – 0x98-0000-0000

"The chip reset status, shown in TABLE 12-19 is maintained for all chip-wide reset and power management commands. The reset source bits in this register are writable to allow software to clear them after the chip reset sequence is complete, in order for thread warm resets to be distinguished from chip resets. HW will copy the current reset status into a shadow status whenever a reset occurs."

"Register64 34 Chip Reset Status Register – RSET_STAT (0x89_0000_0810)"

TABLE 12-19 Reset Status Register

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:12	0	RO	Reserved
FREQ_S	11	0	R/W	Shadow status of FREQ
POR_S	10	0	R/W	Shadow status of POR
WMR_S	9	0	R/W	Shadow status of WMR
RSVD1	8 - 5	0	RO	Reserved
RSVD2	4	0	RO	Reserved

TABLE 12-19 Reset Status Register

FREQ	3	0	R/W	Set to one if the reset is a warm reset that changed frequency.
POR	2	1	R/W	Set to one if the reset is from PWRON_RST_L pin.
WMR	1	0	R/W	Set to one if the reset is from: (1) the PB_RST_L input pin, (2) the WMR_GEN bit of the RESET_GEN register, (3) from a Fatal Error, or (4) the DBR_GEN bit of the RESET_GEN register. Enter Fig. 7 at WMR1.
RSVD2	0	0	RO	Reserved

The shadow versions of the bits only have meaning after a WMR, since by definition, a reset the system controller applies after the machine has been running is a WMR. Since the system controller only applies a POR upon applying power, the shadow versions of the bits then will always be 0.

12.6.5 (0x89-0000-0820) Fatal Error Enable Register, RESET_FEE

Each bit of this register allows the l2t*n*_rst_fatal_error signal, 0 n 7, from one of the l2t banks, to cause a Warm Reset. If the respective Fatal Error Enable bit is set, and the corresponding error type is asserted, the Reset Unit will cause a Warm Reset. (The NCU contains a register named Fatal Error Enable, FEE. That register enables a fatal error to cause NCU to assert the signal ncu_rst_fatal_error to the Reset Unit.)

TABLE 12-20 Fatal Error Enable Register

Field	Bit Position	Initial Value	R/W	Description
RSVD0	63:16	0	RO	Reserved
L2T7_FEE	15	0	R/W	The L2T7 cache detected a fatal error, causing a WMR.
L2T6_FEE	14	0	R/W	The L2T6 cache detected a fatal error, causing a WMR.
L2T5_FEE	13	0	R/W	The L2T5 cache detected a fatal error, causing a WMR.
L2T4_FEE	12	0	R/W	The L2T4 cache detected a fatal error, causing a WMR.
L2T3_FEE	11	0	R/W	The L2T3 cache detected a fatal error, causing a WMR.
L2T2_FEE	10	0	R/W	The L2T2 cache detected a fatal error, causing a WMR.

TABLE 12-20 Fatal Error Enable Register

L2T1_FEE	9	0	R/W	The L2T1 cache detected a fatal error, causing a WMR.
L2T0_FEE	8	0	R/W	The L2T0 cache detected a fatal error, causing a WMR.
RSVD1	7:0	0	RO	Reserved

12.6.6 (0x89-0000-0860) Clock Control Unit Time Register, CCU_TIME

TABLE 12-21 Clock Control Unit Time Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:16	0	RO	Reserved
CCU_TIME	15:0	32_{10}	R/W	CCU_TIME

The value in this register determines the length of two intervals.

1. CCU_TIME determines the interval from when the Reset Unit deasserts rst_ccu_ until it deasserts cluster_arst_l and rst_tcu_clk_stop. This interval must be long enough for the CCU to have begun generating the sync_en pulses. (Historical note: At some point in its operation, the CCU starts to count to 24 and then asserts an internal signal named ccu_rst_sync_stable. The sync_en pulses are stable well before the CCU asserts ccu_rst_sync_stable. The Reset Unit cannot make use of ccu_rst_sync_stable during this interval, because at first the CCU has not yet begun to drive sync_en pulses, so the Reset Unit cannot observe it.)
2. CCU_TIME also determines the interval from when the Reset Unit deasserts cluster_arst_l and rst_tcu_clk_stop, until it asserts rst_tcu_flush_stop_req. The TCU requires some time with its clocks running until it expects to receive rst_tcu_flush_stop_req.

The default value is 32 sys_clk cycles.

12.6.7 (0x89-0000-0870) Lock Time Register, LOCK_TIME

TABLE 12-22 Lock Time Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:16	0	RO	Reserved
LOCK_TIME	15:0	5120_{10}	R/W	LOCK_TIME

We need the reset sequence to be repeatable and deterministic in time for the tester function. Thus feedback from PPL locks and the E-Fuse Cluster is not desirable. It is better to have a predetermined time configured by software. Also, the pre-WMR boot code may wish to perform Warm Reset with the same PLL config. register values, obviating the need to wait for the l2clk PLL to lock.

The value in this register determines the length of time that the Reset Unit asserts rst_wmr_ while various phase-locked loops lock. The Reset Unit uses this register twice in the Power-On Reset Sequence:

1. Starting when the system controller deasserts PWRON_RST_L. During this time, the E-Fuse Cluster scans the E-Fuse Array, and the ddr_pll and NIU PLLs lock. [E-Fuse now occurs later in the sequence.]

2. Starting when the pre-WMR boot code writes a 1 into the CHIP_RESET register. During this time, the above two PLLs lock, and potentially the l2clk PLL locks as well.

Since the PLL config. register might change during WMR, the LOCK_TIME register cannot use l2clk. It must use the system clock.

Reset causes this register to take on the longest time needed, assuming the highest planned reference clock frequency. The longest time needed is the maximum of the time required for the following:

1. 10 sto lock the NIU PLL.

2. 25 sto lock the l2clk PLL.

“System clock is fed into the ref_clk of the cmp PLL. Internal to the PLL, the clock is multiplied up to the VCO frequency of 3 GHz.” The worst case in this context is the highest frequency contemplated for sys_clk, 200 MHz, with a period of 5 ns.

lock time in cycles= 25 s 5 ns/cycle = 5,000 cycles

Thus, the initial value for this register is 5k = 5,120.

12.6.8 (0x89-0000-0880) Propagation Time Register, PROP_TIME

TABLE 12-23 Propagation Time register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:16	0	RO	Reserved
PROP_TIME	15:0	3072_{10}	R/W	PROP_TIME

This register indicates how long it takes for the longest scan chain to flush. After the Reset Unit receives tcu_rst_flush_init_ack, it will wait PROP_TIME pll_sys_clk clock cycles before asserting rst_tcu_flush_stop_req.

Reset causes this register to take on the longest time needed, assuming the highest planned reference clock frequency. The longest time needed is the maximum of the time required for the following:

1. The scan chain to flush.
2. The MAC requires at least 4,000 ns. See Section 4.10, "NIU".

One way to estimate item (1) is to derive a back-of-the-envelope guess for the delay for each stage of the flush reset. We do this by summing up:

(1) the mid-table delay value of si-to-siclk setup, and

(2) the soclk-to-so clock-to-q delay,

which gives 250 ps. We must consider this estimate within certain limitations, as follows:

1. The actual flow through delay arc (si to so), when both latch stages are open, will be different. How much, we don't know. The setup number reflects a failure point number which doesn't necessarily relate to actual delay path condition during flush.
2. Scan paths are a weird mix of back-to-back flops and repeated interconnects. We might guess that the portion of gate-dominated delay is quite high though. We might suppose an _average_ total interconnect (wire + repeater) delay of 30ps. This could be way off.

The physical group will eventually use static timing analysis to more rigorously verify the overall delay. For the moment, the above considerations provide an initial average number as a starting point.

Based on an estimate of 250 ps per stage, the time for flush reset to propagate through any one scan chain would be the number of flops in the chain times 250 ps. OpenSPARC T2 has approximately 1,000,000 flops in 32 scan chains. With reasonable balancing, we expect the longest chain will be 45,000 to 50,000 flops long.

Flush reset time= 50,000 flops 250 ps/flop
= 12,500 ns
= 12.5 s

Flush reset time in cycles= 12,500 ns 5 ns/cycle
= 2500 cycles

Thus, the initial value for this register is 3k = 3,072, providing a 22 percent margin over the value needed for the longest chain.

Since the physical team provided this initial estimate, the library team has provided timing information, as follows:

```
/import/n2-
librel/integration/release/rel_1.4/lib/c021a/cl_sc1/compiled/cl_sc1
.SynT
  pin(so) {
    direction  :     output;
    connection_class   : universal;
     timing() {
      related_pin : "si";
      timing_sense : positive_unate;
      cell_fall(table6_7) {
       index_1 ("   0.009073, 0.018110, 0.027590, 0.049080, 0.123400,
0.245500");
       index_2 ("   0.000500, 0.001000, 0.001500, 0.003500, 0.005000,
0.015000, 0.030000");
        values ( \
"0.145700,0.148100,0.150300,0.158000,0.163200,0.196900,0.247000",\
"0.147500,0.149900,0.152100,0.159800,0.165100,0.198700,0.248900",\
"0.149600,0.152000,0.154200,0.161900,0.167200,0.200800,0.251000",\
"0.154200,0.156600,0.158800,0.166500,0.171800,0.205400,0.255600",\
"0.168900,0.171300,0.173500,0.181200,0.186500,0.220100,0.270300",\
"0.191900,0.194400,0.196500,0.204300,0.209600,0.243200,0.293300");
      }
      cell_rise(table6_7)
       index_1 ("   0.008461, 0.017230, 0.026730, 0.048000, 0.123300,
0.249000");
       index_2 ("   0.000500, 0.001000, 0.001500, 0.003500, 0.005000,
0.015000, 0.030000");
        values ( \
"0.138800,0.141600,0.144100,0.152800,0.158700,0.196800,0.253600",\
"0.141600,0.144300,0.146800,0.155500,0.161400,0.199500,0.256300",\
"0.144700,0.147400,0.149900,0.158600,0.164500,0.202600,0.259400",\
```

```
"0.151200,0.153900,0.156400,0.165100,0.171100,0.209100,0.265900",\
"0.169000,0.171700,0.174200,0.182900,0.188800,0.226900,0.283700",\
"0.192200,0.195000,0.197500,0.206200,0.212200,0.250200,0.307000");
      }
```

If we take the entries in the middle column and the middle rows of the cell_fall table and the cell_rise table, we obtain the following values, in nanoseconds:
0.161900
0.166500
0.158600
0.165100

If we now retain our original estimate of 250 ps per stage, we provide a 50 percent margin over the slowest of these values, in addition to the 22 percent margin we already provided.

12.6.9 (0x89-0000-0890) NIU Time Register, NIU_TIME

TABLE 12-24 NIU Time Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:16	0	RO	Reserved
NIU_TIME	15:0	1600_{10}	R/W	NIU_TIME

This register indicates how long it takes for initial values to shift throughout the NIU.

NIU time= 8 s

NIU time in cycles= 8,000 ns 5 ns/cycle
= 1,600 cycles

Thus, the initial value for this register is 1.5k + 64 = 1,600.

Note: The Reset Unit must assert rst_mio_pex_reset_l for 15 s,so before Warm Reset, software must manipulate LOCK_TIME, PROP_TIME, and NIU_TIME to provide at least this duration of this signal. For example, the Subsystem Reset of the DMU-PEU makes use of NIU_TIME twice (15 8 + 8 s).The Subsystem Reset of the NIU also uses NIU_TIME (4 8 s).

12.7 Power-On Reset Sequence Overview

This section summarizes what the following sections lay out in detail.

TABLE 12-25 shows the major types OpenSPARC T2 structures that hold state (see Section 3.4). It also shows, after each stage of the Power-On Reset sequence, whether each structure:

- holds an unknown value, "X",
- has been reset to 0,
- has been initialized from the E-Fuse Unit, "efu",
- has taken on a value, either 0 or 1, that is deterministic and repeatable, "det".

TABLE 12-25 Structures that Hold State

	pwr-good = 0	POR 1	EFU 1	BISI 1	POR 2	EFU 2	Pre-WMR boot code	WMR 1	BIST 2	WMR 2	Post-WMR boot code
WMR-protected flops	X	0	det	X	0	det	det	det	det	det	det
WMR-exposed flops	X	0	det	X	0	det	det	0	det	0	det
SRAM repair latches	X	X	efu	efu	efu	efu	efu	efu	efu	efu	efu
SRAM array core contents	X	X	X	0	0	0	det	det	0	0	det
core available flops	X	0	efu	efu	0	efu	efu	efu	efu	efu	efu

The only actions required by the system controller are to:

1. assert TRST_ and PWRON_RST_L,
2. start sys_clk and the DMU-PEU and NIU Serdes clocks,
3. deassert TRST_, and then

4. deassert PWRON_RST_L. (Or, deassert TRST_ and PWRON_RST_L simultaneously.)

The Reset Unit will automatically take OpenSPARC T2 through the POR1 through the unpark_thread that fetches the pre-WMR boot code.

The typical OpenSPARC T2 powerup reset sequence is as follows:

1. On powerup of the system (shown in Figure 5 as Off-chip "pwr_good" = 0), the system controller asserts PWRON_RST_L and RST, assisted by CCU and TCU, asserts all other reset signals. This causes (1) the internal state of all SunV clusters to reset, including all control registers and memory refresh state machines, (2) causes IO outputs to reset, and (3) protects the internal tristate muxes. In addition, as soon as the system controller applies sys_clk, the Asic clusters will reset. The Reset Unit will hold in reset the CCU PLL during this time. The other PLLs, in the NIU SERDES and the PEU SERDES, by contrast, will be locking to their frequencies as soon as the system controller applies sys_clk.

2. Once power is up in the system (pwr_good = 1), the system controller then deasserts PWRON_RST_L. The Reset Unit hold most of OpenSPARC T2 in Power-On Reset (POR1) while the CCU PLL locks, waits while the EFuse Controller reads out the EFuse Array (EFU1), and waits while the TCU performs BISI (BISI1). Since the SRAM outputs are enabled during BISI, their initial, unknown state may transfer to flip-flops that had been reset during POR1. To correct this, the Reset Unit applies Power-On Reset a second time (POR2). The second POR resets information in the NCU that had come from the EFuse Array, so the EFuse Controller reads out the EFuse Array a second time (EFU2). Then the cpu fetches reset configuration programming code from the boot PROM where configuration registers (clock ratios, etc.) are programmed (Pre-WMR boot code). RST must deassert all reset signals simultaneously and synchronously to their respective clocks.

3. The boot code modifies the frequency ratio register, and then causes a Warm Reset. The Reset Unit resets most of OpenSPARC T2 (everything except error logs) while relocking the CCU PLL (WMR1). Then it waits while the TCU performs BIST (BIST2), performs a second Warm Reset (WMR2), and restarts instruction fetch of boot code running at the reprogrammed clock ratio (Post-WMR boot code).

4. Subsequent warm resets may take place later via rst_wmr_, which do not disturb states which are reset only by PWRON_RST_L. Any of the resets (POR, WMR, or DBR) may be caused by a write to a RST CSR, CHIP_RESET. Warm resets may also be generated with a system push-button.

5. After we wrote the above, we added a reset after BISI, and another after BIST. After POR1, BISI might change some state, so POR2 resets all flops. After WMR1 comes BIST, then WMR2. Note that BISI is required before turning on the cache.

12.7.1 Power-On Reset Duration in a System

TABLE 12-26 sums up the duration of each step of the Power-On Reset sequence, in which OpenSPARC T2 is in a system:

TABLE 12-26 Power-On Reset Sequence Duration

POR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
Strt POR sequence	Sys ctlr	-	POWER_GOOD	-	-	0.00
PLL resets	Sys ctlr	POWER_GOOD	Deassert Pwron_Rst_L	-	-	2,000.00
por1, pll locks	rst	Deassert rst_ccu_pll_	Deassert rst_ccu_	LOCK_TIME = 5,000	5.000	25,000.00
sync_stable	ccu	Deassert rst_ccu_	ccu_rst_sync_stable	5	0.714	3.57
Deassert Asic clk_stop	tcu	ccu_rst_sync_stable	tcu_rst_flush_stop_ack	2*128 = 256	0.714	182.86
niu pll	rst	tcu_rst_flush_stop_ack	tcu_rst_flush_stop_req	NIU_TIME = 1,600	5.000	8,000.00
5.4.8: Deassert SunV clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	22*128 = 2,816	0.714	2,011.43
5.4.9: efu1	efu	tcu_rst_flush_stop_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14
5.5: bisi1	tcu	tcu_rst_efu_done	tcu_bisx_done	l2d:128k + l2t:8k + l2vuad: 1k = 140,288	0.714	100,205.71
clk_stop	tcu	tcu_rst_flush_init_req	tcu_rst_flush_init_ack	24*128 = 3,072	0.714	2,194.29
por2	rst	tcu_rst_flush_init_ack	tcu_rst_flush_stop_req	PROP_TIME = 3,000	5.000	15,000.00
5.8: Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
5.10: efu2	efu	tcu_rst_flush_stop_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14
5.12	ncu	rst_ncu_unpark_thread	core_running	4	2.857	11.43

TABLE 12-26 Power-On Reset Sequence Duration *(Continued)*

	spc	core_running	Request from spc to ncu	10-15	0.714	10.71
SSI pll locks	ncu	-	-	3FFFF = 256k = 262,144	11.428	2,995,931.42
5.12 +	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End POR sequence	-	-	-	-	-	3,175,439.98

12.7.2 Power-On Reset Duration on a Tester

Second, we consider the case in which the part is on the tester, not in a system, so there is no need for an external PLL to synchronize to the SSI.

TABLE 12-27 sums up the duration of each step of this minimal-delay Power-On Reset sequence:

TABLE 12-27 Power-On Reset Duration on Tester

POR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
Strt POR sequence	Sys ctlr	-	POWER_GOOD	-	-	0.00
PLL resets	Sys ctlr	POWER_GOOD	Deassert Pwron_Rst_L	-	-	2,000.00
por1, pll locks	rst	Deassert rst_ccu_pll_	Deassert rst_ccu_	LOCK_TIME = 5,000	5.000	25,000.00
sync_stable	ccu	Deassert rst_ccu_	ccu_rst_sync_stable	5	0.714	3.57
Deassert Asic clk_stop	tcu	ccu_rst_sync_stable	tcu_rst_flush_stop_ack	2*128 = 256	0.714	182.86
niu pll	rst	ccu_rst_sync_stable	tcu_rst_flush_stop_req	NIU_TIME = 1,600	5.000	8,000.00
5.4.8: Deassert SunV clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	22*128 = 2,816	0.714	2,011.43
5.4.9: efu1	efu	tcu_rst_flush_stop_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14

TABLE 12-27 Power-On Reset Duration on Tester *(Continued)*

5.5: bisi1	tcu	tcu_rst_efu_done	tcu_bisx_done	l2d:128k + l2t:8k + l2vuad: 1k = 140,288	0.714	100,205.71
clk_stop	tcu	tcu_rst_flush_init_req	tcu_rst_flush_init_ack	24*128 = 3,072	0.714	2,194.29
por2	rst	tcu_rst_flush_init_ack	tcu_rst_flush_stop_req	PROP_TIME = 3,000	5.000	15,000.00
5.8: Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
5.10: efu2	efu	tcu_rst_flush_stop_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14
5.12	ncu	rst_ncu_unpark_thread	core_running	4	2.857	11.43
	spc	core_running	Request from spc to ncu	10-15	0.714	10.71
SSI pll locks	ncu	-	-	0	11.428	0.00
5.12 +	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End POR sequence	-	-	-	-	-	179,508.56

12.7.3 Warm Reset Duration in a System

We now consider the case in which OpenSPARC T2 is in a system, and software has:

1. Configured the MBIST engines to perform MBIST, and
2. Configured the Clock Control Unit to change to a new frequency.

TABLE 12-28 sums up the duration of each step of this maximum-delay Warm Reset sequence:

TABLE 12-28 Maximum Delay Warm Reset Sequence

WMR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
Start WMR sequence	soft ware	WMR_GEN bit	tcu_rst_flush_init_req	-	-	0.00
clk_stop	tcu	tcu_rst_flush_init_req	tcu_rst_flush_init_ack	24*128 = 3,072	0.714	2,194.29

TABLE 12-28 Maximum Delay Warm Reset Sequence

wmr1, reset pll	rst	Assert rst_ccu_pll_	Deassert rst_ccu_pll_	PROP_TIME = 3,000	5.000	15,000.00
pll locks	rst	Deassert rst_ccu_pll_	Deassert rst_ccu_	LOCK_TIME = 5,000	5.000	25,000.00
sync_stable	ccu	Deassert rst_ccu_	ccu_rst_sync_stable	5	0.714	3.57
9.2: Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
9.5: bist2	tcu	tcu_rst_efu_done	tcu_bisx_done	(128k + 8k + 1k)*8*21 = 23,568,384	0.714	16,834,560.00
9.6: clk_stop	tcu	tcu_rst_flush_init_req	tcu_rst_flush_init_ack	24*128 = 3,072	0.714	2,194.29
wmr2	rst	tcu_rst_flush_init_ack	tcu_rst_flush_stop_req	PROP_TIME = 3,000	5.000	15,000.00
9.8: Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
9.12	ncu	rst_ncu_unpark_thread	core_running	4	2.857	11.43
	spc	core_running	Request from spc to ncu	10-15	0.714	10.71
SSI PLL locks	ncu	-	-	3FFFF = 256k = 262,144	11.428	2,995,931.42
5.12 +	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End WMR sequence	-	-	-	-	-	19,894,314.27

12.7.4 Warm Reset Duration on a Tester

Finally, we consider the case in which the part is on the tester, not in a system, so there is no need for an external PLL to synchronize to the SSI. Also, software has:

1. Configured the MBIST engines to skip MBIST, and
2. Configured the Clock Control Unit to retain the same frequency.

TABLE 12-29 sums up the duration of each step of this minimal-delay Warm Reset sequence:

TABLE 12-29 Minimum Warm Reset Duration

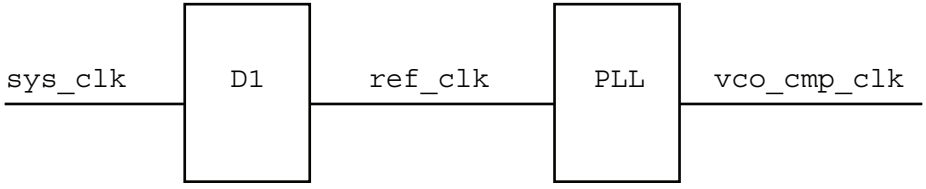
WMR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
Start WMR sequence	soft ware	WMR_GEN bit	tcu_rst_flush_init_ req	-	-	0.00
clk_stop	tcu	tcu_rst_flush_ini t_req	tcu_rst_flush_init_ ack	24*128 = 3,072	0.714	2,194.29
wmr1, reset pll	rst	Assert rst_ccu_pll_	Deassert rst_ccu_pll_	PROP_TIME = 3,000	5.000	15,000.00
pll locks	rst	Deassert rst_ccu_pll_	Deassert rst_ccu_	LOCK_TIME = 0	5.000	0.00
sync_stable	ccu	Deassert rst_ccu_	ccu_rst_sync_stabl e	5	0.714	3.57
9.2: Deassert clk_stop	tcu	tcu_rst_flush_sto p_req	tcu_rst_flush_stop _ack	24*128 = 3,072	0.714	2,194.29
9.5: bist2	tcu	tcu_rst_efu_done	tcu_bisx_done	0	0.714	0.00
9.6: clk_stop	tcu	tcu_rst_flush_ini t_req	tcu_rst_flush_init_ ack	24*128 = 3,072	0.714	2,194.29
wmr2	rst	tcu_rst_flush_ini t_ack	tcu_rst_flush_stop _req	PROP_TIME = 3,000	5.000	15,000.00
9.8: Deassert clk_stop	tcu	tcu_rst_flush_sto p_req	tcu_rst_flush_stop _ack	24*128 = 3,072	0.714	2,194.29
9.12	ncu	rst_ncu_unpark_ thread	core_running	4	2.857	11.43
	spc	core_running	Request from spc to ncu	10-15	0.714	10.71
SSI PLL locks	ncu	-	-	0	11.428	0.00
5.12 +	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End WMR sequence	-	-	-	-	-	38,822.85

12.8 Deterministic Behavior

We require that any sequence of actions on OpenSPARC T2 be deterministic, and so repeatable. Thus, the relative alignment of the Ratioed Synchronous Clocks, cmp, dr, io, and io2x, must be identical following any two identical resets.

The RSCs follow a pattern that repeats with a period equal to or less than the period of the reference clock. The system clock drives the input to the divider D1 (divide by 2), which in turn drives ref_clk. Since ref_clk has a period twice that of sys_clk, the RSCs also repeat during every sys_clk cycle.

FIGURE 12-6 Clock Cycles



The alignment of the RSCs will be the same if the Reset Unit drives its outputs at the same time relative to ref_clk. The outputs of the Reset Unit will be the same, relative to ref_clk, if the inputs are the same, relative to ref_clk.

The events that can initiate a reset are:

1. The FPGA (asserts and) deasserts the PWRON_RST_L chip input pin.
2. The FPGA asserts the PB_RST_L chip input pin.
3. L2 asserts an l20_rst_fatal_error-l27_rst_fatal_error signal.
4. NCU asserts ncu_rst_fatal_error signal.
5. Software sets the WMR_GEN bit in the RESET_GEN register.
6. Software sets the DBR_GEN bit in the RESET_GEN register.
7. Software sets the NIU bit in the SSYS_RESET register (only resets NIU).
8. Software sets the PIU bit in the SSYS_RESET register (only resets PIU).

For Power-On Reset, the Reset Unit deasserts rst_ccu_pll_ on the rising edge of sys_clk. This signal will release the PLL's D1 flop from reset. That will determine the relative phase relationship between the sys_clk and ref_clk, and in turn, between

sys_clk and the RSCs, so every Power-On Reset will be deterministic. By the way, this sequence of events will also occur during a Warm Reset in which the frequency changes.

The l20_rst_fatal_error-l27_rst_fatal_error signals originate in the cmp_clk domain. ncu_rst_fatal_error comes from the io_clk domain. Software sets each CSR bit through the UCB interface to the NCU, running at io_clk. Thus, all the other events that can initiate a reset, except for PB_RST_L, come across a Clock Domain Crossing. The ccu asserts each sync_en signal only once during each ref_clk period, so every reset, except for PB_RST_L, will be deterministic.

The FPGA might assert PB_RST_L in time for the synchronizer to register it on a sys_clk during the first half of a ref_clk cycle, or it might assert it during the second half. These two cases would cause the Reset Unit to drives its outputs at different times relative to the RSCs. To eliminate this possibility, after synchronizing PB_RST_L to sys_clk, the Reset Unit retimes it to the cmp_clk domain, then retimes it back to sys_clk again. Since the ccu asserts each sync_en signal only once during each ref_clk period, every reset due to PB_RST_L will be deterministic.

12.9 Power-On Reset Sequence

This is the sequence we envision a machine in normal use would follow. During debug, an engineer may choose to forgo some steps, such as the Warm Reset.

FIGURE 12-7, FIGURE 12-8, FIGURE 12-9, and FIGURE 12-10 show the entire Power-On Reset sequence. A number, 1 through 9.8, indicates a step in the sequence and corresponds to the description of a step in this section. A solid arrow from one step to another indicates that the completion of the first step causes the second to begin. By contrast, there are three dashed arrows:

1. From the assertion of POWER_GOOD to step 5
2. From step 5 to step 5.4
3. From step 5.2 to step 5.4, via step 5.3

These indicate not cause and effect, but rather the ordering of these steps that the System Processor will impose.

There are five headings at the top of the figure:

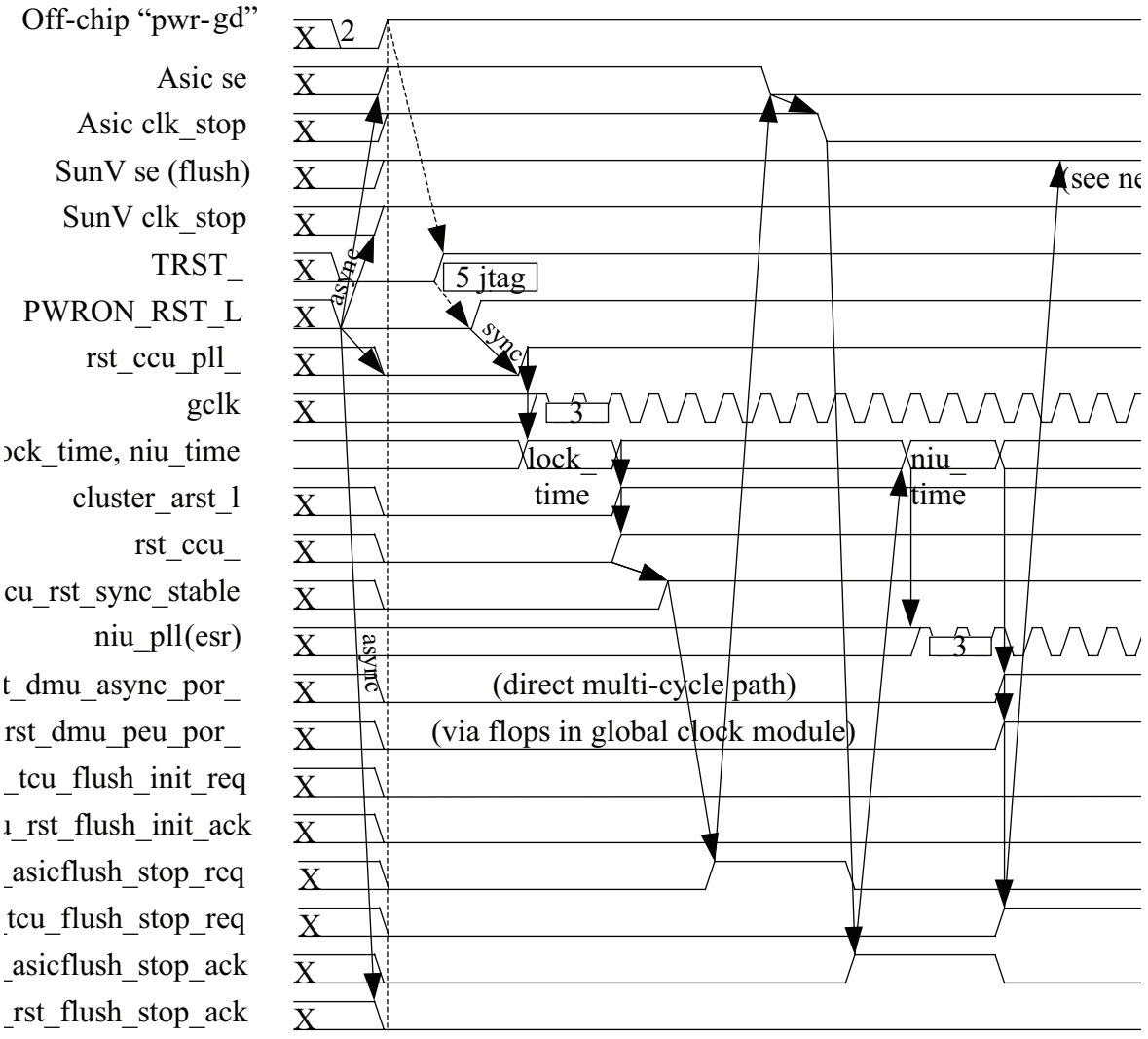
1. During PWRON_RST_L (including POR 1)
2. After PWRON_RST_L (including POR 2)
3. Pre-WMR boot code

4. During WMR WMR 1
5. After WMR 2
6. Post-WMR boot code

12.9.1 During PWRON_RST_L (including POR1)

1. Service processor asserts TRST_ and PWRON_RST_L.

FIGURE 12-7 Power-On Reset Sequence - Start of POR1



a. “A TRST_ event will initialize the PLL config. register to a default setting.” “Note that the TCU BIST config. registers as well as the TCU BIST result register will not be reset by the flush reset function; these registers will be reset only by a TRST_ event.” The default setting is for a divisor of 11 [now 8], the lowest supported divisor, corresponding to the lowest supported frequency. For pll_sys_clk = 133.33 MHz, the cmp_freq is 733.33 MHz, for 166.67 it is 916.67, and for 400 it is 1100.

b. “L2 Directory (of L1 tags) is marked invalid on [Power-On] Reset.” A “flash clear” signal”with one or two clocks” clears the directory valid bits.

c. POR 1: RST asserts rst_por_, rst_wmr_, dbg_init_, rst_niu_, and PCI_EXPRESS_RESET_. These signals will directly reset the NIU, DMU, and PEU. Except for the Jtag registers and registers in the RST itself, the RST, in cooperation with the TCU, will flush-reset all flip-flops in the rest of the chip. RST and TCU cause reset "by asserting both the scan_in_clk and scan_out_clk simultaneously while driving logic 0 onto the scan_in_data of every scan chain." Figure 5 shows this as TCU asserting se, scan enable, during the interval labelled "POR 1".
The NIU has at least two PLLs. They will start oscillating when (1) power is applied and (2) rst_niu_ clears a bit in a control register. Once locked, they will stay locked, even if the NIU is subsequently reset again.

d. TCU needs to come out of POR1 with clk_stop asserted.

2. Power ramps up.(

3. PLLs start up, and clock tree = RClk (Regional Clock) = l2clk, iol2clk, and enl2clk start toggling.

4. For debugging, service processor may supply JTag portion of TCU with its own clock, TCK. This is not needed in production systems.

5. Service processor deasserts TRST_. "Once TRST_ is deasserted, registers in [the JTag portion of] the TCU may be accessed via the JTag TAP" while the service processor holds the rest of the chip in reset.

   a. Steps 5.2-5.10 below correspond to steps 9.2-9.10 in "During WMR" and "After WMR", below, starting with "PLLs lock." An exception is that the portions of step 5.4 which involve the EFU only occur during the POR portion.

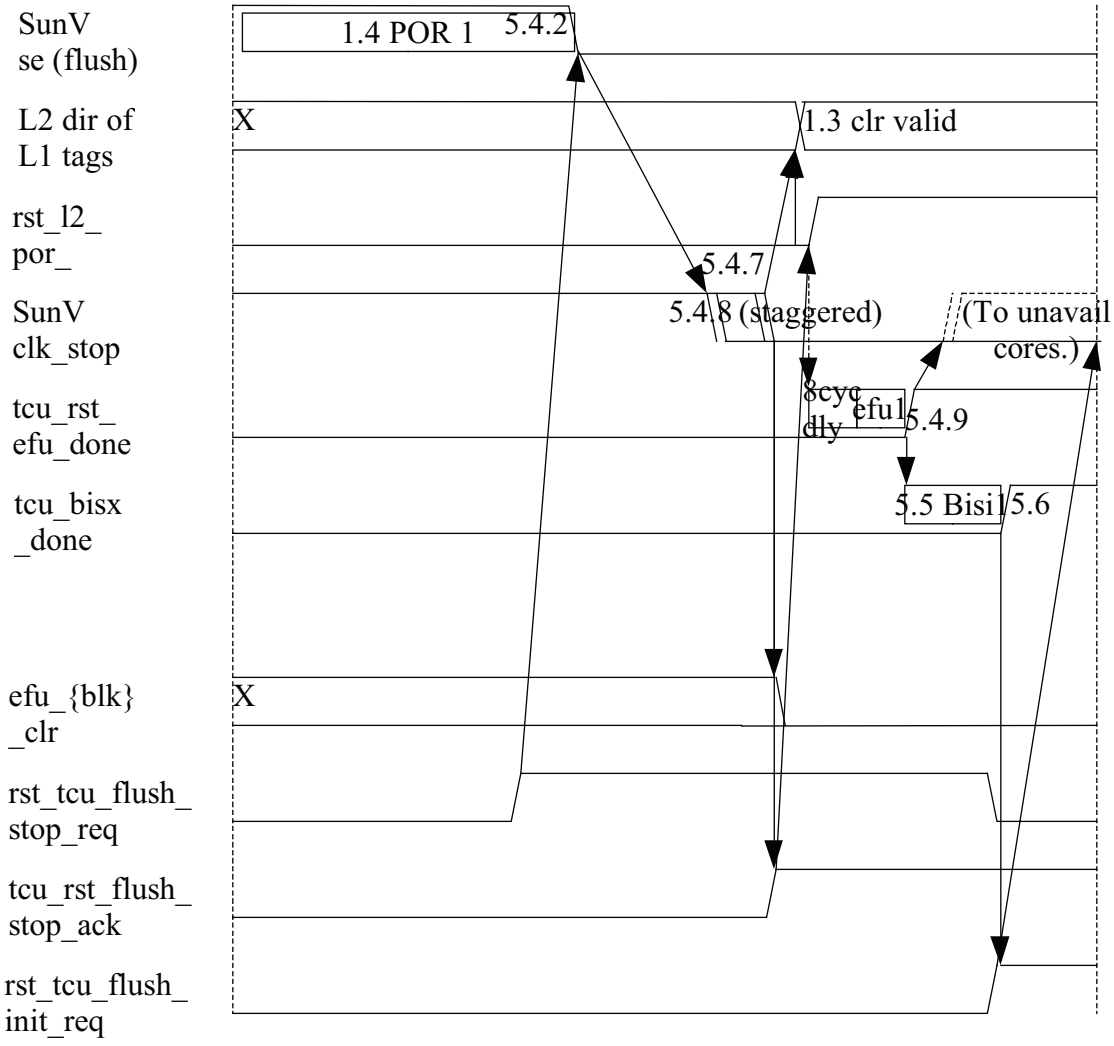
   b. PLLs lock.

   c. CCU asserts ccu_pll_locked and dr_pll_locked. These are analog signals, derived from l2clk, and independent of any reset signal and all other clock signals. Despite the existence of these signals, OpenSPARC T2 ignores them. Instead, we rely on the external signals PWRON_RST_L and PB_RST_L, or counting down the lock_time register, so that the chip's behavior is repeatable. OpenSPARC T2 does, however, provide ccu_pll_locked and dr_pll_locked as external output pins. We can use these pins to determine if it is safe to decrease lock_time
   "Upon power-up, the system must assert POR for a period of time sufficient to guarantee that power has stabilized and the on-chip PLL has locked."
   The interval from the time the system asserts the Tomatillo pwr_ok input, to the time Tomatillo deasserts J_POR_L, is 5 ms.
   The NIU needs 10 sfor its PLLs to lock.

12.9.2 After PWRON_RST_L (including POR2)

d. Service processor deasserts PWRON_RST_L. Note that it must deassert PWRON_RST_L at the same time as, or after, deasserting TRST_.

FIGURE 12-8 Power-On Reset Sequence - End of POR1



i. RST deasserts rst_por_ synchronous with the various clocks.

ii. TCU deasserts se. Deassertion of rst_por_ and se propagates.

iii. niu_pll locks. This must occur before the NIU starts.

iv. Optionally, on OpenSPARC T1, the service processor had asserted PB_RST_L, synchronous with PWRON_RST_L. If it did, then it now deasserts PB_RST_L synchronous with the reference clock. On OpenSPARC T2, this forces an alignment of the rising edges of the cmpclk, ioclk, and

ddrclk clocks. Since OpenSPARC T2 has two PLLs, there is no way to instantaneously force their outputs to align. The best we can do is to force a reset to the IO_d1 and IO2X_d1 logic, which generate io_r, io_f, io2x_r, and io2x_f in the CCU. In conclusion, deassert PB_RST_L upon power-up.

v. The Reset Unit waits for a number of cycles of the sys_clk. The Lock Time register holds that number. (This could be done by the service processor.) This allows the signals that had caused flush reset, such as se, time to propagate. When Lock Time has passed, the Reset Unit deasserts rst_wmr_, dbg_init_, and PCI_EXPRESS_RESET_L.

vi. TCU deasserts clock_stop to each of the 17 clock-stop domains in sequence. This deasserting in a staggered fashion minimizes di/dt. TCU must deassert clock_stop, even to a domain that will eventually have its bit in ASI_CORE_ENABLE set to zero, because the E-Fuse Unit needs its recipient's clocks enabled in order to communicate with it.
The valid bits in the L2 directory of L1 tags, the NIU, DMU, and PEU all require a clock edge to reset. The TCU asserts tcu_rst_flush_stop_ack to signal the Reset Unit that it has deasserted clk_stop. (The TCU will continue with the last two POR1 sequence steps, EFU1 and BISI1.) Now that L2T, NIU, DMU, and PEU have clocks, the Reset Unit has reset them and is able to deassert rst_niu_, rst_dmu_, and rst_peu_. It must do so before EFU1 starts, so that L2T and other SRAM headers can receive EFU data. The dotted line in Figure 5, from rst_l2t_ to efu1, represents this sequence requirement.

vii. EFU 1: E-Fuse Unit, EFU, scans out E-Fuse Array, EFA. EFU asserts efu_done after 700 [2816] io_clk clock cycles. EFU takes 44 io_clk cycles to scan out each of the 64 locations, so it takes:
44 cycles/location * 64 locations= 2816 io_clk cycles
2816 cycles * 2.857 ns/cycle= 8045.7 ns

e. BISI 1: "If at-[default-]speed [BISI or] BIST is desired," controlled by the service processor setting the TCU BISI or BIST registers in Step 5, then "TCU launches [BISI or] BIST on caches."
"L2 Tag, Data, and VUAD arrays, when BISTed to zeros, are initialized to empty with good parity and good ECC."
"L1 I-cache, L1 D-cache, when BISTed to zeros, initialized to good parity"
l2dtakes 128k (131,072)cmp_clk cycles
l2ttakes 8k (8,192)cmp_clk cycles
VUADtakes 256cmp_clk cycles
"When BIST is complete, the part will store the BIST results and flush reset the part in preparation to begin code execution."

f. TCU asserts tcu_bisx_done.

g. POR 2: BISI or BIST may have changed the state of some flip-flops connected to SRAM outputs. RST causes a second POR to reset those flops. Figure 5 shows this sequence as “POR 2”. RST asserts rst_por_ and rst_wmr_, and TCU asserts se.

h. After lock_time clock cycles, TCU deasserts se.

i. After a further lock_time clock cycles, RST deasserts rst_por_ and rst_wmr_. This allows the signals that had caused flush reset, such as se, time to propagate. See Section 7.6, "Propagation Time Register, PROP_TIME".
The valid bits in the L2 directory of L1 tags, the NIU, DMU, and PEU all require a clock edge to reset. The TCU asserts tcu_rst_flush_stop_ack to signal the Reset Unit that it has deasserted clk_stop. (The TCU will continue with the last two POR sequence step, EFU2.) Now that L2T, NIU, DMU, and PEU have clocks, the Reset Unit has reset them and is able to deassert rst_l2b_l2t_, rst_niu_, rst_dmu_peu_por_, and rst_dmu_peu_wmr_. It must do so before EFU2 starts, so that L2T and other SRAM headers can receive EFU data. The dotted line in Figure 5, from rst_l2t_ to efu2, represents this sequence requirement. The Reset Unit knows that EFU2 is done by tcu_rst_efu_done.

j. EFU 2: E-Fuse Unit, EFU, scans out E-Fuse Array, EFA. EFU asserts efu_done. This restores values that POR 2 cleared. Now that the E-Fuse Unit has communicated with its recipients, TCU can stop the clock in a clock-stop domain that has its bit in ASI_CORE_ENABLE set to zero. TCU conditionally reasserts clock_stop to each of the 17 clock-stop domains in sequence. This reasserting in a staggered fashion minimizes di/dt.
The EFA sets first the Core Available register, then the NCU copies this to the Core Enable register.

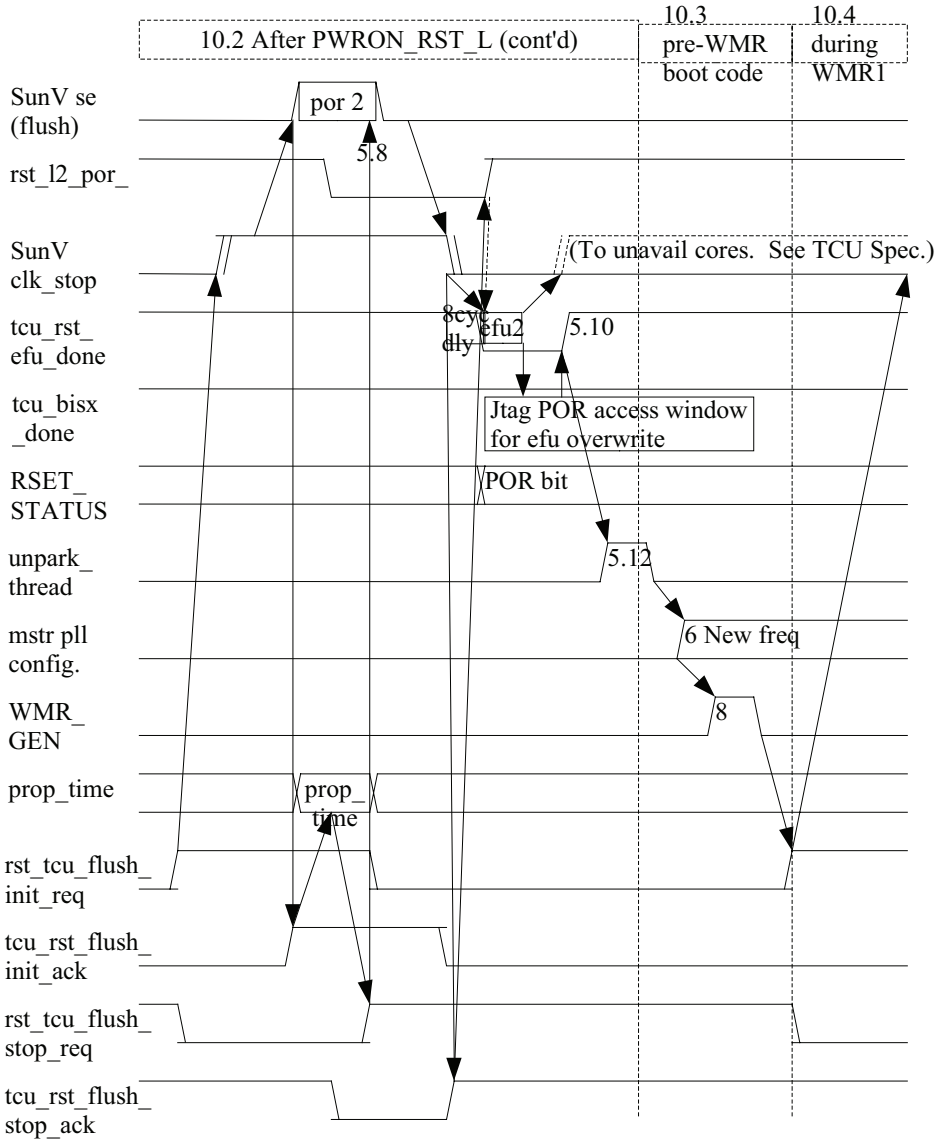
k. RST sets the POR bit of the RSET_STATUS register.

l. RST asserts rst_unpark_thread to NCU. NCU asserts core_running to Trap Unit of lowest-numbered available SPC (which will be the same as the lowest-numbered running SPC).

12.9.3 Pre-WMR Boot Code

m. The lowest-numbered available SPC begins fetching and executing instructions at RSTVaddr || 0x20.The MMUs are turned off, in bypass mode, with default mapping. At first, only PROM working. Software has to enable everything else.

FIGURE 12-9 Power-On Reset Sequence - POR2



n. Pre-WMR boot code starts by reading RSET_STAT register, which indicates POR (as opposed to WMR).

6. ASI instructions “set up master configuration registers such as PLL config., BIST program config, and I/O drive strength.”
“The new configuration will take effect when the CR [WMR] state is exited.”

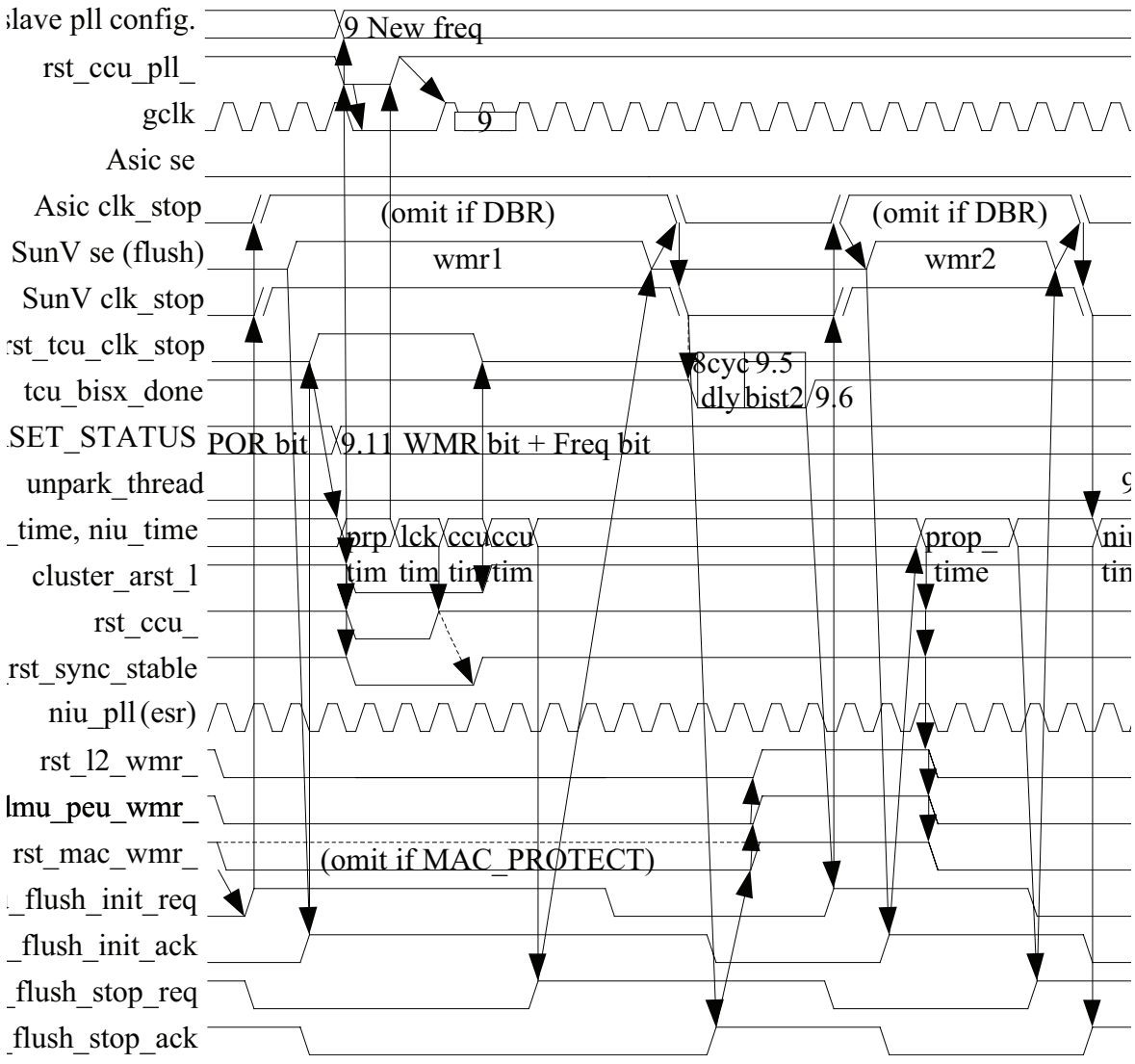
7. Pre-WMR boot code “clears error logs. (Alternately, this could be moved to later.)”

8. Pre-WMR boot code finishes by writing a 1 to the WMR_GEN bit of the Reset Generation Register, RESET_GEN. NCU deasserts core_running. RST deasserts rst_soc_run.

12.9.4 During WMR1

9. WMR 1: RST asserts rst_wmr_, dbg_init_, and PCI_EXPRESS_RESET_. The RST, in cooperation with the TCU, will flush-reset all WMR flip-flops in the chip. RST and TCU cause reset “by asserting both the scan_in_clk and scan_out_clk simultaneously while driving logic 0 onto the scan_in_data of every scan chain.” Figure 5 shows this as TCU asserting se, scan enable, during the interval labelled “wmr 1”.
   The NIU has at least two PLLs. Once locked, they will stay locked, even if the NIU is subsequently reset again.
   “The TCU contains a PLL config register accessible by ASI instructions. New PLL configurations will take effect at the next [warm reset] event. That event must persist sufficiently long for the PLL to stabilize at its new setting.... The assertion of [rst_wmr_] will cause the part to load configuration registers such as PLL config, BIST program config, and I/O drive strength.” Warm reset “will also flush... other flip-flops in the part.”

FIGURE 12-10 Power-On Reset Sequence - Warm Reset: WMR1+WMR2



a. Steps 9.2-9.10 below correspond to steps 5.2-5.10 in “During PWRON_RST_L”, above, starting with “PLLs lock.”

b. PLLs lock.

c. CCU asserts ccu_pll_locked. This is an analog signal, derived from l2clk, and independent of any reset signal and all other clock signals. Despite the existence of this signal, OpenSPARC T2 ignores it. Instead, we rely on the

external signals PWRON_RST_L and PB_RST_L, or counting down the lock_time register, so that the chip's behavior is repeatable.
The NIU needs 10 sfor its PLL to lock.

12.9.5 After WMR

d. Optionally, the service processor had asserted PB_RST_L. If it did assert it, then it now deasserts PB_RST_L synchronous with the reference clock. This forces an alignment of the rising edges of the cmpclk, ioclk, and ddrclk clocks.

   i. The Reset Unit waits for a number of cycles of the reference clock. The Lock Time register holds that number.

   ii. When both (1) the service processor has deasserted PB_RST_L and (2) Lock Time has passed, then RST deasserts rst_wmr_, dbg_init_, and PCI_EXPRESS_RESET_L.

e. BIST 2: "If at-speed [BISI or] BIST is desired," by pre-WMR boot code setting the TCU BISI and BIST registers, then "TCU launches [BISI or] BIST on caches."
"L2 Tag, Data, and VUAD arrays, when BISTed to zeros, are initialized to empty with good parity and good ECC."
"L1 I-cache, L1 D-cache, when BISTed to zeros, initialized to good parity"
l2dtakes 128k*8= 1,048,576cmp_clk cycles
l2ttakes 8k*8= 65,536cmp_clk cycles
VUADtakes 256*8= 2,048cmp_clk cycles
Total= 1,116,160cmp_clk cycles
"When BIST is complete, the part will store the BIST results and flush reset the part in preparation to begin code execution."

f. TCU asserts tcu_bisx_done.

g. WMR 2: BISI or BIST may have changed the state of some flip-flops connected to SRAM outputs. RST causes a second flush WMR to reset those flops. Figure 7 shows this sequence as "WMR 2". RST asserts rst_niu_, and rst_dmu_peu_wmr_, and TCU asserts se.

h. After lock_time clock cycles, TCU deasserts se.

i. After a further lock_time clock cycles, RST deasserts rst_niu_, and rst_dmu_peu_wmr_. This allows the signals that had caused flush reset, such as se, time to propagate.

j. (E-Fuse Unit, EFU, remains idle during WMR.)

k. RST sets the WMR bit of the RSET_STATUS register. Since the frequency changed, it also sets the FREQ bit.

l. RST asserts rst_unpark_thread to NCU. NCU asserts core_running to Trap Unit of lowest-numbered enabled SPC.

12.9.6 Post-WMR boot code

m. The lowest-numbered enabled SPC begins fetching and executing instructions at RSTVaddr || 0x20.
The MMUs are turned off, in bypass mode, with default mapping. At first, only PROM working. Software has to enable everything else.
"Come out of warm reset, again at reset vector."

n. Post-WMR boot code starts by reading RSET_STAT register, which indicates WMR, with clock change.

10. [TCU already launched BIST on caches, in step 9.5.]

11. Initialize the L1 tags of the lowest-numbered available processor core. [Not necessary since TCU did BISI or BIST.]
"L1 I-tags, L1 D-tags need to be explicitly ASI written to invalid, with good parity".

12. [This step moved in sequence to be before WMR. Was:
"9. Wait for BIST_DONE indications."]

13. Enable error detection on L1 and L2 caches.

14. Enable L1 and L2 caches.

15. Post-WMR boot code continues as outlined in Programmer's Reference Manual.

12.10 Warm Reset Sequence

Since the entire Power-On Reset sequence includes a warm reset, a warm reset that is not caused by POR is similar. In fact, if the clock divider register is changed, it is identical to the WMR step of the POR sequence.

12.10.1 Before rst_mwr_

Three agents can cause a Warm Reset, as follows:

1. The user presses the Warm Reset pushbutton, or the external system processor asserts the PB_RST_L input pin.

2. Software writes a 1 to the WMR_GEN bit of the RESET_GEN register, as in Step 8 of the “Reset sequence for POR”, above.

3. NCU or the L2 cache detects a Fatal Error and asserts ncu_rst_fatal, l2t0_rst_fatal,..., or l2t7_rst_fatal.

NCU deasserts core_running. RST deasserts rst_soc_run.

12.10.2 During rst_wmr_

RST asserts rst_wmr_, dbg_init_, and PCI_EXPRESS_RESET_L.
The RST, in cooperation with the TCU, will flush-reset all WMR flip-flops in the chip. RST and TCU cause reset “by asserting both the scan_in_clk and scan_out_clk simultaneously while driving logic 0 onto the scan_in_data of every scan chain.” Figure 5 shows this as TCU asserting se, scan enable, during the interval labelled “wmr 1”.
The NIU has at least two PLLs. Once locked, they will stay locked, even if the NIU is subsequently reset again.
“The TCU contains a PLL config register accessible by ASI instructions. New PLL configurations will take effect at the next [warm reset] event. That event must persist sufficiently long for the PLL to stabilize at its new setting.... The assertion of [rst_wmr_] will cause the part to load configuration registers such as PLL config, BIST program config, and I/O drive strength.” Warm reset “will also flush all other flip-flops in the part.”

PLLs lock.

CCU asserts pll_locked. This is an analog signal, derived from l2clk, and independent of any reset signal and all other clock signals. Despite the existence of this signal, OpenSPARC T2 ignores it. Instead, we rely on the external signal PB_RST_L, or counting down the lock_time register, so that the chip's behavior is repeatable.

Optionally, the service processor had asserted PB_RST_L. If it did, then it now deasserts PB_RST_L synchronous with the reference clock. This forces an alignment of the rising edges of the cmpclk, ioclk, and ddrclk clocks.

The Reset Unit waits for a number of cycles of the reference clock. The Lock Time register holds that number.

12.10.3 After rst_wmr_

When both (1) the service processor has deasserted PB_RST_L and (2) Lock Time has passed, then RST deasserts rst_wmr_, dbg_init_, and PCI_EXPRESS_RESET_L.

"If at-speed [BISI or] BIST is desired," by pre-WMR code setting the TCU BISI and BIST registers, then "TCU launches [BISI or] BIST on caches."
"L2 Tag, Data, and VUAD arrays, when BISTed to zeros, are initialized to empty with good parity and good ECC."
"L1 I-cache, L1 D-cache, when BISTed to zeros, initialized to good parity"
"When BIST is complete, the part will store the BIST results and flush reset the part in preparation to begin code execution."

TCU asserts tcu_bisx_done.

BISI or BIST may have changed the state of some flip-flops connected to SRAM outputs. RST causes a second WMR to reset those flops. Figure 7 shows this sequence as "WMR 2". RST asserts rst_wmr_, and TCU asserts se.

After lock_time clock cycles, TCU deasserts se.

After a further lock_time clock cycles, RST deasserts rst_wmr_. This allows the signals that had caused flush reset, such as se, time to propagate.

(E-Fuse Unit, EFU, remains idle during WMR.)

RST sets the WMR bit of the RSET_STATUS register. In addition, if the frequency changed, it also sets the FREQ bit.

RST asserts rst_unpark_thread to NCU. NCU asserts core_running to Trap Unit of lowest-numbered enabled SPC.

12.10.4 Post-WMR boot code

The lowest-numbered enabled SPC begins fetching and executing instructions at RSTVaddr || 0x20.)
The MMUs are turned off, in bypass mode, with default mapping. At first, only PROM working. Software has to enable everything else.

Post-WMR boot code starts by reading RSET_STAT register, which indicates WMR.)
(If RSET_STAT register indicates WMR, *with* clock change, go to the step, circa Step 9.10, with that condition in "POR Reset Sequence, after WMR", above.)

Check local error logs.

Post-WMR boot code continues as outlined in Programmer's Reference Manual.

12.11 Reset Sequence for DBG

DBG is the same as WMR, except that the Reset Unit does not reset DMU, PEU, nor NIU.

12.12 Reset Sequence for NIU

1. Software makes sure that all outstanding transactions are complete.
2. Software writes to the NIU bit of the SSYS_RESET register.
3. RST asserts rst_niu_.
4. The NIU needs 10 sfor its PLL to lock. RST waits the number of system clock (pll_sys_clkp, soon to be ccu_rst_sys_clk+) cycles specified in the LOCK_TIME register.
5. RST deasserts rst_niu_. RST clears the NIU bit of the SSYS_RESET register.

12.13 Reset Sequence for XIR

1. Software writes a 1 to the XIR_GEN bit of the RESET_GEN register, or the user presses the BUTTON_XIR_ pushbutton.
2. RST does not need to debounce BUTTON_XIR_ input pin.
3. RST asserts rst_ncu_xir_.
4. NCU asserts ncu_rst_xir_done.
5. RST deasserts rst_ncu_xir_.
6. RST clears the XIR_GEN bit of the RESET_GEN register.

12.14 Reset and Scan of the Reset Unit

Three clocks drive the four blocks in the Reset Unit:

1	cmp clock	rst_cmp_ctl
2	sys clock	rst_fsm_ctl
3	io clock	rst_io_ctl
4	io clock	rst_ucbflow_ctl

The sys clock drives rst_fsm_ctl directly, with no cluster header. The Reset Unit gates the.tcu_clk_stop input port of the cmp cluster header, but not that of the io cluster header, as described below.

12.14.1 tcu_rst_clk_stop

```
module rst...

Guided by suggestions from the TCU designers, the Reset
Unit


gates tcu_rst_clk_stop with tcu_rst_scan_mode, as follows:

  clkgen_rst_cmp       clkgen_rst_cmp          (
    .tcu_clk_stop    (tcu_clk_stop_scan_mode),// =
tcu_rst_scan_mode ?
                                        //   rst_clk_stop : 1'b0;
...
  rst_fsm_ctl              rst_fsm_ctl           (
   .rst_clk_stop        (tcu_rst_clk_stop    ),// Assign stmt.

module rst_fsm_ctl ...
  assign   tcu_clk_stop = tcu_rst_scan_mode ? rst_clk_stop :
1'b0;
  assign   tcu_clk_stop_scan_mode = tcu_clk_stop;
```

Thus, in scan mode, the Reset Unit passes tcu_rst_clk_stop to the.tcu_clk_stop input port of clkgen_rst_cmp, so the TCU can scan the Reset Unit. Otherwise, it passes 1'b0, so the cmp clock runs whenever the PLL is running.

Since the rst_cmp_ctl block consist only of sync_en flops, it is possible to reset that block by simply allowing values from upstream flops to shift through it in the first few cycles after they are reset.

12.14.2 tcu_rst_io_clk_stop

The Reset Unit does not gate the.tcu_clk_stop (tcu_rst_io_clk_stop) input port of the other cluster header, clkgen_rst_io. The TCU is free to stop the io clock as it sees fit. The two Reset Unit blocks that operate on the io clock, the sync_en block rst_io_ctl and the UCB block rst_ucbflow_ctl, are reset by synchronous reset. The Reset Unit asserts ucb_clr_io_ for a longer period of time than just the flush reset time, allowing the io clock to run again and reset those two blocks. Since the rst_io_ctl block consist only of sync_en flops, it is possible to reset that block by simply allowing values from upstream flops to shift through it in the first few cycles after they are reset.

12.15 Reset Unit Ports

12.15.1 Input Ports

We consider these inputs to be asynchronous to the system clock:

1. PWRON_RST_L(mio_rst_pwron_rst_l)
2. PB_RST_L(mio_rst_pb_rst_l)
3. BUTTON_XIR_L(mio_rst_button_xir_l)

For each of these signals, to ensure that OpenSPARC T2 reliably captures it, the FPGA must assert it for a minimum of either:

1. the system clock period plus the set-up time of cl_sc1_clksyncff_4x, the synchronizer cell, or
2. the system clock period plus the hold time of cl_sc1_clksyncff_4x,

whichever is longer.

In addition, the Reset Unit requires that any input signal that crosses to the sys_clk domain must be held steady for at least two sys_clk cycles. This is because the CCU asserts the sync_en signals only once every ref_clk cycle, and ref_clk has a period of

two sys_clk cycles. This applies to all inputs except the sync_en pulses themselves, and the UCB signals ncu_rst_vld, ncu_rst_data[3:0], and ncu_rst_stall. The NCU launches them on io_clk, and the Reset Unit captures them on the same clock.

TABLE 12-30 Inport Ports Clocks

Source	Clock	Input port	
FPGA	sys	ccu_rst_sys_clk	
ccu	gclk	gclk	
ccu	io	ccu_io_out	
tcu	-	tcu_div_bypass	
tcu	cmp	scan_in	tcu_soc6_scan_out
		tcu_rst_clk_stop	Not used.
		tcu_rst_io_clk_stop	Not used.
		tcu_pce_ov	
		tcu_aclk	
		tcu_bclk	
		tcu_scan_en	
		tcu_rst_scan_mode	
		tcu_atpg_mode	(Reset Unit ignores.)
ccu	cmp	ccu_io_cmp_sync_en	
ccu	cmp	ccu_cmp_io_sync_en	
ccu	cmp	ccu_sys_cmp_sync_en	Synchronization pulse for each signal that crosses between synchronous clock domains.
ccu	cmp	ccu_cmp_sys_sync_en	
FPGA	async	mio_rst_pwron_rst_l	Assert for at least 2 sys_clk cycles.
FPGA	async	mio_rst_button_xir_l	Assert for at least 2 sys_clk cycles.
FPGA	async	mio_rst_pb_rst_l	Assert for at least 2 sys_clk cycles.
tcu	cmp	tcu_rst_flush_init_ack	
tcu	cmp	tcu_rst_flush_stop_ack	
tcu	cmp	tcu_rst_asicflush_stop_ack	

TABLE 12-30 Inport Ports Clocks *(Continued)*

tcu	io	tcu_test_protect	During mbist, lbist, jtag scan, trans test may want to block tcu, rst and ccu from seeing random activity from ucb (NCU), SPC's, etc. This signal synched to ioclk, and set via jtag id for blocking.
ccu	io	ccu_rst_change	Only assert rst_ccu_ and rst_ccu_pll_, and wait LOCK_TIME, when ccu holds ccu_freq_change high.
ccu	cmp	ccu_rst_sync_stable	Not used.
tcu	cmp	tcu_bisx_done	
tcu	cmp	tcu_rst_efu_done	
l2t	io	l2t0_rst_fatal_error	Asserted for one clock cycle.
		l2t1_rst_fatal_error	
		l2t2_rst_fatal_error	
		l2t3_rst_fatal_error	
		l2t4_rst_fatal_error	
		l2t5_rst_fatal_error	
		l2t6_rst_fatal_error	
		l2t7_rst_fatal_error	
ncu	io	ncu_rst_fatal_error	Asserted for one clock cycle.
		ncu_rst_xir_done	
		ncu_rst_vld	
		ncu_rst_data[3:0]	
		ncu_rst_stall	

12.15.2 Output Ports

TABLE 12-31 lists the output ports of the Reset Unit.

TABLE 12-31 Output Ports Clocks

Sink	Clock	Ultimate clock	Output port	
tcu	-	-	scan_out	rst_scan_out. Untimed.
efu, l2b, l2t	cmp	cmp	rst_l2_por_	Vestige of OpenSPARC T I heritage of L2 cache.
	cmp	cmp	rst_l2_wmr_	Vestige of OpenSPARC T I heritage of L2 cache.
fc	sys	asyn	rst_wmr_protect	
mcu	io(was cmp)	dr	rst_mcu_selfrsh	Equal to MCU_SELFRSH bit of SSL_RESET register.
tcu	cmp	cmp	rst_tcu_flush_init_req	
	cmp	cmp	rst_tcu_flush_stop_req	
	cmp	cmp	rst_tcu_asicflush_stop_req	
	cmp	cmp	rst_tcu_dbr_gen	
	cmp	cmp	rst_tcu_clk_stop	See Figure 8.
	cmp	cmp	rst_tcu_pwron_rst_l	
niu	cmp	cmp	rst_niu_mac_	Goes to mac. The Reset Unit will reset mac on POR, and also on WMR1 if ccu_rst_change == 1, unless MAC_PROTECT is set.
niu	cmp	cmp	rst_niu_wmr_	Goes to the other niu clusters, rtx, tds, and rdp. The Reset Unit will reset them on both POR and WMR
dmu, peu	cmp	io, pc	rst_dmu_peu_por_	Assert for 15 s.
	cmp	io, pc	rst_dmu_peu_wmr_	Assert for 15 s.
	async, sys	async, cmp	rst_dmu_async_por_	
ncu	io	io	rst_ncu_unpark_thread	
	io	io	rst_ncu_xir_	

TABLE 12-31 Output Ports Clocks *(Continued)*

mio	sys	asyn	rst_mio_pex_reset_l	(Follows rst_dmu_peu_wmr_.) Assert for 15 us. .
	sys	fpga (asyn)	rst_mio_ssi_sync_l	Assert for 50-100 s. Assert duringWMR. (Was rst_mio_fatal_error.)
	sys	sys	rst_mio_rst_state[5:0]	Reset Unit state machine state
ncu	io	io	rst_ncu_stall	UCB
	io	io	rst_ncu_vld	
	io	io	rst_ncu_data[3:0]	
ccu	sys	cmp	rst_ccu_	
	sys	cmp	rst_ccu_pll_	
	sys	asyn	cluster_arst_l	

17 total clock_stop domains (increased to 24):

1. 8 sparc cores
2. 4 mmu + l2 pair
3. 1 ddr
4. 1 IO (ncu)
5. 1 pci express
6. 1 niu
7. 1 ccx, tags

12.16 Appendices

12.16.1 Appendix I: Glossary

ASI	Address Space Identifier
ASR	Ancillary State Register
MISR	Multiple-Input Signature Register. Used in LBIST
RClk	Regional Clock

12.16.2 Appendix II: Glossary of shadow terms

Master configuration register

Holds the value that will be placed in the slave version of the register, sometimes called the shadow register, when the next WMR occurs. The slave register then supplies the value to operational logic. Examples are PLL clock divider and I-O drive strength.

Shadow scan configuration register

Used to control the scanning of the shadow scan registers.

Shadow scan register

"A number of internal states can be captured and scanned out without stopping the clocks using the shadow scan instruction. The state of the target nodes is captured when the JTag state machine enters the Capture-DR state and the scan occurs in the Scan-DR state. A great many internal states are available for observation but a limited number of shadow scan flops are dedicated to the task of capturing and scanning those states. The shadow scan configuration register controls which internal nodes will be captured into the shadow scan chain. The shadow scan config. register can be accessed with a JTag DR scan operation."

Shadow status bits of RSET_STATUS register

“HW will copy the current reset status into a shadow status whenever a reset occurs.”

12.16.3 Appendix III: Promotion among Core Available, Enable, and Status registers

Before the first instruction executes, TABLE 12-32 shows registers that must contain their correct values:

TABLE 12-32 Register Abbreviations

Register ASI name	Abbreviation in this appendix
ASI_CORE_AVAILABLE	CA
ASI_CORE_ENABLE	CE
ASI_CORE_ENABLE_STATUS	CES

TABLE 12-33 shows the sequence of events during Power-On Reset. The NCU accepts the initial, and only, value from the E-Fuse Unit into CA during EFU1 and then again during EFU2. The NCU then transfers that value into CE and CES during times labeled NCU1 and NCU2. The NCU controls NCU1, NCU2, and NCU pre-compute.

TABLE 12-33 Power-On Reset sequence of Events

	POR1	EFU1	NCU1	MBisi1	POR2	EFU2	NCU2	Soft ware	NCU pre-compute
CA	0	EFU1 CA	CA	CA	0	EFU2 CA	CA	CA	CA
CE	0	0	CA CE1	CE1	0	0	CA CE1	CE2	CE2
CES	0	0	CA CES1	CES1	0	0	CA CES1	CES1	CES1
CESpre	-	-	-	-	-	-	-	-	f(CE2, CES1)

TABLE 12-34 shows the sequence of events during Warm Reset. Warm Reset skips the EFU1 and EFU2 steps shown in the previous table, so the corresponding columns in this table are blank.

TABLE 12-34 Warm reset Sequence of Events

	WMR1		NCU1	MBist2	WMR2		NCU2		
CA	CA								
CE	CE2								
CES	CESpre -> CES2		CE2 -> CES2	CES2	CES2		CE2 -> CES2		