OpenSPARC™ T2 System-On-Chip (SoC) Microarchitecture Specification (Part 2 of 2)

Sun Microsystems, Inc.
www.sun.com

Part No. 820-5090-10
May 2008, 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 is divided into two volumes, (Part 1 of 2) P/N 820-2620-10 and (Part 2 of 2) P/N 820-5090-10.

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

This *OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification (Part 2 of 2).*, P/N 820-5090-10 includes detailed functional descriptions of the following OpenSPARC T2 System-on-Chip I/O components.

Chapter 1 describes the Data Management Unit (DMU)

Chapter 2 describes the Miscellaneous I/O (MIO)

Chapter 3 describes the Debug Functions

Chapter 4 describes the Electronic Fuse Unit (EFU)

Chapter 5 describes the Reset Functions

Chapter 6 describes the Network Interface Unit

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 (Part 1 of 2)*	820-2620	PDF	Online
Documentation	*OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification (Part 2 of 2)*	820-5090	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 1 of 2), part number 820-2620-10.

OpenSPARC T2 System-On-Chip (SoC) Microarchitecture Specification (Part 2 of 2), part number 820-5090-10.

CHAPTER 1

Data Management Unit (DMU)

This chapter contains the following sections:

1.1 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, OpenSPARC T2 SIU (system interface unit) and OpenSPARC T2 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 documents the high level DMU function.

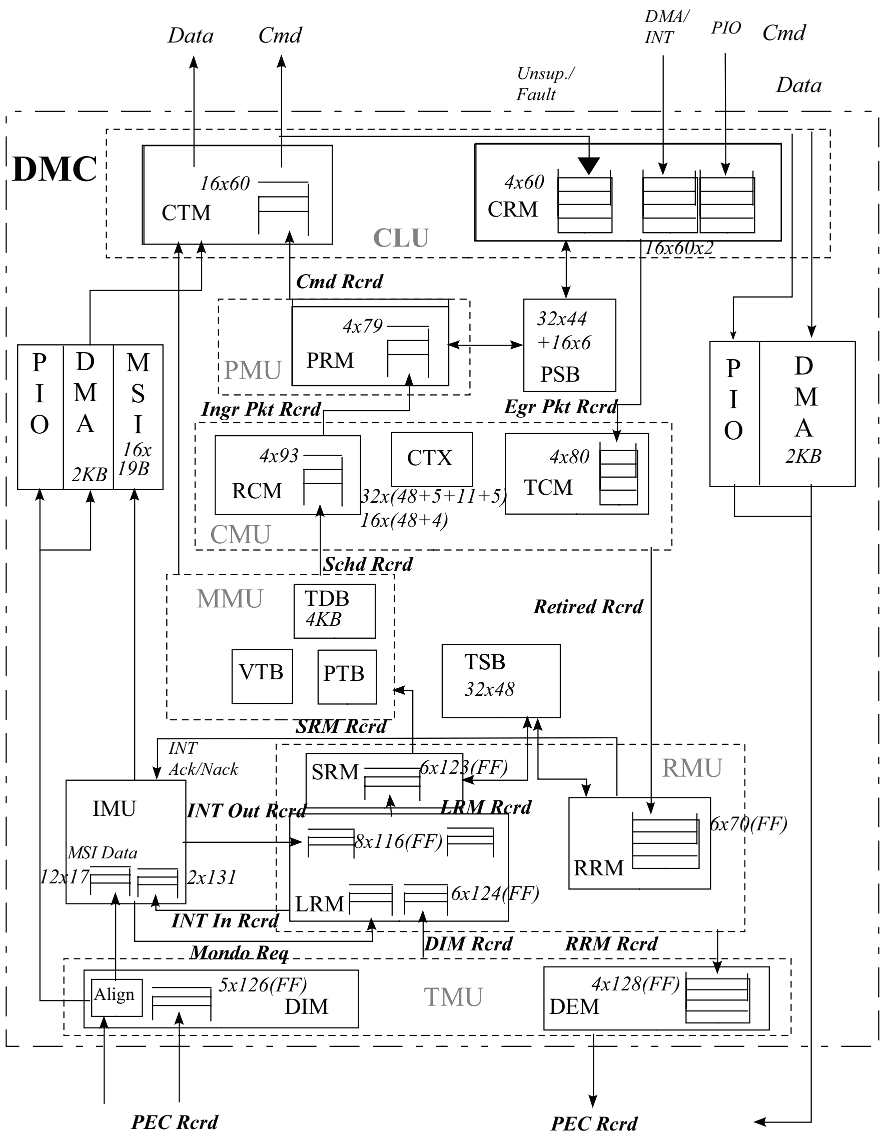
The DMU is responsible for managing and directing all command and data flows from/to PCI-Express Unit (PEU), System Interface Unit (SIU), and Non-Cacheable Unit (NCU). The DMU has three primary external interfaces, one to the SIU, one to the NCU and one to the PEU.

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, and reassembles 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).

1.1.1 DMU Block Diagram

FIGURE 1-1 DMU Block Diagram



1.1.2 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

1.1.3 General Ingress Pipeline (IP) 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.

1.1.4 General Egress Pipeline (EP) Operations

DMA Read 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 Manager) 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 DMU 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 Manager 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 DMU PIO Pool to VC Data Buffer

1.2 Functional Description of DMU Sub-blocks

The DMU 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) and Interface Layer Unit (ILU). The following sections describe the architecture, functionality and change requirement of each groups.

1.3 Transaction Manager Unit (TMU)

1.3.1 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).

1.3.1.1 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 manager. 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 DMU. 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.

1.3.1.2 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.

1.3.1.3 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.

1.4 Interrupt Message Unit (IMU)

1.4.1 IMU Functional Description

1.4.1.1 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 DMU 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 OpenSPARC T2 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).

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 MSIs 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 MSIs 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 increments 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.

1.4.1.2 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 FIGURE 1-2.

- 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.

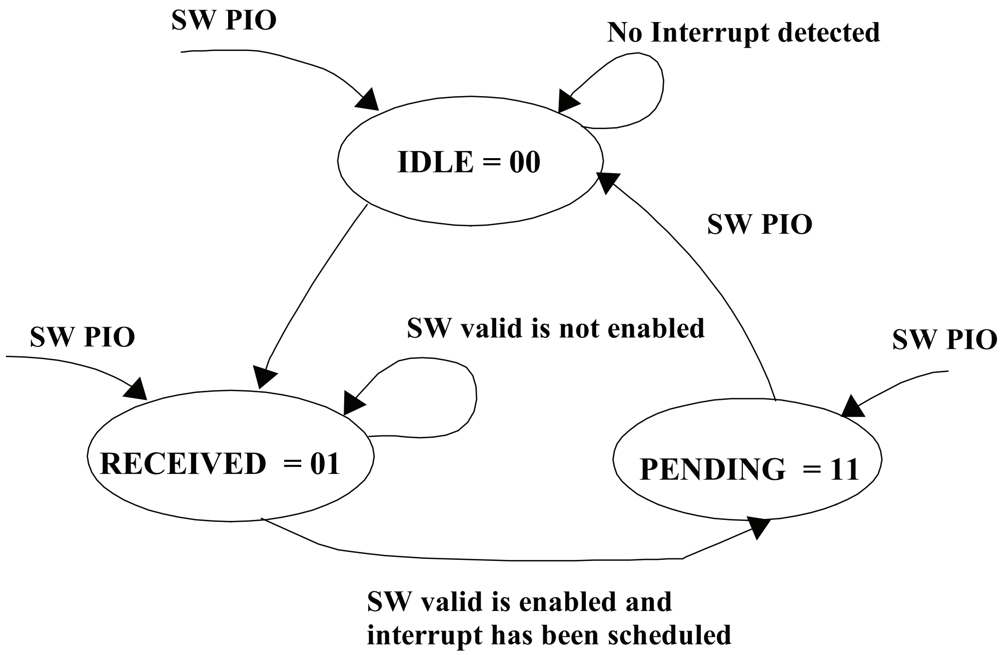
    b. RECEIVED - indicates that an interrupt has been detected by the hardware and should be delivered to the processor if/when the valid bit is set in its mapping register.

    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 1-2 IMU Mondo State Machine



1.4.1.3 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).

Capability Structure for 32-bit Message Address

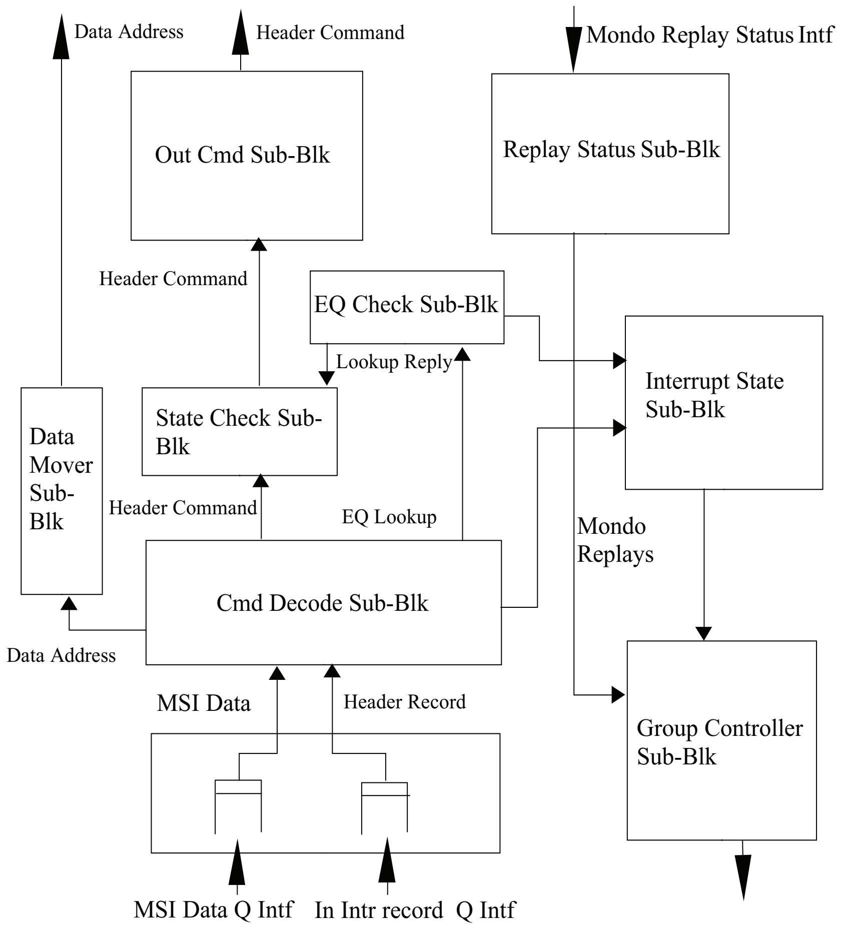
31 ---- 16	15 ----- 8	7 ----- 0
Message Control	Next Ptr	Cap ID
	Message Address	
		Message Data

.

Capability Structure for 64-bit Message Address

31 ---- 16	15 ----- 8	7 ----- 0
Message Control	Next Ptr	Cap ID
	Message Address	
	Message Upper Address	
		Message Data

FIGURE 1-3 IMU Block Diagram



1.4.1.4 IMU Mondo INO Mapping Table

TABLE 1-1 IMU Mondo INO Mapping

INO	Function
INOs 0-19	Reserved
INO 20-23	4 interrupts for PCI Express INTx Emulation -20 INTA -21 INTB -22 INTC -23 INTD
INO 24 – 59	36 Event Queue Interrupts
INO 60 – 61	Reserved
INO 62	DMU Internal Interrupt
INO 63	PEU Internal Interrupt

1.4.1.5 IMU CSRs Change List

Interrupt Mapping Registers (0x601000 – 0x601150) 42 consecutive registers, one for each Mondo)

TABLE 1-2 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 two mondo formats the mondo will use. A value of 1 = Data Bearing Mondo. A value of 0 = Non Data Bearing 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.

TABLE 1-2 Interrupt Mapping Registers *(Continued)*

Field	Bits	Reset Name	Reset Value	Type	Description
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 1-3 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 may 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 1-4 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 (OpenSPARC T2 IO Clock Frequency)

TABLE 1-5 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 two 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 1-6 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 two 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 1-7 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 1-8 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 1-9 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 1-10 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 1-11 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 1-12 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 1-13 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 1-14 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 1-15 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 1-16 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 1-17 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 1-18 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 1-19 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 1-20 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 1-21 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 1-22 ERR COR Mapping Register (0x00630000)

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 1-23 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 1-24 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 1-25 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 1-26 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 1-27 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 1-27 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 1-28 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 1-28 IMU Interrupt Enable Register (0x00631008) *(Continued)*

Field	Bits	Reset Name	Reset Value	Type	Description
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 1-29 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
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

TABLE 1-29 IMU Interrupt Status Register – (0x00631010) *(Continued)*

Field	Bits	Reset Name	Reset Value	Type	Description
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
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 1-30 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

TABLE 1-30 IMU Error Status Clear Register (0x00631018) *(Continued)*

Field	Bits	Reset Name	Reset Value	Type	Description
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
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

TABLE 1-30 IMU Error Status Clear Register (0x00631018) *(Continued)*

Field	Bits	Reset Name	Reset Value	Type	Description
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 1-31 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

TABLE 1-31 IMU Error Status Set Register (0x00631020) *(Continued)*

Field	Bits	Reset Name	Reset Value	Type	Description
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
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 1-32 IMU RDS Error Log Register (0x00631028)

Field	Bits	Reset Name	Reset Value	Type	Description TYPE
TYPE	63:58				The lowest six 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
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 1-33 IMU SCS Error Log Register (0x00631030)

Field	Bits	Reset Name	Reset Value	Type	Description TYPE
TYPE	63:58				The lowest six 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

TABLE 1-33 IMU SCS Error Log Register (0x00631030)

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 1-34 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

1.5 Record Management Unit (RMU)

1.5.1 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.

1.5.1.1 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.

1.5.1.2 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.

1.5.1.3 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.

1.6 Transaction Scoreboard Unit (TSB)

1.6.1 TSB Function Description

The TSB is responsible for tagging and tracking all DMA Read requests and unsupported transactions through the DMU 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.

1.7 Memory Management Unit (MMU)

1.7.1 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 eight 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 eight 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).

1.7.1.1 IOMMU Bounds Check for Bypass Mode

Fire DMU follows the JBUS spec and uses {PA[42:41]==00, PA[40:36]==agent_id} for cacheable space. In 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.

1.7.1.2 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 has custom designed the Virtual Tag Buffer. The following 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]).

1.7.1.3 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 has custom designed the Physical Tag Buffer. 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.

1.8 Context Manager Unit (CMU)

1.8.1 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.

1.8.1.1 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.

1.8.1.2 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.

1.8.1.3 Context Record (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.

1.9 Packet Manager Unit (PMU)

1.9.1 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).

1.9.1.1 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.

1.10 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 PIOs and another for DMAs.

1.10.1 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 OpenSPARC T2. However this is only 64 id's, one more bit is to be added to account for JTAG access by the NCU.

1.11 Cache Line Unit (CLU)

1.11.1 CLU Function Description

The CLU manages the DMU-DSN interface. It consists of two sub-blocks, Cacheline Transmit Manager (CTM) and Cacheline Receive Manager (CRM).

1.11.1.1 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.

1.11.1.2 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.

1.11.1.3 Mondo Interrupt -> One Data Beat

The CTM block currently only extracts 1 data beat from the DIU ram, and then constructs the last three 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.

1.12 Data In Unit (DIU)

1.12.1 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.

1.13 Data Out Unit (DOU)

1.13.1 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.

1.13.2 SRAM

1.13.2.1 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.

The current SRAMs are TDB, DIU, and two DOU rams. The existing synthesized cam in the MMU will be implemented as a custom cam/ram block.

There will be two 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.

1.14 DMU SIU/NCU Interface Unit (DSN)

1.14.1 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.

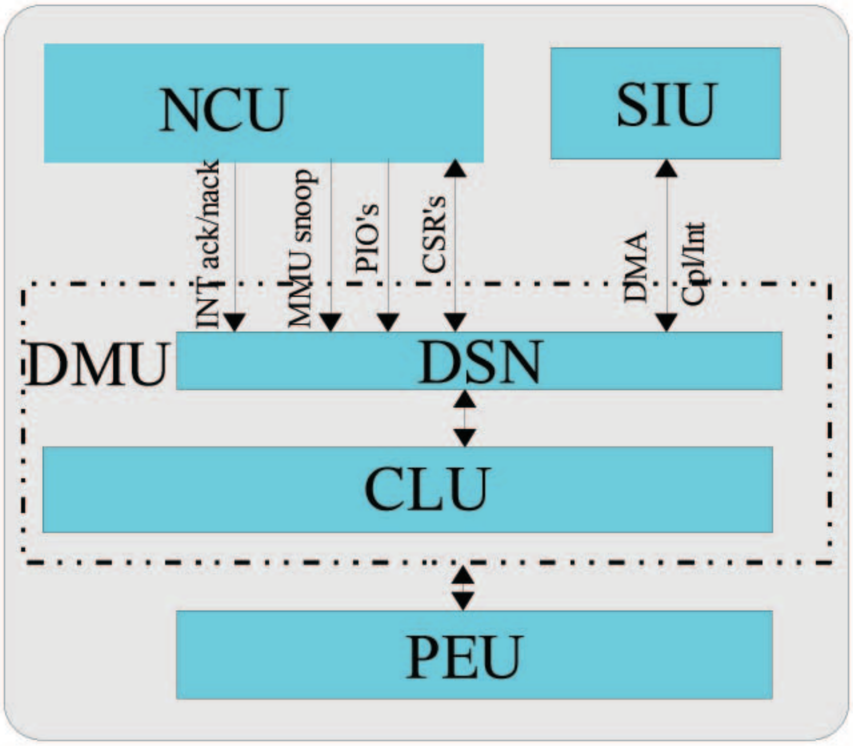
Also regarding the address buses:

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 PIOs the NCU must manage PA[39:36] such that they always map to the PA[35:0] expected by the DMU.

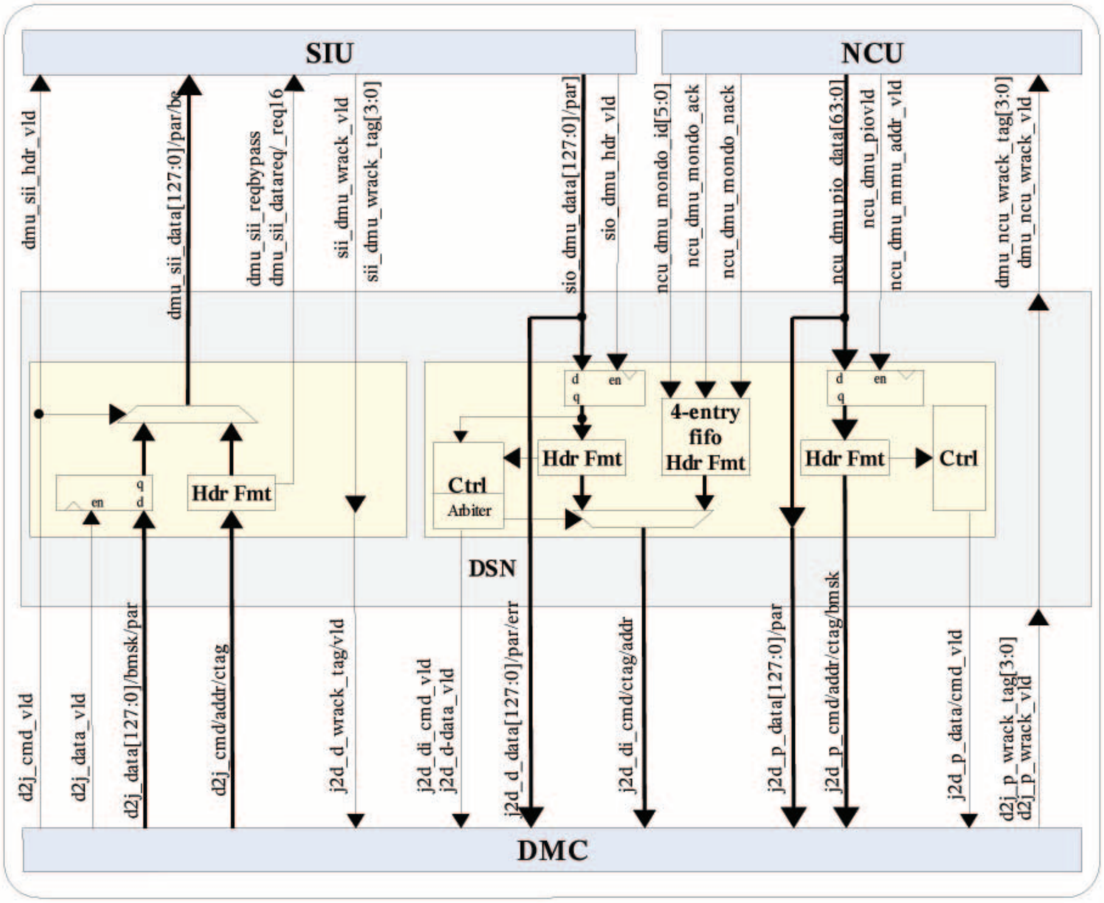
1.14.2 DSN Block Diagrams

FIGURE 1-4 Interface Block Diagram



1.14.3 DSN Detailed Block Diagram

FIGURE 1-5 Detailed Block Diagram



1.14.4 DSN Interface Descriptions

1.14.4.1 DSN-SIU Interface

The DSN-SIU interface will be used for all DMAs, 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. DMAs 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 (MSIs 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 PIOs 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 eight byte PIO rd cpls 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 two bit scoreboard of pio_addr[3] indexed by the credit_id, written on NCU vld, and reading on pio_rd_cpls. 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 eight byte, if 16 byte then the data is not replicated, with a second bit in the scoreboard.

1.14.4.2 DSN-SIU Interface List

TABLE 1-35 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 cpls
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 two cases, 16 byte and <=8byte cpls, in the case of 8byte PIO cpls 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 1-35 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).

1.14.4.3 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 1-36 SIU to DSN Egress Commands

Transaction type	Cmds	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 OpenSPARC T2
Interrupt Ack	2’b11	N/A for OpenSPARC T2

TABLE 1-37 DMC_TAG Field Definitions

Field	Bits	Description
DMA transactions		
dmc[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

TABLE 1-37 DMC_TAG Field Definitions *(Continued)*

Field	Bits	Description
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	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 cpls by using dmc_tag[15]. And the NCU will use the thread_id from the mondo data to determine which thread to interrupt.

1.14.4.4 SIU to DSN Outbound Header sent by SIU (DMA rd cpls only)

TABLE 1-38 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.

1.14.4.5 Bit Mapping from DSN to SII for DMA rd/wr 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]

dmu_sii_data[39:6]= d2j_addr[33:0] -> PA[39:6]

dmu_sii_data[5:0]= 0

1.14.4.6 Bit Mapping from NCU/SIU Header to DMU 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 DMU 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.

1.14.4.7 DMU 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.

The DSN logic will build the header to the SIU using the following commands from the DMU.

Note – 64 byte PIOs are not supported in OpenSPARCT2, and on PIO read completions the address will not be returned.

TABLE 1-39 DMU to SIU Ingress Command Bit Definitions

Transaction type	**Cmds**	**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/FIFOs etc. making in jbc_tag[11:0] would have added a bit throughout the entire crm/pmu/psb/scoreboard/ctm path.

1.14.4.8 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. TABLE 1-40 gives the header values:

TABLE 1-40 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 cpls, 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]	UnmappedAddressError	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 1-40 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 DMAs 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]	0s Always cache line aligned

1.14.4.9 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 three fields, ctag, address and command/status. Each group will have its own protection, ecc on the ctag and parity on the other two.

The ctag ecc will use a SECDCD code with six 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]};
```

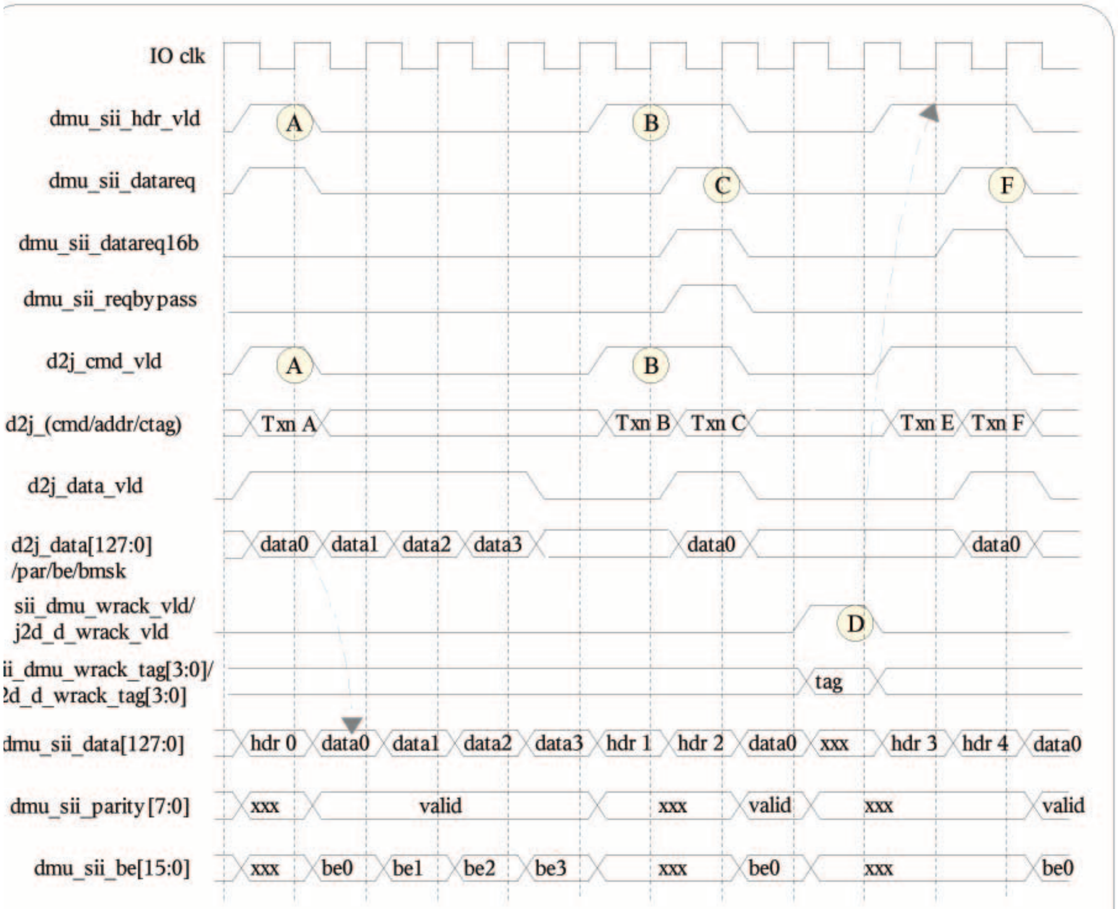
where di[15:0] = dmc_tag[15:0];

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]}

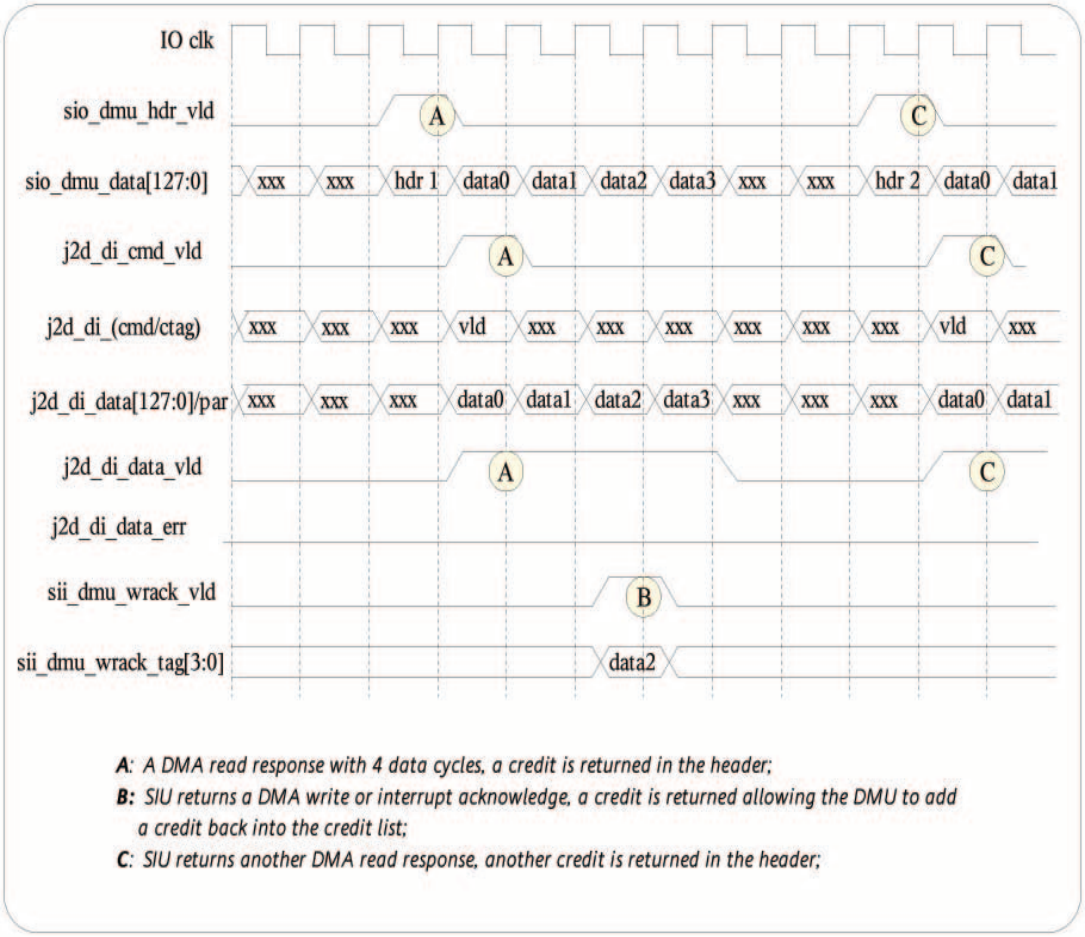
1.14.4.10 DSN-SII Interface Timing Diagrams

FIGURE 1-6 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 1-7 Egress Interface Timing Diagram


IO clk
sio_dmu_hdr_vld
sio_dmu_data[127:0]
j2d_di_cmd_vld
j2d_di_(cmd/ctag)
j2d_di_data[127:0]/par
j2d_di_data_vld
j2d_di_data_err
sii_dmu_wrack_vld
sii_dmu_wrack_tag[3:0]
xxx
hdr 1
data0
data1
data2
data3
hdr 2
vld
A
B
C
A: A DMA read response with 4 data cycles, a credit is returned in the header;
B: SIU returns a DMA write or interrupt acknowledge, a credit is returned allowing the DMU to add a credit back into the credit list;
C: SIU returns another DMA read response, another credit is returned in the header;

Only DMA rd cpls and wracks.

1.14.4.11 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 eight 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 eight bytes sent by the NCU on the 16 byte SIU bus and set the bytemask correctly. The DMU does not allow eight byte requests to cross eight 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.

1.14.4.12 DSN-NCU Interface Pin List

TABLE 1-41 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 acks		
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 DMU internal, tbd
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 1-41 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 DMU/MMU rams (deviostb & tdb
Note: the error injection Signals are levels, thus will force errors on all transactions until undriven		

1.14.4.13 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 1-42 NCU to DSN PIO Command Bit Definitions

Transaction type	Cmds	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 1-43 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 eight bits of thread id are not sent by the NCU to the DSN for PIOs, only Cores can send PIOs, bit thread_id[6]== 1 implies JTAG access.

1.14.4.14 Bit Mapping from NCU Header to DMU for PIO rd/wr

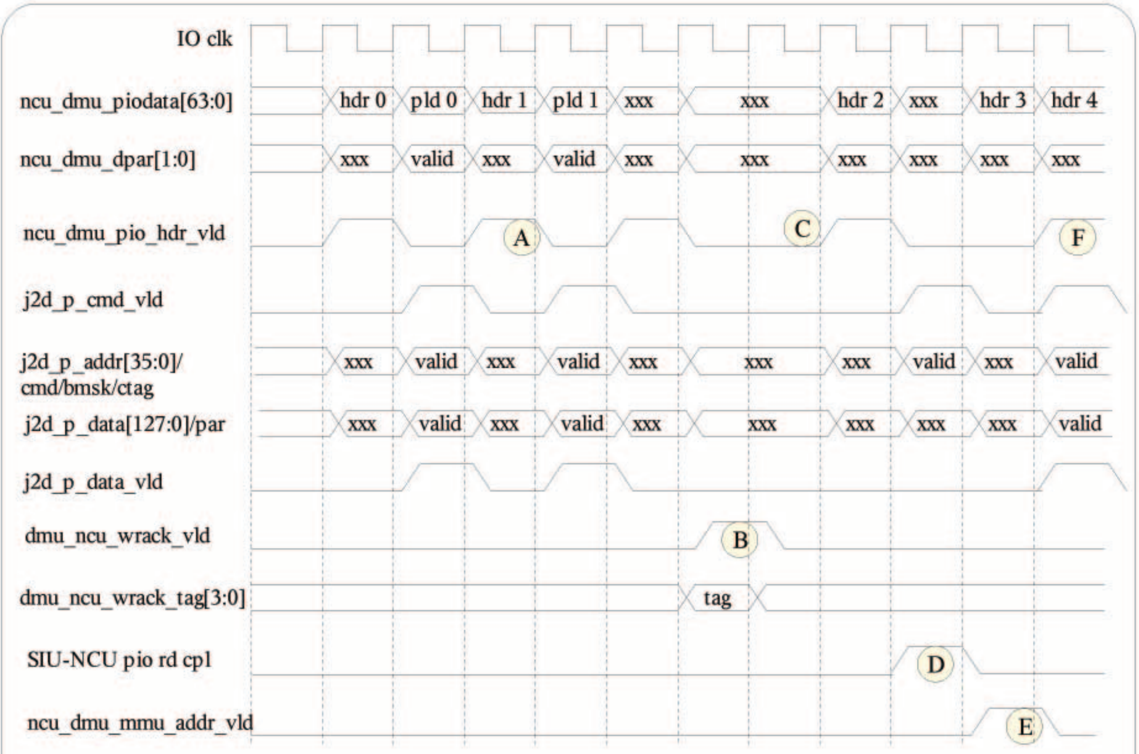
```
  j2d_p_addr[35:0]=  ncu_dmu_pio_data[35:0]
  j2d_p_cmd[3]= ncu_dmu_pio_data[60]
  j2d_p_cmd[2]= 1'b1
  j2d_p_cmd[1]=  !ncu_dmu_pio_data[37] && ncu_dmu_pio_data[36] ||
                !ncu_dmu_pio_data[37] && !ncu_dmu_pio_data[36]
  j2d_p_cmd[0]=  ncu_dmu_pio_data[37] && !ncu_dmu_pio_data[36] ||
               !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 eight byte writes are allowed.

Note – j2d_p_xx(cmds only, not data) are delayed by 1 clock from ncu_dmu_piodata[63:0]

1.14.4.15 NCU-DSN Timing Diagram

FIGURE 1-8 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;*

1.14.4.16 NCU to DSN Command Header Info

TABLE 1-44 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

1.14.4.17 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 1-45 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

1.14.5 DSN-DMU Interface

The DSN-DMU interface is left as is, TABLE 1-46, and the DSN block adapts the new SIU and NCU interfaces to this existing set of Signals.

TABLE 1-46 DSN-DMU 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 1-46 DSN-DMU 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 (csrs 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 1-46 DSN-DMU 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]		

1.15 Interface Layer Unit (ILU)

1.15.1 Overview

The following is a list of core, block and sub-block abbreviations which are frequently used throughout this chapter and should be used as a reference for increased readability:

TABLE 1-47 Abbreviation List

core	block	sub-block	description
DMU			Data Manager Unit
	DSN		DMU SIU/NCU Interface Unit
	CRU		CSR Register Unit with all the DCCs
	CLU		Cache Line Unit
		CRM	Cache Line Receive Manager
		CTM	Cache Line Transmit Manager
	CMU		Context Manager Unit
		RCM	Receive Context Manager
		TCM	Transmit Context Manager
	IMU		Interrupt Manager Unit
	RMU		Record Manager Unit
		LRM	Link Receive Manager

TABLE 1-47 Abbreviation List *(Continued)*

core	block	sub-block	description
		RRM	Receive Record Manager
	TMU		Transaction Manager Unit
		DIM	Data Ingress Manager
	ILU		Interface Layer Unit
		IIL	Ingress Interface Layer
		EIL	Egress Interface Layer
		CIB	CSR Interface Block with DCCD, DCCS
		ISB	Interface Score Board
PEU			PCI Express Core
	PTL		PCI-Express Transaction Layer
		IDB	Ingress Data Buffer
		IHB	Ingress Header Buffer
		ITL	Ingress Transaction Layer
		EDB	Egress Data Buffer
		EHB	Egress Header Buffer
		ETL	Egress Transaction Layer
		CTB	CSR Transaction Block with DCCD, DCCS, DCM
		RSB	Request Score Board
	PLP		PCI-Express Link/Physical Layer

The ILU operates at the IO clock rate, as opposed to the PTL where a time domain crossing is implemented. The ILU provides the interface between the DMU and the PEU core’s transaction layer and the header and data buffers in the PEU core. The ILU interfaces with the TMU, CMU, IMU, CRU blocks in the DMU and the PTL in the PEC. The main function of the ILU block is processing TLP header records, moving payload data between the DMU, PEC cores’ data buffers, releasing and processing data buffer credits, keeping track of the header and data buffer credits as a receiver of a PCI Express device, providing a CSR interface for the PTL to the OpenSPARC T2 PEU CSR rings, acting as an agent for the PTL to interface with the IMU and CMU.

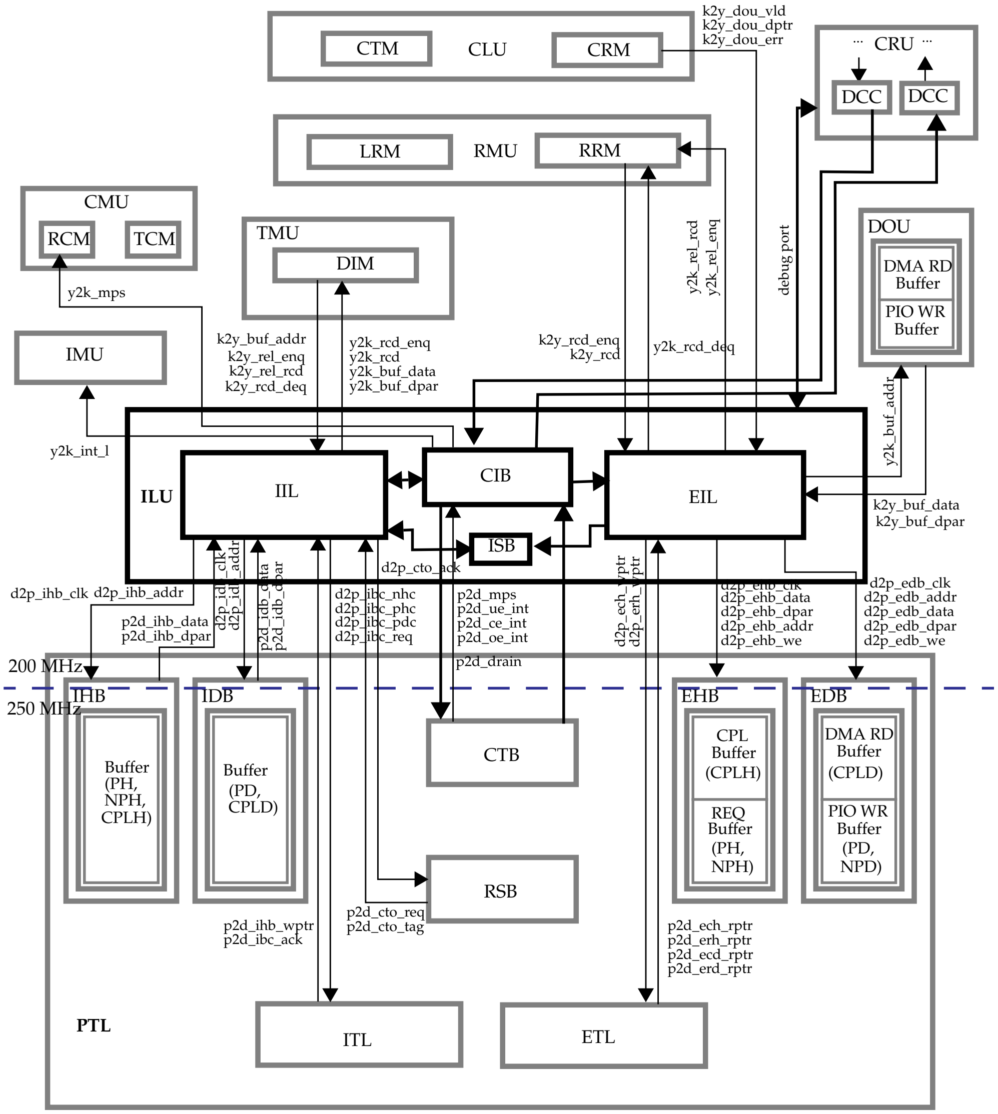
The ILU block functions are accomplished by the following four sub-blocks.

- **Ingress Interface Layer (IIL)** - It supports the transfer of TLPs from the PTL to the TMU. It pushes transaction records to the TMU and supports the pulling of payload data by the TMU from the IDB within the PTL. It collects and processes the release records from the TMU to keep track of the PCI Express receiver's header and data credits and passes the information down to the PTL. It checks and reports the data parity correctness of TLP headers from the IHB.

- **Egress Interface Layer (EIL)** - An analogous header-push/data-pull protocol transfers packets from the TMU to the ILU. It manages the spaces of the EDB, EHB buffers within the PTL. It pushes TLP headers and payload into the EHB and EDB. It generates PIO transaction credit and DMA read buffer release records and pushes them to the RMU. It is responsible for aligning data pulled from the cache line oriented data buffer in the DMU to the packed data buffer in the PEC.
- **CSR Interface Block (CIB)** - It provides an synchronous interface for the CSR ring connection between the two clock domains. It provides CSRs to log errors and generates interrupt request. It acts as an agent for the PTL to interface with the IMU and CMU.
- **Interface Score Board (ISB)** - It tracks outstanding non-posted PIO requests. It stores outgoing non-posted PIO requests two low address bits addr[3:2] to substitute the low_addr[3:2] in the corresponding PIO completion, which is used for data alignment in TMU. It's used to form timed out PIO completions when it's in a drain state.

1.15.2 Block Diagram

FIGURE 1-9 ILU Block Diagram



1.15.3 Functional Description

The ILU block has the following responsibilities in the PEC core:

- Provides a synchronous DCC ring connection between the two clock domains for the PEC core to talk to the DCCs in the DMU CRU block, provides CSRs to log errors and generates interrupt request, acts like an agent for the PTL for interrupt request processing
- On the ingress side of the pipeline:
  - Pulls and processes the TLP header records from the IHB in the PTL block and pushes the new formed records to the TMU
  - Converts all different types of non-posted unsupported requests into one type just called unsupported request to the DMU (data is dropped by the PTL if there is payload associated)
  - Converts all types of unsuccessful PIO Cpl status to "unsupported req" status
  - Checks the pulled header record's parity, loges and reports an error if there is an error
  - Accepts the PIO completion time out request from the PTL and generates appropriate completion record to be arbitrated by the ingress record pipeline
  - Provides an interface for the TMU block to pull payload data out of the IDB
  - Processes data buffer release records from the TMU block
  - Keeps track of the header and data buffer credits for different types and passes the information to the PTL block for PCI-E transaction layer flow control
  - Passes the value of the max. payload size to the CMU from the PEC core control CSR
  - When it's in the drain state, generates PIO completions for all the outstanding non-posted PIO requests with "unsupported request" status, no more processing of IHB record and completion time out request from PTL until it comes out of the drain state
- On the egress side of the pipeline:
  - Manages the buffer allocation for the EHB and EDB in the PTL
  - Processes the header records pushed from the TMU and computes the data parity for the header records, then pushes them together to the EHB
  - Keeps track of DOU DMA Rd buffer status (data availability and error status on cache line basis)
  - Pulls the payload (if data is available on DOU for DMA read return) from the DOU and pushes them to the EDB in the PTL
  - Aligns the data from the cache line oriented non-packed DMU data to the packed 16-byte wide data in the EDB
  - Generates release records to the TMU for PIO transaction credit and DMA read data buffer credit

- Set "EP" bit in "DMA Rd CplD" record to EHB if any of the associated payload cache lines is marked as "error" on DOU DMA Rd buffer status
- When it's in the drain state, drain the records in the record FIFO (posts them on the SBD if they are non-posted PIO requests, drops them on the floor otherwise, the associated payload is also drained from DOU and dropped on the floor)

1.15.4 Interface Signals

The following signal interface table summarizes the connections between the ILU and other blocks, sub-blocks, and cores with which it communicates. The src/dest field of the interface table specifies the connection to the sub-block level. All signals into and out of the ILU source or terminate within sub blocks IIL, EIL, CIB, ISB or block ILU.

TABLE 1-48 ILU Signal Interface

Signals	Width	Direction	Src/Dst	Function/Comment
ILU-DMU Interface				
ILU-TMU Interface (please refer to *OpenSPARC T2 Soc Microarchitecture Specification, Part 1 of 2*				
CSR Ring Interface				
k2y_csr_ring_out	32	In	CRU/CIB	CSR ring
y2k_csr_ring_in	32	Out	CIB/CRU	CSR ring
ILU-RMU Interface				
y2k_rel_rcd	9	In	ILU/DEM	The credit release record
y2k_rel_enq	1	In	ILU/DEM	The enqueue signal for the y2k_rel_rcd
ILU-CLU Interface				
k2y_dou_vld	1	In	CRM/ILU	Valid of a Cacheline in Egress DOU DMA Cpl Buffer
k2y_dou_dptr	5	In	CRM/ILU	Egress DOU DMA Cpl Buffer Cacheline address
k2y_dou_err	1	In	CRM/ILU	Data error status in k2y_dou_dptr cacheline
ILU-CMU Interface				
y2k_mps	3	Out	CSR/CMU	max payload size passed from PEC CSR, the encoding is as same as specified in the PCI Express spec.
ILU-IMU Interface				
y2k_int_l	1	Out	CSR/IMU	interrupt req for PEC core (level based)
ILU-DOU Interface				
y2k_buf_addr	8	Out	EIL/DOU	The DMU egress data buffer read address

TABLE 1-48 ILU Signal Interface *(Continued)*

Signals	Width	Direction	Src/Dst	Function/Comment
k2y_buf_data	128	In	DOU/EIL	16 byte data from DMU egress data buffer
k2y_buf_dpar	4	In	DOU/EIL	4-bit parity from DMU egress data buffer
Debug Interface				
k2y_dbg_sel_a	6	In	CRU/ILU	port a debug select
k2y_dbg_sel_b	6	In	CRU/ILU	port b debug select
y2k_dbg_a	8	Out	ILU/CRU	port a debug vector
y2k_dbg_b	8	Out	ILU/CRU	port b debug vector
ILU-PTL Interface				
Ingress Header Buffer Interface				
d2p_ihb_clk	1	Out	IIL/PTL	IHB clk from 200MHz domain
d2p_ihb_addr	6	Out	IIL/PTL	binary read address to IHB
p2d_ihb_wptr	7	In	PTL/IIL	gray-coded write pointer to IHB
p2d_ihb_data	128	In	PTL/IIL	4DW (or 3DW + 1DW reserved TLP header
p2d_ihb_dpar	4	In	PTL/IIL	parity bits for p2d_ihb_data
Ingress Data Buffer Interface				
d2p_idb_clk	1	Out	IIL/PTL	IDB clk from 200MHz domain
d2p_idb_addr	8	Out	IIL/PTL	read pointer for IDB
p2d_idb_data	128	In	PTL/IIL	payload data from IDB
p2d_idb_dpar	4	In	PTL/IIL	parity bits for p2d_idb_data
Ingress Buffer Credit Interface				
d2p_ibc_req	1	Out	IIL/PTL	req for ingress buffer credits
p2d_ibc_ack	1	In	PTL/IIL	ack for ingress buffer credits
d2p_ibc_nhc	8	Out	IIL/PTL	ingress buffer credit for non-posted header (NPH)
d2p_ibc_phc	8	Out	IIL/PTL	ingress buffer credit for posted header (PH)
d2p_ibc_pdc	12	Out	IIL/PTL	ingress buffer credit for posted data (PD)
Completion Timeout Interface				
p2d_cto_req	1	In	PTL/IIL	req for PIO cpl time out rcd generation
p2d_cto_tag	5	In	PTL/IIL	lower five bits PIO tlp tag for cpl timeout rcd generation
d2p_cto_ack	1	Out	IIL/PTL	ack for p2d_cto_req
Status Interface				

TABLE 1-48 ILU Signal Interface *(Continued)*

Signals	Width	Direction	Src/Dst	Function/Comment
p2d_drain	1	In	PTL/CIB	drain state
p2d_mps	3	In	PTL/CIB	max payload size from PEC control CSR
p2d_ue_int	1	In	PTL/CIB	uncorrectable error interrupt request
p2d_ce_int	1	In	PTL/CIB	correctable error interrupt request
p2d_oe_int	1	In	PTL/CIB	other error interrupt request
Egress Header Buffer Interface				
d2p_ehb_clk	1	Out	EIL/PTL	EHB clk from 200MHz domain
d2p_ech_wptr	6	Out	EIL/PTL	gray-coded write pointer to EHB-CPL buffer
d2p_erh_wptr	6	Out	EIL/PTL	gray-coded write pointer to EHB-REQ buffer (PIOs)
p2d_ech_rptr	6	In	PTL/EIL	gray-coded read pointer to EHB-CPL buffer
p2d_erh_rptr	6	In	PTL/EIL	gray-coded read pointer to EHB-REQ buffer
d2p_ehb_we	1	Out	EIL/PTL	write strobe for EHB
d2p_ehb_addr	6	Out	EIL/PTL	binary write pointer for EHB
d2p_ehb_data	128	Out	EIL/PTL	4DW (or 3DW + 1DW reserved TLP header
d2p_ehb_dpar	4	Out	EIL/PTL	parity bits for d2p_ehb_data
Egress Data Buffer Interface				
d2p_edb_clk	1	Out	EIL/PTL	EDB clk from 200MHz domain
p2d_ecd_rptr	8	In	PTL/EIL	gray-coded read pointer to EDB Cpl buffer. MSB is roll over bit, reset value is 8’b0
p2d_erd_rptr	8	In	PTL/EIL	gray-coded read pointer to EDB Req buffer. MSB is roll over bit, reset value is 8’b0
d2p_edb_we	1	Out	EIL/PTL	write strobe for EDB
d2p_edb_addr	8	Out	EIL/PTL	write pointer for EDB
d2p_edb_data	128	Out	EIL/PTL	payload data to EDB
d2p_edb_dpar	4	Out	EIL/PTL	parity bits for d2p_edb_data
CSR Interface				
d2p_csr_req	1	Out	CIB/PTL	CSR ring request
p2d_csr_ack	1	In	PTL/CIB	CSR ring acknowledge
d2p_csr_rcd	96	Out	CIB/PTL	CSR ring
p2d_csr_req	1	In	PTL/CIB	CSR ring request
d2p_csr_ack	1	Out	CIB/PTL	CSR ring acknowledge

TABLE 1-48 ILU Signal Interface *(Continued)*

Signals	Width	Direction	Src/Dst	Function/Comment
p2d_csr_rcd	96	In	PTL/CIB	CSR ring
Spare interface				
d2p_spare	5	Out	ILU/PTL	Spare DMU to PEC connections
p2d_spare	5	Out	PTL/ILU	Spare PEC to DMU connections
ILU internal sub block interface				
eil2isb_log	1		EIL/ISB	valid tag to set on the ISB
eil2isb_tag	5		EIL/ISB	PIO request's tlp_tag[4:0]
eil2isb_low_addr	2		EIL/ISB	PIO request's two low address bits addr[3:2]
iil2isb_tag	5		IIL/ISB	PIO response's tlp_tag[4:0]
isb2iil_vld	1		ISB/IIL	valid bit output from ISB to IIL
iil2isb_clr	1		IIL/ISB	clear scoreboard entry indexed as iil2isb_tag
isb2iil_low_addr	2		ISB/IIL	corresponding low address from ISB to IIL
iil2cib_par_err	1		IIL/CIB	IIL informs CIB for header record data parity error
cib2iil_drain	1		CIB/IIL	CIB tells IIL to go to drain state
cib2eil_drain	1		CIB/IIL	CIB tells EIL to go to drain state

1.15.5 Transaction Flow

The major functions in processing the pulled header records from the IHB in the PTL on the ingress side are

- clear SBD entry
- record type out vs. the type in
- setting align addr[5:2] = {00, low_addr[3:2]} for PIO Cpl/CplD
- PCIE flow control header buffer credit collection
- convert PIO Cpl status (other than successful) to "unsupported request" status

Please note, the low_addr[3:2] mentioned above is its corresponding non-posted PIO requests low_addr[3:2], which is saved on the scoreboard ISB.

The major transaction flow functions on the ingress side of the pipeline are summarized in TABLE 1-49.

TABLE 1-49 Transaction Summary on the Ingress Side flowing in/out of the ILU Block

transaction type in	encoded type in	transaction type out	encoded type out	clear SBD	align addr[5:2]	header buffer credit	modify PIO Cpl status
DMA MRd	0x00000b	DMA MRd	0x00000b	NO	n/a	YES - NPH type	n/a
DMA MRdLk	0x00001b	DMA MRdLk	0x00001b	NO	n/a	YES - NPH type	n/a
DMA MWr	1x00000b	DMA MWr	1x00000b	NO	n/a	YES - PH type	n/a
DMA IORd	0000010b	unsupported request	0001001b	NO	n/a	YES - NPH type	n/a
DMA IOWr	1000010b	unsupported request	0001001b	NO	n/a	YES - NPH type	n/a
DMA CfgRd	000010xb	unsupported request	0001001b	NO	n/a	YES - NPH type	n/a
DMA CfgWr	100010xb	unsupported request	0001001b	NO	n/a	YES - NPH type	n/a
Msg	0110xxxb	Msg	0110xxxb	NO	n/a	YES - PH type	n/a
PIO Cpl (other than successful)	0001010b	PIO Cpl	0001010b	YES	YES	NO	YES, convert to "unsupported request"
PIO Cpl (successful status)	0001010b	PIO Cpl	0001010b	YES	YES	NO	NO
PIO CplD	1001010b	PIO CplD	1001010b	YES	YES	NO	n/a
PIO Cpl (timeout status) through p2d_cto_req interface	n/a	PIO Cpl	1001010b	YES	n/a	NO	n/a
PIO Cpl (unsupported request status) when it's in drain state	n/a	PIO Cpl	1001010b	YES	n/a	NO	n/a

---

Note – According to the PCI-Express spec, all reserved status values (i.e. 011, 101, 110, and 111) must be treated as "Unsupported Request". Moreover, CLU maps the two unreserved status (unsupported request and completer abort) to one type - JBUS bus error. Therefore, ILU converts PIO Cpl with status other that successful to unsupported request status.

Note – OpenSPARC T2 PEU as a HW will NOT retry the PIO config request if the associated completion's status is "cfg retry", but set a CSR config retry status bit and log both PIO request header and associated cpl header. At meantime, the Cpl record is trapped in PTL and a PIO timeout Cpl generation is requested from PTL to ILU through p2d_cto_req. Therefore, ILU will NOT receive a PIO Cpl with "cfg retry" status.

Note – The original PCI Express completion packets include the associated requesting lower address[6:0]. However, the lower address is 7'b0 if the Cpl/CplD is resulted from an io/cfg rd/wr request. Therefore, ILU substitutes the low address [3:2] with the value retrieved from ISB for correct data alignment which is done in the TMU block in DMU. In order to align the PIO partial read completion data to the first row (16-byte) of a cache line, the IIL sets the align address [5:4] to 2'b0. If it's a PIO block read completion, the formula is still true. This way, there is no need to propagate the align address up to the CTM in the header records and the CTM will always pull PIO partial read completion data from the first row of the cache line.

The major functions in processing the header records dequeued from the EIL record FIFO on the egress side are

- determine which buffer in the EHB to push the header record to
- log the tlp tag and low_addr[3:2] on the ISB for non-posted PIO requests
- pull data from DOU
- align data (all data pulled from DOU need data alignment, thus it's not shown in TABLE 1-50)
- determine which buffer in the EDB to push the payload to
- generate release record for DMA RD buffer in the DOU
- generate release record for PIO transaction credit

The major transaction flow functions on the egress side of the pipeline are summarized inTABLE 1-50.

TABLE 1-50 Transaction Summary on the Egress Side flowing in/out of the ILU Block

transaction type in	encoded type in	log to ISB	the buffer in EHB record is pushed to	pull data & the buffer in EDB data is pushed to	DMA buffer release	PIO credit release
PIO MRd	0x00000b	YES	REQ buffer	NO	NO	NO
PIO MWr	1x00000b	NO	REQ buffer	YES, PIO WR buffer	NO	YES
PIO IORd	0000010b	YES	REQ buffer	NO	NO	NO
PIO IOWr	1000010b	YES	REQ buffer	YES, PIO WR buffer	NO	NO
PIO CfgRd	000010xb	YES	REQ buffer	NO	NO	NO
PIO CfgWr	100010xb	YES	REQ buffer	YES, PIO WR buffer	NO	NO
DMA Cpl	0001010b	NO	CPL buffer	NO	NO	NO
DMA CplD	1001010b	NO	CPL buffer	YES, DMA RD buffer	YES	NO
DMA CplLk	0001011b	NO	CPL buffer	NO	NO	NO

Note – For "DMA CplD" record written into EHB, the "EP" bit is set if any of the associated payload cache lines in DOU is in error.

Note – There are two cases which would cause an error in a cache line in DOU DMA RD buffer: (1) DMA Rd response error from JBC to DMU (no data written into DOU); (2) j2d_d_data_err is asserted for any data cycles associated with a DMA Rd response. Therefore, it doesn't guarantee the data parity correctness for a DOU cache line in error. However, when ETL pulls data out of DOU DMA Rd data buffer, it checks data parity and reports and logs the error if a data parity error is detected. Moreover, JBC detects the two error cases, reports and logs the error when the error is detected. Thus, it might cause two place to report and log the same error for those two error cases.

1.15.6 Passing Data Across Clock Domains

There are four mechanisms used here to pass data across the clock domains at the ILU-PTL across clock domain interface.

- Synchronizer scenario
- gray-coded buffer pointers
- Auto-update req-ack interface
- Demand-based req-ack interface

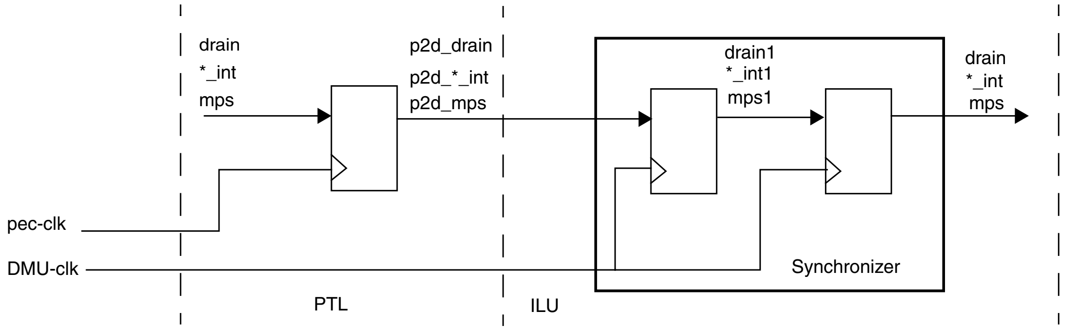
1.15.6.1 Synchronizer Scenario

This scenario is used to pass data of

- p2d_ue_int
- p2d_ce_int
- p2d_oe_int
- p2d_mps[2:0]
- p2d_drain

from the PTL to ILU because these data are stable after they are set.

FIGURE 1-10 Asynchronous Clocks & Synchronizer Scenario



1.15.6.2 Gray-Coded Buffer Pointers

The both ingress and egress header buffers (IHB and EHB) are accessed through binary buffer pointers. The buffer access signals are d2p_ihb_addr, d2p_ehb_addr, d2p_idb_addr, and d2p_edb_addr. However, the buffer pointers across the clock domain for buffer management are gray-coded.

Here is the list of signals across the clock domains in gray-code:

- p2d_ihb_wptr

- d2p_ech_wptr
- d2p_erh_wptr
- p2d_ech_rptr
- p2d_erh_rptr
- p2d_ecd_rptr
- p2d_erd_rptr

1.15.6.3 Auto-Update Req-Ack Interface

On the ingress side of the pipeline, the IIL collects the credits for the PCI-E transaction flow control variable CREDITS_ALLOCATED for the behavior of the PCI-E receiver (please refer to the PCI Express spec for the definition of the variable CREDITS_ALLOCATED in *PCIE Standard Specification Version 1.0a*).

It collects credits for the header types of posted (PH), non-posted (NPH); for the data type of posted (PD). There is no credit collection for non-posted data type (NPD) because the non-posted payload is not stored in the IDB at all (the non-posted TLP requests are unsupported requests and the PTL drops them). There is no credit collection for completion header (CPLH) and data (CPLD) because OpenSPARC T2 PEU advertises them as infinite. All the credit counters are initialized as zero after reset. The PTL will take the credit values received from the ILU, plus the credit values collected by itself, and plus the corresponding initial credit, then pass them to the Data Link Layer.

On the egress side, the PCI Express transaction flow control mechanism is adopted to manage the TLP RD buffer segment of the EDB. The PTL collects the credit whenever it transmits completion payload to the link layer. Then the PTL passes the credit to the EIL for TLP RD buffer credit tracking.

The credit values mentioned above are passed across the clock domains by means of an auto-update req-ack interface. The signals are listed below:

On the ingress side:

- d2p_ibc_req
- d2p_ibc_nhc[7:0]
- d2p_ibc_phc[7:0]
- d2p_ibc_pdc{11:0]
- p2d_ibc_ack

FIGURE 1-11 shows the schematic block diagram of the req-ack interface.

Note – It's a auto-update req-ack interface if the new_req is tied to 1b in FIGURE 1-11, schematic req-ack interface block diagram. Otherwise, it's a demand-based req-ack interface.

FIGURE 1-11 Schematic req-ack Interface across Clock Domains

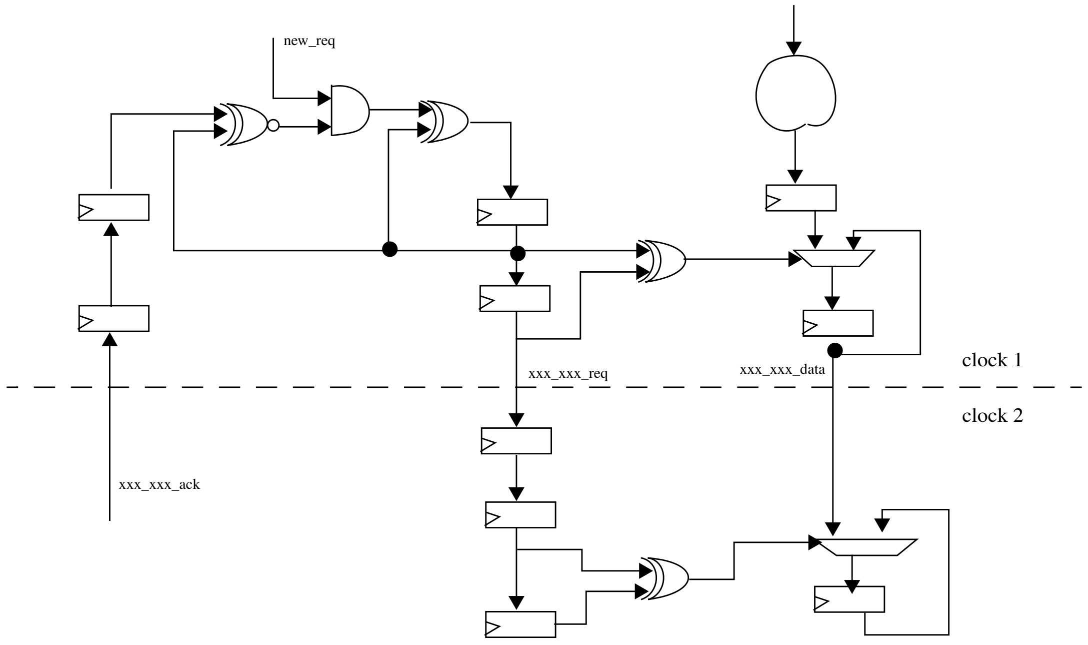
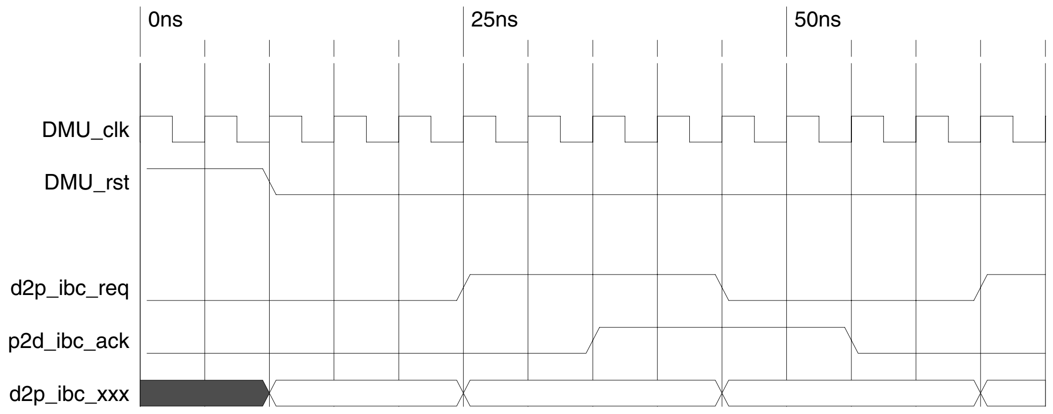


FIGURE 1-12 Auto-update req-ack on the Ingress Side Timing Diagram



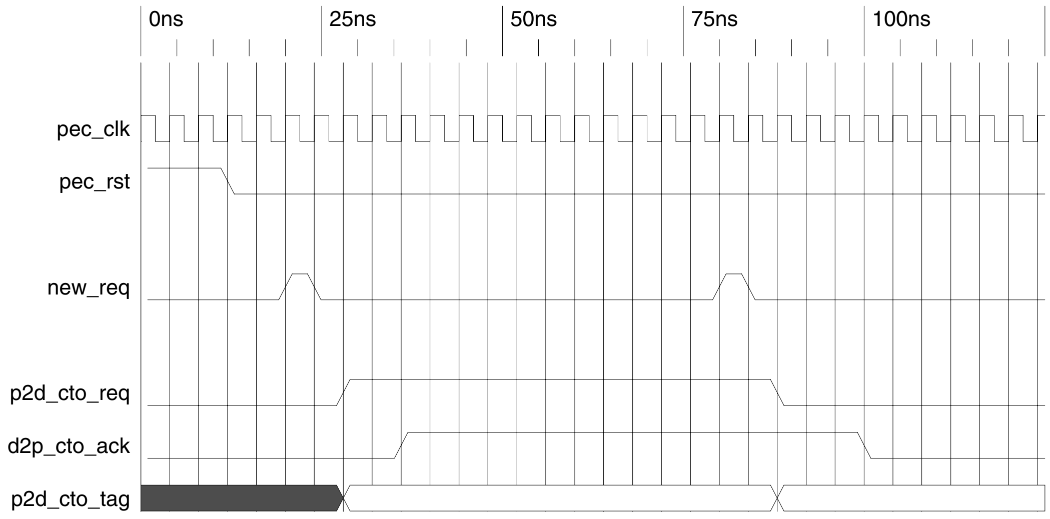
1.15.6.4 Demand-Based Req-Ack Interface

When the timer is expired in the PIO scoreboard in the PTL for a outstanding non-posted PIO request before it receives a corresponding completion, the PTL requires the IIL to generate a corresponding PIO time out error completion record to propagate up to JBus. The completion time out request is passed from the PTL to the ILU across the clock domains by the mean of demand-based req-ack interface. The signals are listed below:

- p2d_cto_req
- p2d_cto_tag[4:0]
- d2p_cto_ack

Please refer to FIGURE 1-11 for the schematic block diagram.

FIGURE 1-13 Demand-based req-ack Timing Diagram

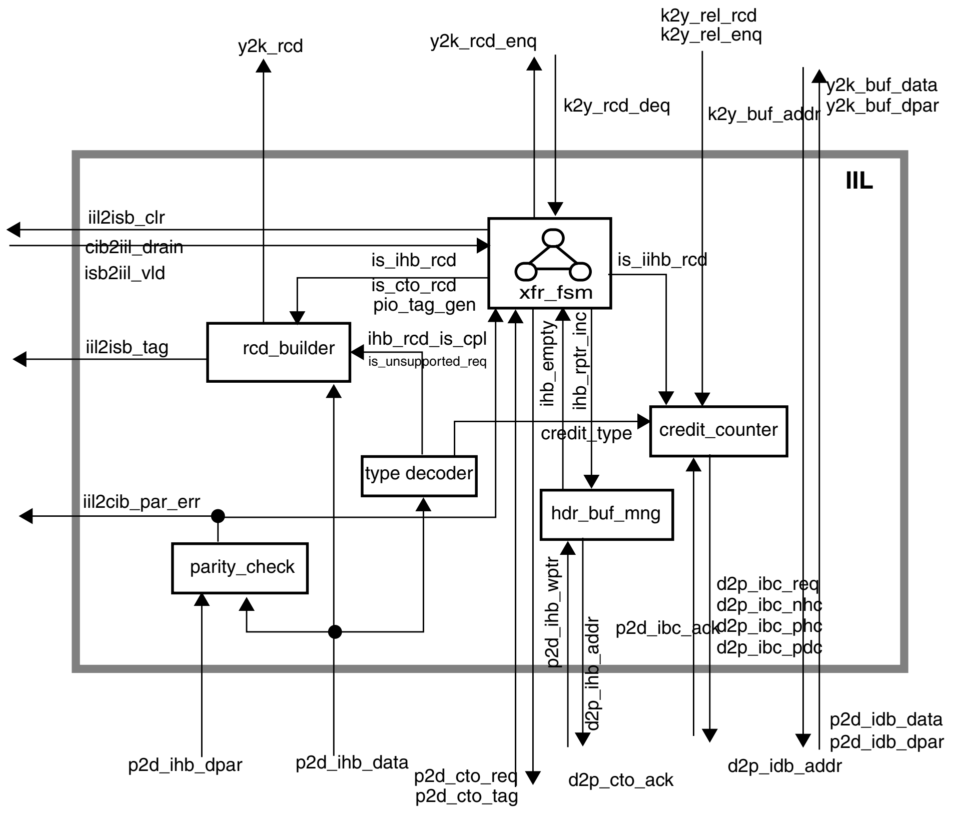


1.15.7 IIL Sub Block

In the ingress direction the IIL reads a header records out of the IHB and checks its data parity. If there is a parity error, it's a fatal error (the IHB is trashed), the IIL drops the record and reports the error to the CIB sub block, at meantime, the ILU goes into a drain state (this will be discussed in more detail later). Otherwise, the IIL pushes the processed header records to the TMU record FIFO if the FIFO is not full, and increases the appropriate header buffer credit counters. The IIL clears the ISB entry for completions. The IIL services PIO completion timeout request from PTL by generating PIO completion record with an internal encoded timeout status value. For PIO Cpl records, IIL converts all types of unsuccessful PIO Cpl status to "unsupported req" status. The TMU pulls payload data out of the IDB through the ILU. The TMU sends a release record to the ILU after pulling the payload from the IDB to release the IDB's packet allocation. The IIL uses the release records to update the appropriate data buffer credit counters.

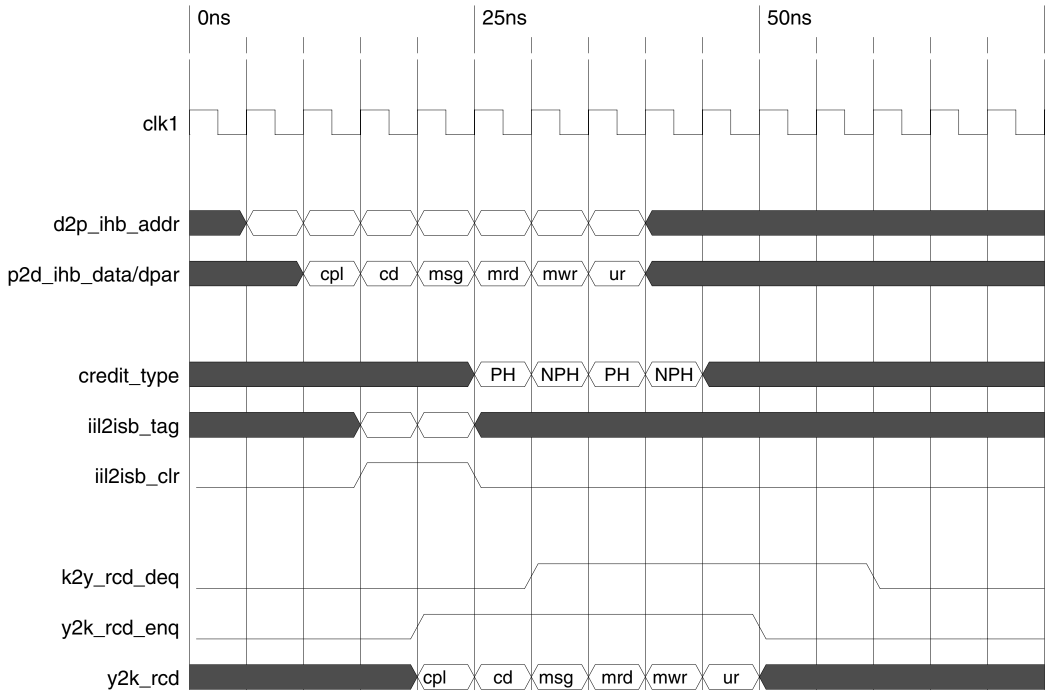
1.15.7.1 IIL Block Diagram

FIGURE 1-14 IIL Block Diagram



1.15.7.2 IIL Timing Diagram

FIGURE 1-15 IIL Timing Diagram



cpl - PIO Cpl
cd - PIO CplD
msg - Msg
mrd - DMA MRd
mwr - DMA MWr
ur - unsupported request (DMA io/cfg rd/wr)

1.15.7.3 Assumptions

Note – OpenSPARC T2 PEU will use "Infinite" flow control advertisement for types of CPLH, CPLD, and NPD. Thus, there is no need for the IIL to collect credits for CPLH and CPLD (the IIL will never see NPD type payload).

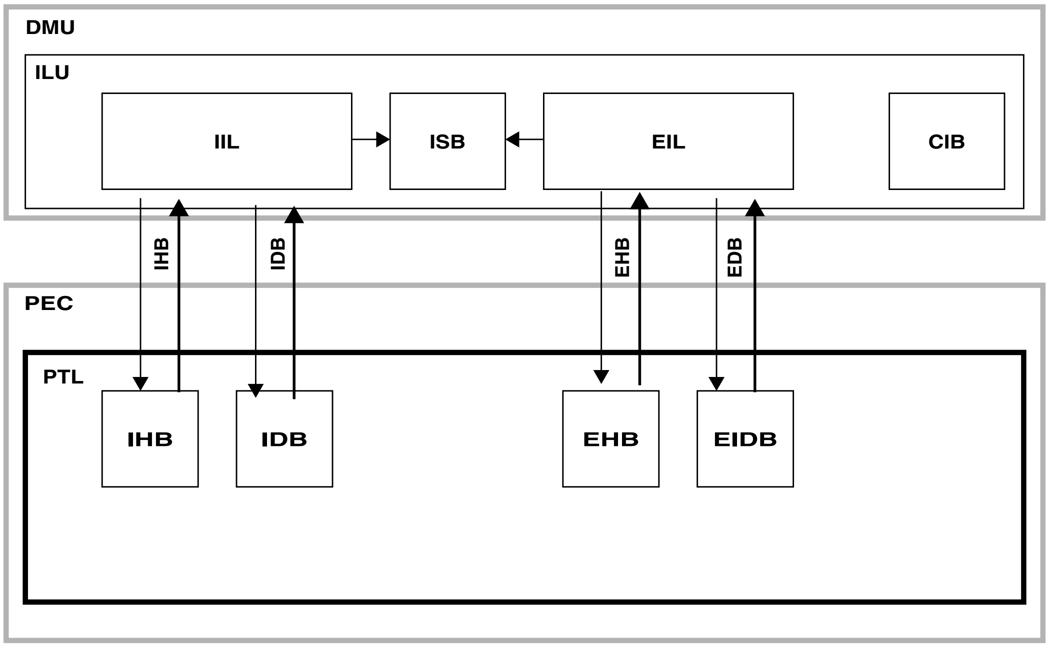
---

1. The PTL will trap (drop on the floor) incoming unsupported posted request (MsgD) header records and associated payload.
2. The PTL will trap the associated payload (not stored in the IDB) for the incoming unsupported non-posted requests, which are transaction types of DMA IOWr, DMA CfgWr.
3. The PTL will identify and trap unsolicited, malformed, and "cfg retry" error completion header records and associated payload.
4. The DIM makes data buffer release on a row basis (16 bytes data), which means it sends a release record to the IIL every time the DIM pulls a row of data out of the IDB.

1.15.8 ILU PEU Interface

1.15.8.1 Block Diagram

FIGURE 1-16 ILU-PTL Connection Diagram



1.15.8.2 ILU-PTL Signal Interface

TABLE 1-51 ILU-PTL Signal Interface

Signals	Width	Direction	Src/Dst	Function/Comment
IHB-ILU interface				
d2p_ihb_clk	1	In	ILU/IHB	DMU clock
d2p_ihb_addr	6	In	ILU/IHB	IHB read address
p2d_ihb_data	128	Out	IHB/ILU	IHB read data
p2d_ihb_dpar	4	Out	IHB/ILU	IHB read data parity
IDB-ILU interface				
d2p_idb_clk	1	In	ILU/IDB	DMU clock
d2p_idb_addr	8	In	ILU/IDB	IDB read address
p2d_idb_data	128	Out	IDB/ILU	IDB read data
p2d_idb_dpar	4	Out	IDB/ILU	IDB read data parity
EHB-ILU interface				
d2p_ehb_clk	1	In	ILU/EHB	DMU clock
d2p_ehb_we	1	In	ILU/EHB	EHB write enable
d2p_ehb_addr	6	In	ILU/EHB	EHB write address
d2p_ehb_data	128	In	ILU/EHB	EHB write data
d2p_ehb_dpar	4	In	ILU/EHB	EHB write data parity
EDB-ILU interface				
d2p_edb_clk	1	In	ILU/EDB	DMU clock
d2p_edb_we	1	In	ILU/EDB	EDB write enable
d2p_edb_addr	8	In	ILU/EDB	EDB write address
d2p_edb_data	128	In	ILU/EDB	EDB write data
d2p_edb_dpar	4	In	ILU/EDB	EDB write data parity

1.15.8.3 Data Buffers

There are four data buffers modelled as part of the PTL, which are

- ingress header buffer (IHB)
- ingress data buffer (IDB)
- egress header buffer (EHB)

- egress data buffer (EDB)

1.15.8.4 Buffer Management

IHB And IDB

IHB and IDB are treated as a single circular buffer.

As a consumer of IHB, ILU needs to detect IHB's emptiness.

The IHB's emptiness detection is through its read/write pointers. There are 64 entries in IHB. Therefore, it's 6-bit IHB write address (d2p_ihb_addr). However, the IHB write pointer passed from PTL to ILU is 7-bit (p2d_ihb_wptr) with the MSB as a roll-over bit. ILU keeps its own 7-bit IHB read pointer with the MSB as a roll-over bit too. It's empty if the 7-bit read/write pointers are the same.

The IHB's fullness detection is through global PCIE flow control credit mechanism. It's managed on the transmit's device side. It's out of scope of this spec.

There is no need for IDB emptiness detection because it's guaranteed that the transaction associated payload is ready to pull in IDB when the transaction header is processed down the pipeline.

The IDB's fullness detection is through global PCIE flow control credit mechanism. It's managed on the transmit's device side. It's out of scope of this spec.

EHB And EDB

EHB and EDB are treated as two circular buffers (half/half). The low address space (one half) is partitioned for completion (DMA Cpl/CplD) records and their associated payload (named as ECH & ECD buffer); the high address space (the other half) for request (PIOs) records and their associated payload (named as ERH & ERD buffer).

For each header circular buffer, ILU passes its write pointer to EIL (d2p_ech_wptr for ECH and d2p_erh_wptr for ERH); ETL passes its read pointer to ILU (p2d_ech_rptr for ECH and p2d_erh_rptr for ERH). The MSB in these read/write pointers is a roll-over bit.

For each data circular buffer, ETL passes its read pointer to ILU (p2d_ecd_rptr for ECD and p2d_erd_rptr for ERD). The MSB in these read pointers is a roll-over bit.

As a producer of EHB and EDB, ILU needs to detect their fullness for both circular buffers. ILU keeps its own set of write pointers to ERD and ECD with the MSB as a roll-over bit. A circular buffer is full if their roll-over bits in read/write pointers vary and the rest are the same.

As a consumer of EHB and EDB, ETL needs to detect the emptiness for the two header circular buffers. However, there is no need to detect the emptiness for the two data circular buffers because it's guaranteed that the transaction associated payload is ready to pull in EDB when the transaction header is processed in ETL. A circular buffer is empty if their read/write pointers are the same.

The PEC record from the IIL to the DIM in the TMU (across ILU-TMU interface) at TABLE 1-48.

1.15.8.5 IIL Type Decoder

The type decoder block takes the 7-bit type field, the completion status field in the records pulled from the IHB in the PTL as inputs to sort the transaction as:

- unsupported non-posted requests (DMA IO/Cfg Rd/Wr)
- PIO Completions (Cpl)
- non-posted requests (DMA MRd)
- posted requests (DMA MWr)

If it's a IHB record header data parity error, the record will be discarded. Different actions will be taken when the transaction falls into a different type, which is notified by the outputs of the type decoder. The outputs are

- ihb_rcd_is_cpl
- is_unsupported_req
- credit_type[1:0] - encoding for three types of PCI-E header

TABLE 1-52 Encoded Signal Credit_type[1:0]

signal	encoding
credit_type[1:0]	10 - non-posted header (NPH)
	01 - posted header (PH)
	00 - completion header (CPLH)
	11 - illegal

TABLE 1-53 IIL Type Decoder Block Functions

record type	input type[6:0]	output ihb_rcd_is_cpl	output is_unsupported_req	output credit_type [1:0]	action modify type field value	action modify status value
DMA MRd	0x00000b	0	0	10b	No	n/a
DMA MRdLk	0x00001b	0	0	10b	No	n/a
DMA MWr	1x00000b	0	0	01b	No	n/a
DMA IORd	0000010b	0	1	10b	Yes, to 0001001b	n/a
DMA IOWr	1000010b	0	1	10b	Yes, to 0001001b	n/a
DMA CfgRd	000010xb	0	1	10b	Yes, to 0001001b	n/a
DMA CfgWr	100010xb	0	1	10b	Yes, to 0001001b	n/a
Msg	0110xxxb	0	0	01b	No	n/a
PIO Cpl (unsuccessful status)	0001010b	1	0	00b	No	yes, to "UR"
PIO Cpl (successful status)	0001010b	1	0	00b	No	no
PIO CplD (status must be 3’b000)	1001010b	1	0	00b	No	no

1.15.8.6 Drain State

The purpose of the drain state is to unstall the egress pipeline from JBC to PEC. This is required to enable internal PIOs issued by software to propagate to the CSR ring. In JBC, external and internal PIOs are stored in a single queue. If the pipeline stalls, internal PIOs could be blocked in the queue. Software would then be unable to read/clear the internal CSRs. The drain state alleviates this blockage and allows CSR access to Software.

There are three scenarios, which will get the IIL and EIL into a drain state.

- The PTL detects an egress header/data parity error from EHB/EDB.

- The IIL detects an ingress header parity error from IHB. The IIL signals the error to CIB sub block. Then, the CIB sub block sends "drain" to the IIL through the block internal interface signal cib2iil_drain. At meantime, the CIB sub block sends "drain" to the EIL also through the signal cib2eil_drain, which is connected from an internal flop inside of CIB.

When it's in the drain state, on the egress side the EIL will process the records normally except that the dequeued header records and pulled payload data are not stored in the EHB and EDB, instead, they are dropped on the floor.

When it's in the drain state, on the ingress side the IIL will stop pulling any header records from the IHB. The xfr_fsm block will run into an infinite loop to look into the ISB block to take off the outstanding PIOs and generate PIO completion header records with the completion status field as "unsupported request". The completion records will be pushed to the record FIFO in the DIM by the xfr_fsm. The reason of completing PIOs is to free the processors who issued the PIOs from waiting PIO completions and be able to issue new PIOs to internal CSRs.

When any one of the three scenarios happens, the detector of the error will set its corresponding CSR error status bit if enabled and request for interrupt if enabled. Here are the steps SW will take when it receives the interrupt:

- a PIO read to the CSR to find out the problem
- a PIO write to the CSR to clear the interrupt enable bit to prevent further interrupt requests sourced from the same error
- a PIO write to the CSR to clear the error status bit
- reset the chip (it will get the ILU out of the drain state after the link is up)
- a PIO write to the CSR to set the interrupt enable bit

Normal operations will then resume.

Note – Any new PIO completion timeout generation request from PTL (p2d_cto_req) will be ignored by ILU when it's in the drain state. No more IHB records processing after it goes into the drain state. The only way coming out of the drain state is through a SW reset.

1.15.8.7 PCI-E Flow Control Credit Processing

For PCI-E flow control credit, the IIL only collects two types of header credit, which are non-posted header (NPH) and posted header (PH), 1 type of data credit, which is posted data (PD).

Note – The PTL will collect PCI-E flow control credit for PH header type and PD data type by receiving unsupported posted requests. Then, as a receiver, the PTL adds its own collected PH and PD credits (the counters are initiated as the initial credits and the collected credits are increased) and the ILU collected PH and PD credits accordingly as the total PH and PD PCI-E flow control credits. The PTL adds the ILU collected NPH credit with the initial credits as the total NPH PCI-E flow control credit.

The credit counter block serves this purpose.

The header credits are sourced internally. There are two separate header type credit incrementing counters to accumulate the header releases by processing the input signal credit_type[1:0] from the type decoder along with the signal is_ihb_rcd from the xfr_fsm. The two header credit counters are d2p_ibc_phc (posted header credit), d2p_ibc_nhc (non-posted header credit).

The PD data credit is collected when the TMU sends the buffer release record to the ILU after the TMU pulls the data out from the IDB on a DMA MWr operation. The PD data credit counter is d2p_ibc_pdc (posted data credit). No data credit is collected when the TMU sends the buffer release record to the ILU after the TMU pulls the data out from the IDB on a PIO CplD operation.

Note – The TMU sends DMA write buffer release record down to the ILU whenever it pulls a row of data out of the IDB (essentially, the k2y_rel_enq is asserted). Thus, it's one data credit for one release record. The IIL will consume one release record by incrementing the data credit counter by one if it's a PD type.

1.15.8.8 PIO Completion Time Out

PTL determines if there is a need to time out an outstanding non-posted PIO request. If the answer is yes, it sends a request with the timed out PIO associated tag to the IIL through the signals p2d_cto_req and p2d_cto_tag[4:0]. When the IIL sees the request, it will generate a completion header record with the completion status field set as time out error (value of 3'b111, which is for OpenSPARC T2 PEU internal use only).

The IIL arbitrates this generated timeout PIO completion record with higher priority over the records in the IHB (by the way, the drain state has the highest priority) and inserts it to the record pipeline by pushing it to the record FIFO resident in the DIM if there is space in the FIFO. After pushing the timeout PIO completion record to the FIFO, the IIL will acknowledge it back to the PTL by asserting d2p_cto_ack. At meantime, the IIL will clear the ISB entry with the associated tag.

Note – The PTL will NOT request another PIO completion timeout generation to the ILU before it gets the acknowledge back from the ILU for the current request.

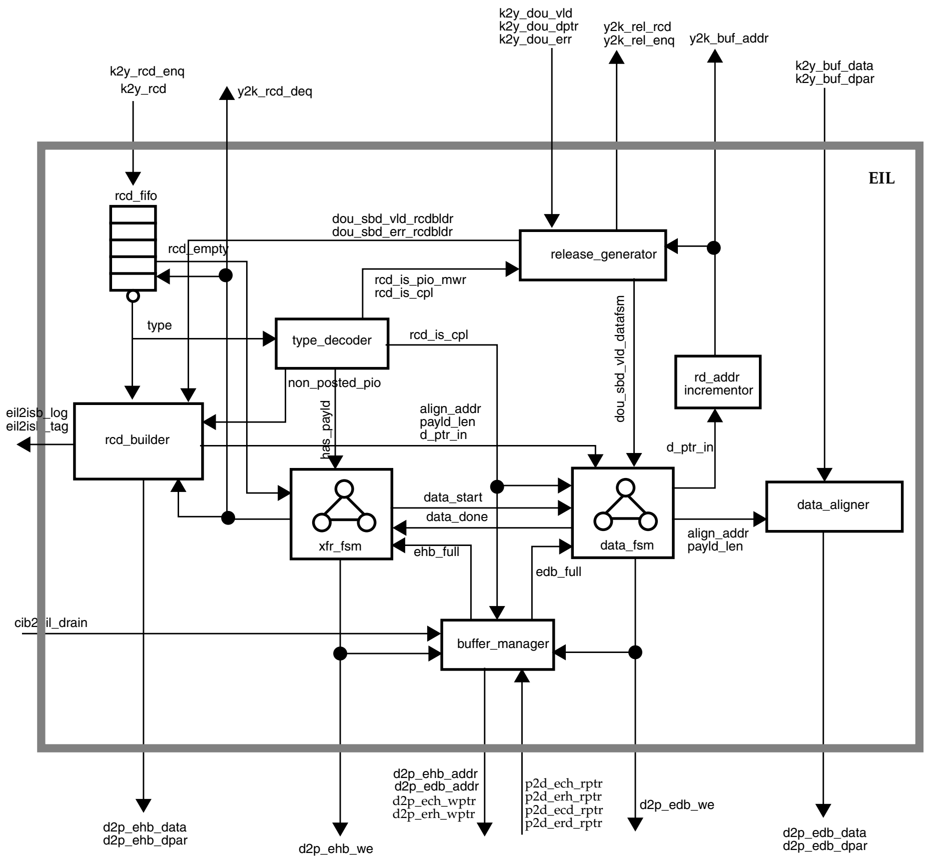
1.15.9 EIL Sub Block

In the egress direction, the header records are pushed by the DMU to an record FIFO resident in the EIL. These records include a pointer to the first payload cacheline address to the DMU's DOU. Using this pointer, the EIL manages the pulling of data and parity from the DOU. The EIL manages the buffer credit and allocation for the EHB and EDB. The EIL keeps track of DOU DMA Rd buffer status (data availability and error status on cache line basis). The EIL pushes the header records to the EHB and pulls the associated payload out of the DOU if it's available in DOU and aligns the data from the cache line oriented format to a 16-byte packed format, then pushes them to the EDB. The "ep" bit in the "DMA CplD" header record is set when any of the associated payload cachelines' DOU status is in error. The EIL logs the tlp_tag[4:0] and two low address bit addr[3:2] to the ISB sub block if the processed record is an non-posted PIO. The ILU sends a release record for transaction type of DMA CplD to the RMU after a cache line payload has been pulled from DMA Rd buffer to release DOU DMA Rd buffer space. The ILU also sends a release record to the RMU for PIO transaction credit if the processed record is a PIO MWr.

Note – There is no need to send release records to the RMU after pulling PIO write associated data because the PIO write buffer in the DOU is globally managed by the JBC to only issue up to 16 PIO requests and the size of the PIO write buffer is 16 cache lines.

1.15.9.1 EIL Block Diagram

FIGURE 1-17 EIL Block Diagram

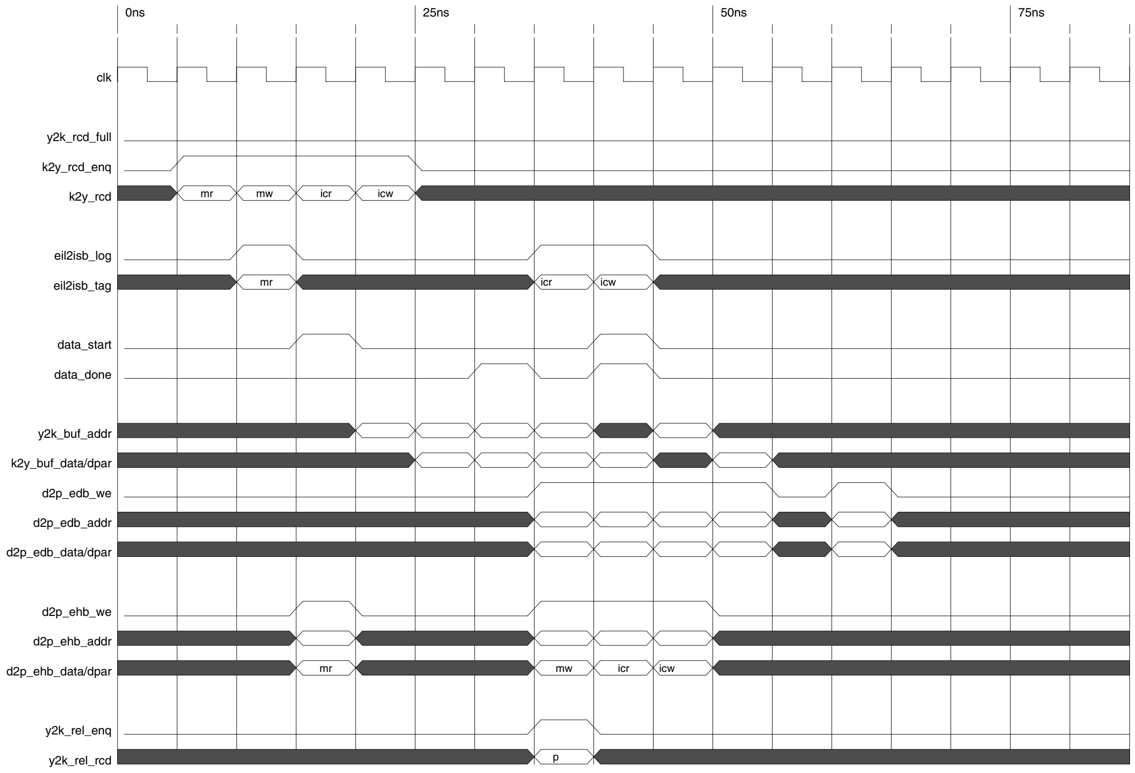


NOTES:

- The size of rcd_fifo queue is a depth of 4.

1.15.9.2 EIL Timing Diagram

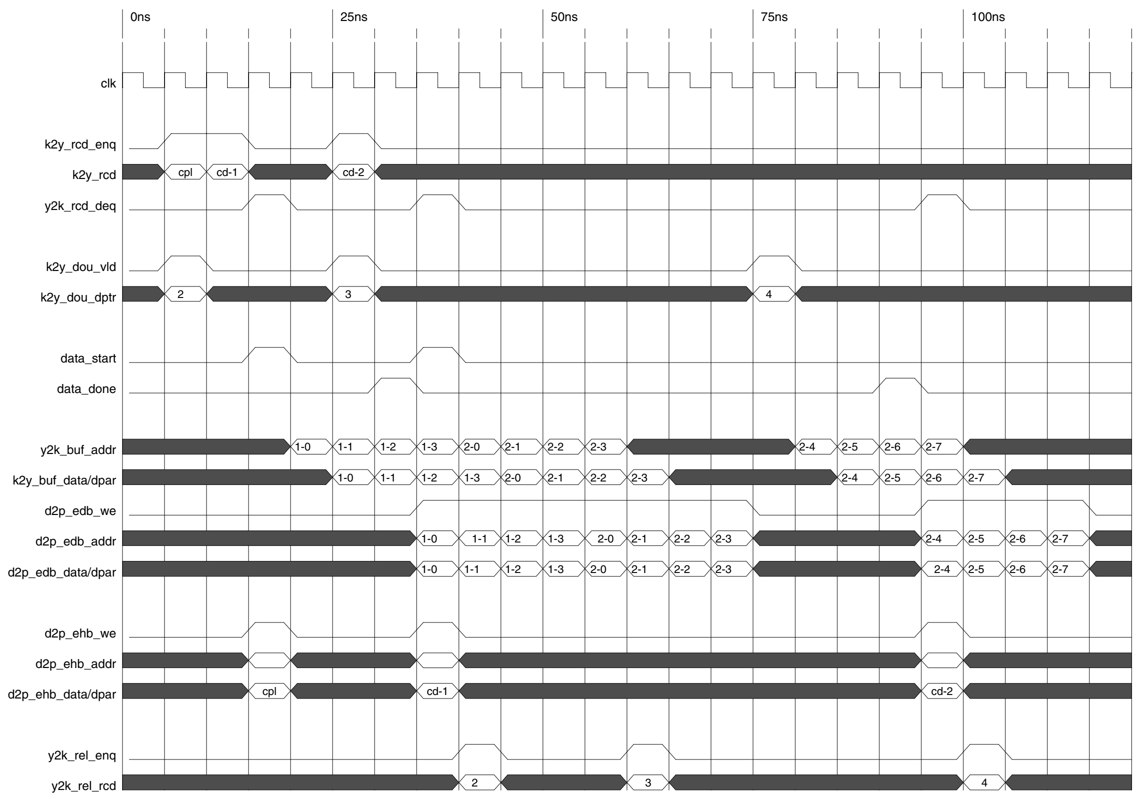
FIGURE 1-18 EIL Timing Diagram: PIO requests



mr - PIO MRd
mw - PIO block MWr
icr - PIO IORd / CfgRd
icw - PIO IOWr / CfgWr

p - release record for PIO transaction credit

FIGURE 1-19 EIL Timing Diagram: DMA Completions



cd-1 - DMA CplD with one cacheline payload, dptr = 2
cd-2 - DMA CplD with two cacheline payload, dptr = 3
cpl - DMA Cpl

1.15.9.3 EIL Record Format

For the record formats in/out of the EIL, please refer to:

The PEC record from the RMU/RRM to the EIL (across DMU-ILU interface) at Transaction Manager Unit (TMU).

d_ptr field is the starting DOU's cache line address for the associated record's payload.

The EHB record from the EIL to the EHB in the PTL (across PTL-ILU interface) at Interface Layer Unit (ILU)

In forming EHB record,

- TD field is zero filled for all types of records;
- EP field is zero filled for all types of records except CplD. For CplD record, if there is no errors in the associated payload read out from DOU, EP field is 1'b0, otherwise, EP field is 1'b1. Please refer to Data Out Unit (DOU) for DOU data status interface between CLU and ILU.
- Reserved fields are zero filled for all types of records except the last DW header (bits [31:0]) in 3-DW TLP headers, which are don't cares because PTL doesn't transmit this DW header for 3-DW TLP headers.

Along with a EHB record being pushed to EHB is 4-bit header data parity. The header data parity is calculated in EIL from the newly-formed EHB record and it's odd parity.

1.15.9.4 EIL Type Decoder

The type decoder block takes the 7-bit type field in the current record of the rcd fifo in the EIL to determine the outputs

- if it is a completion (DMA completion with/without data) (rcd_is_cpl)
- if it has associated payload (has_payld)
- if it's a non-posted PIO request (non_post_pio).
- if it's a PIO memory write request (rcd_is_pio_mwr)

Output rcd_is_cpl tells release generator block to generate DMA read data buffer release record at the time when a release record is enqueued; tells buffer manager block which buffer space in EHB/EDB to allocate for the record and the associated data if the record has payload.

Output has_payld tells xfr_fsm to assert data_start to trig data_fsm to move data.

Output non_post_pio tells rcd_builder to log the pio_tag and two bits of lower_addr[3:2] to scoreboard ISB.

Output rcd_is_pio_mwr tells release generator block to generate PIO credit release record at the time when a release record is enqueued;

TABLE 1-54 EIL Type Decoder Block Functions

transaction type in	**input type[6:0]**	**output has_payld**	**output rcd_is_cpl**	**output non_post_pio**	**output rcd_is_pio_mwr**
PIO MRd	0x00000b	0	0	1	0
PIO MWr	1x00000b	1	0	0	1
PIO IORd	0000010b	0	0	1	0
PIO IOWr	1000010b	1	0	1	0

TABLE 1-54 EIL Type Decoder Block Functions *(Continued)*

transaction type in	input type[6:0]	output has_payld	output rcd_is_cpl	output non_post_pio	output rcd_is_pio_mwr
PIO CfgRd	000010xb	0	0	1	0
PIO CfgWr	100010xb	1	0	1	0
DMA Cpl	0001010b	0	1	0	0
DMA CplD	1001010b	1	1	0	0
DMA CplLk	0001011b	0	1	0	0

1.15.9.5 EIL Buffer Manager

The buffer manager manages buffer space and allocation for both the EHB and EDB.

The EHB is segmented into completion header buffer (Cpl, CplD, and CplLk header records go here) and request header buffer (PIO requests go here). To manage the EHB, the buffer manager:

- keeps track of its binary write pointers to each header buffers called ech_wptr & erh_wptr;
- converts its write pointers ech_wptr & erh_wptr to gray-code pointers d2p_ech_wptr & d2p_erh_wptr to be passed to PTL for header buffer emptiness determination in PTL;
- converts gray-coded header buffer read pointers p2d_ech_rptr & p2d_erh_rptr passed from PTL to binary read pointers ech_rptr & erh_rptr;
- compares the binary read/write buffer pointers to determine header buffers' fullness ech_full & erh_full;
- tells xfr_fsm about the header buffer fullness through the signals ehb_full by muxing out of ech_full & erh_full based on the in-processing transaction type.

The EDB is segmented into completion data buffer (DMA read data) and request data buffer (PIO write data). PTL can always assume that at the time when PTL processes a record from EHB, its associated payload is in EDB. Thus, no EDB buffer write pointers passed from ILU to PTL since there is no need for EDB buffer emptiness check in PTL. To manager the EDB, the buffer manager:

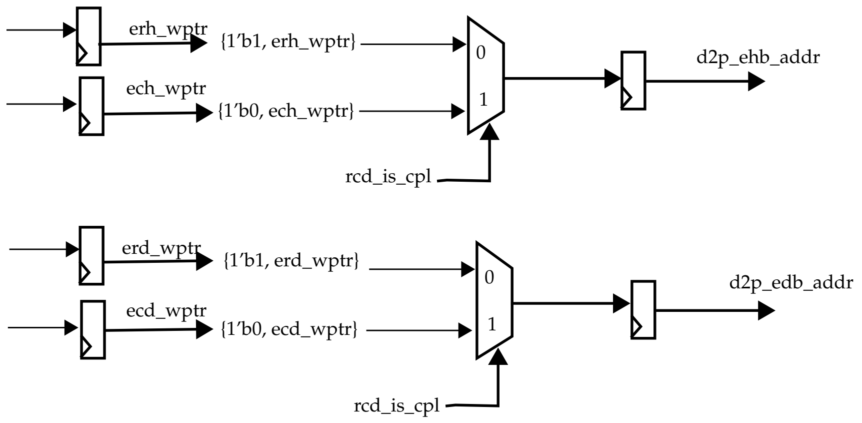
- keeps track of its binary write pointers to each data buffers called ecd_wptr & erd_wptr;
- converts gray-coded data buffer read pointers p2d_ecd_rptr & p2d_erd_rptr passed from PTL to binary read pointers ecd_rptr & erd_rptr;
- compares the binary read/write buffer pointers to determine data buffers' fullness ecd_full & erd_full;

- tells data_fsm about the data buffer fullness through the signals edb_full by muxing out of ecd_full & erd_full based on the in-processing transaction type.

When it's in the drain state, the buffer manager always informs the xfr_fsm and data_fsm that there is room to process more records to let the EIL to drain the record fifo and pull data out of the DOU. However, the processed records and pulled data from DOU are discarded in EIL when it's in the drain state. Therefore, the d2p_ehb_we will be deasserted right away at the time it goes into the drain state and d2p_edb_we will be deasserted at most three cycles later since data path is pipe lined.

FIGURE 1-20 illustrates some signal relations involved within the buffer manager block.

FIGURE 1-20 Signal Relations in Buffer Manager



1.15.9.6 EIL Finite State Machines

There are two finite state machines in EIL, which are xfr_fsm and data_fsm. If there is payload in the current processing record, the xfr_fsm trig data_fsm through signal data_start. The data_fsm starts to read data out of DOU, to align the pulled data, and to push the data to EDB in a pipeline fashion. In order to stream line the data path when it's in steady state, data_fsm asserts data_done before last write to EDB. At the time when last DOU read is injected into the pipeline and there are enough space in EDB for the remaining writes, or the last EDB write is injected into the pipeline, the data_fsm tells the xfr_fsm that it's ready to process next record by asserting data_done.

At the next cycle of data_done asserted from the data_fsm, the xfr_fsm enqueues the EHB record along with its data parity to EHB if there is space in EHB. Since data_done is asserted ahead of the time of last data beat write to EDB, the EHB record may be written into EHB early than the last data beat to EDB.

The worst case is that the EHB record is pushed to EHB three cycles earlier than the first; 4 cycles earlier than the last write to EDB since there are three more stages in data pipeline than the record pipeline and the earliest time data_fsm can assert data_done is at the time of last read to DOU and two more writes to EDB. However, d2p_ech_wptr/d2p_erh_wptr is updated 1 cycle later than d2p_ehb_we asserted. Therefore, from ETL side point of view, header arrives EHB is two cycles earlier than the first; three cycles earlier than the last data beat write to EDB in DMU clock speed.

1.15.9.7 EIL Data Alignment

The payload is stored differently in the DMU-DOU and PEC-EDB data buffers. In the EDB, the data is stored in packed 16 byte wide format. In the DOU the data is stored in non-packed 16 byte wide format and it's a cache line oriented data buffer.

Before the payload, which is pulled from DOU by EIL, is pushed to EDB, data alignment is needed.

A double word (DW) is 4-byte, a term from PCI Express spec. For a DOU buffer read, there are 4-DW (16-byte) data (k2y_buf_data[127:0]) and 4-bit parity (k2y_buf_dpar[3:0]). The 4-bit parity covers the 4-DW data in the way of

k2y_buf_dpar[3] <--> k2y_buf_data[127:96]

k2y_buf_dpar[2] <--> k2y_buf_data[95:64]

k2y_buf_dpar[1] <--> k2y_buf_data[63:32]

k2y_buf_dpar[0] <--> k2y_buf_data[31:0]

The length in the transaction records is in DW and the virtual address is DW aligned address. Therefore, the data alignment is on the DW granularity. Since the parity is a DW parity, parity bits are aligned with the data.

Note – When pulling data from the DOU, the EIL will only read to the row entries in the DOU which has valid data. The valid DW data stored in the DOU is contiguous for each transaction record.

Note – Since CLU always writes the single data beat for a PIO Wr 16 transaction into the first row of a cache aligned data block in the PIO data buffer, EIL will always pull only one data beat from the first row of a cache aligned data block in the PIO data buffer for PIO non-block writes (less than a cacheline).

The number of data beats to pull from the DOU for the transaction is

- NUM_DB_DOU = (length + addr[3:2]) >> 2 + (|end_addr[3:2])

where length is the value of the length field from the PEC record, addr[3:2] is the A[3:2] for PIO request and lower_addr[3:2] for DMA completion from the PEC record. The end_addr[3:2] = addr[3:2] + length and "|" is the "Reduction OR" operator.

The number of data beats to write to the EDB for the transaction is

- NUM_DB_EDB = length >> 2 + (|length[1:0])

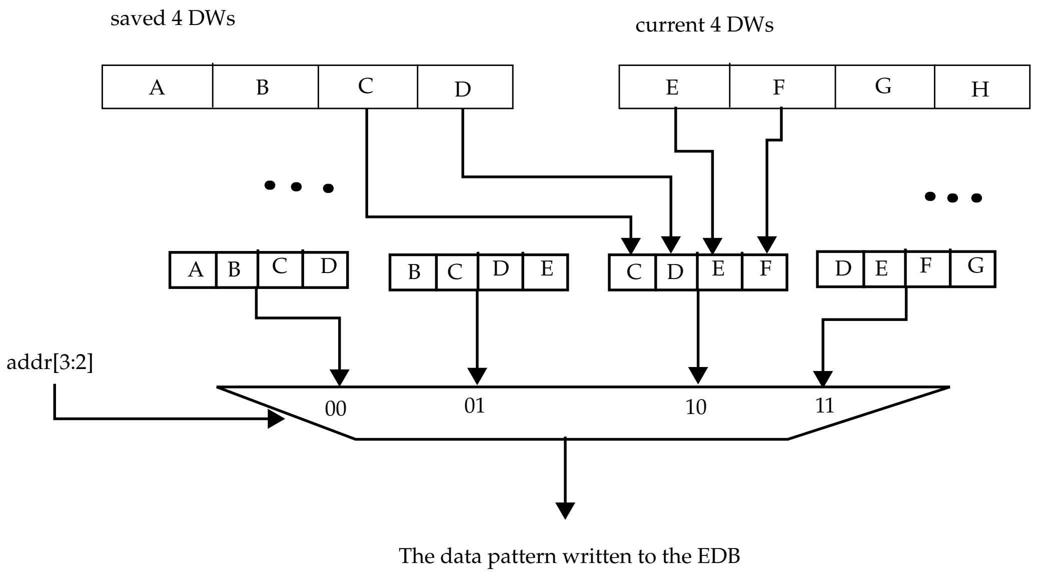
The first four DWs data pulled from the DOU is saved in the "saved four DWs" flops. For cases of NUM_DB_DOU > 1, when the second four DWs data pulled from the DOU is arrived at the "current four DWs" the first row of data will be pushed to the EDB. Thereafter, the "current four DWs" is loaded to "saved four DWs" flop and the next new read is loaded to "current four DWs". This pattern continues until all the payload is transferred.

For cases of NUM_DB_DOU = 1, only one data beat written to EDB for that transaction. However, in order to stream line the data path, the "current four DWs" flop might loaded with the next transaction's associated payload. Thus, the data written into EDB will be the combination of the payload for that transaction and the next one if the align address is not 16-byte aligned.

Note – For a transaction in a case of NUM_DB_DOU = 1 and not 16-byte aligned address, if a parity error happens on the next transaction's associated payload, it might affect this transaction's associated payload being pushed to EDB. Since the ETL does not distinguish between valid data DWs and non valid and it expects that ALL parity is good for all four DWs, ETL will detect the parity error one packet too early. This is not the end of the world because parity errors are fatal so if we detect the error one packet too early it is still okay, we just stop transmitting one packet ahead of where we would have.

The data pattern written into the EDB from the "saved four DWs" and "current four DWs" is shown in FIGURE 1-21.

FIGURE 1-21 Data Pattern written to the EDB



1.15.9.8 EIL Release Generating

The PIO write data buffer in the DOU is maximum sized (16 cache lines) and it's global controlled by the JBC which will only issue up to 16 outstanding PIO requests. Therefore, there is no need to manage the space and the EIL will NOT make releases when it pulls the data out from the PIO write buffer in the DOU.

However, the DMA read data buffer in the DOU is managed by the CLU and the EIL will make releases whenever it finishes pulling the data out of a cache line in the DMA read data buffer in the DOU. Thus, the releases for the DMA read data buffer will always be a cache line.

The EIL will not only generate release record for DMA read data buffer in the DOU, but also generate release record for PIO transaction credit when it finishes processing a posted PIO write request (PIO memory write). Thus, there are two portions in the release record generated from the EIL, one of which is PIO transaction credit, the other is DMA read data buffer release. The release records are sent to the RMU block for further processing.

For the release record format from the ILU to the RMU, please refer to Record Management Unit (RMU).

When the signal rcd_is_pio_mwr from the type decoder and the data_done from the data_fsm are asserted at same time, a release record for PIO transaction credit only is generated from the EIL.

When rcd_is_cpl from the type decoder is asserted, whenever the read address bit[2] to the DOU (y2k_buf_addr[2]) flips, which implies a cache line is done, a release record will be generated for one cache line DMA read buffer. A release record of one cache line DMA read buffer will also be generated when data_done is asserted. The following summarizes the two cases that the release_generator generates a DMA read data buffer release record.

- rcd_is_cpl is asserted (it means that the EIL is processing a DMA CplD record), the y2k_buf_addr[2] flips.
- rcd_is_cpl is asserted, data_done is asserted.

Note – The ILU releases DMA read buffer on one cache line bases.

1.15.10 CIB Sub Block

The CIB provides an synchronous interface for the CSR ring connection between the two clock domains (ILU-PTL). The CIB provides CSRs to log its own header record data parity error and generates interrupt request to the IMU. Moreover, the CIB mirrors the PTL status registers and generates interrupt request to the IMU on the behavior of the both ILU and PTL (acts like an agent for the PTL).

Note – Data parity is NOT checked when the EIL moves payload from the DOU in the DMU to the EDB in the PEC. Data parity is checked when the IIL pulls header records out of the IHB in the PEC and error is logged in the CSR and reported through interrupt request to the IMU if there is a header record data parity error.

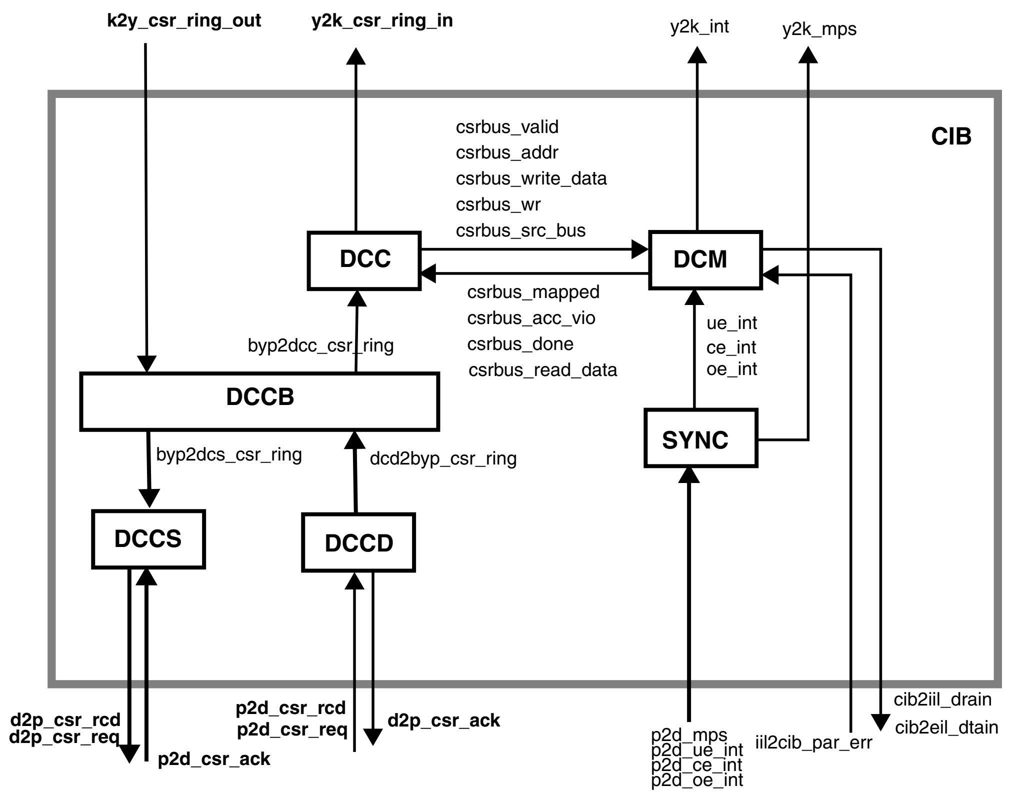
The number of CSRs which need to be accessible by software is contained in the DCM. An interface is provided for the CRU block to perform read and write accesses to the registers inside the DCM. This interface connects directly to the DCC in the DMU/CRU Block.

The SYNC block is for synchronizing some data passed from a different clock domain.

The DCCS (synchronizing DCC Source), DCCD (synchronizing DCC Destination), and DCCB (DCC Bypass) blocks are specialized DCCs from CSRtool.

The CIB logics are shown in FIGURE 1-22.

FIGURE 1-22 CIB Block Diagram



On errors, TABLE 1-55, ILU status bits will be set and an interrupt will be generated.

TABLE 1-55 ILU Status

Status	Reset[*]	Description
HEADER_PAR_ERR	0	Header data parity error
PEC_UE_ERR	0	Uncorrectable error from PEC
PEC_CE_ERR	0	Correctable error from PEC
PEC_OE_ERR	0	Other error from PEC

* Only on hard reset

1.15.11 ISB Sub Block

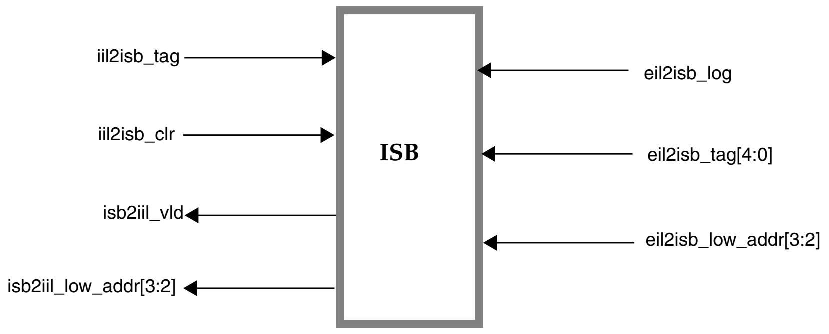
The ISB block is used to keep track of the outstanding non-posted PIO requests and store their two-bit lower address[3:2]. The tlp_tag[4:0] in the header record is used as an index to the score board. Thus, there are 32 entries on the ISB. The ISB provides an interface to set the entry indexed by a tag, which is connected to the EIL block. The ISB provides another interface to feed back the status of the entry and clear the entry indexed by a tag, which is connected to the IIL block.

Based on the PCI-E spec, 7-bit lower address field in Cpl/CplD is always 7'b0 for Cpl/CplD resulted from IO/Cfg read/write requests. Thus, DMU can't rely on the original value in this field to align data from IDB to DIU. Therefore, the two-bit lower address[3:2] is saved for egress PIO non-posted requests and read out from IIL for the corresponding Cpl/CplD to be substituted to lower address[3:2] in the Cpl/CplD for ingress PEC records.

Note – The ISB is sized as 32 entries because the tlp_tag[4:0] is used as an index to access the score board. The tlp_tag[4] tells the PIO read from the PIO write requests. Thus, no additional information needs to store on the score board to tell PIO read from PIO write. Due to the fact that there is only up to 16 outstanding PIO requests global controlled by the JBC, the ISB will never be full, will only be half full at the most.

The interfaces of the ISB are shown in FIGURE 1-23.

FIGURE 1-23 ISB Block Diagram



1.15.12 ILU Idle Check

ILU is in idle state under the following conditions:

- IIL sub block:
  - IHB is empty
  - xfrfsm is in idle state
- EIL sub block:
  - xfrfsm is in idle state
  - datafsm is in idle state
  - record FIFO is empty
- ISB sub block
  - no PIO transaction pending

The idle status flop ilu_is_idle is at the ILU top module and it's fed to a debug port.

1.16 Pin Mapping

This table shows the signal mapping between the DMU pin name and the new SIU or NCU name.

TABLE 1-56 Pin Mappings from Existing DMU 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		

TABLE 1-56 Pin Mappings from Existing DMU to DSN *(Continued)*

DMU name	SIU/NCU name	Description
j2d_d_wrack_tag[3:0]		
j2d_d_wrack_vld		
CRM Command Completion Port		
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		

1.17 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. DMU internal errors such as parity on the internal rams will be reported through the existing DMU mondo interrupt mechanism (tbd and devtsb rams). 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....

1.17.1 DSN/SII-SIO RAS Interface

The RAS features for this interface include:

1. The DMA write and PIO rd return FIFOs have 1 parity bit per 32 data bits, and the SOC RAS spec requires two 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 DMU-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 two 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_ids 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 ues 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 DMU.

1.17.2 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

1.17.3 DMU 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.

1.17.4 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 1-57 RAS Signals

Signal name	direction	Description
DSN to SII RAS Signals		
dmu_sii_parity[7:0]	output	two 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	two 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]

TABLE 1-57 RAS Signals *(Continued)*

Signal name	direction	Description
Note: any detected parity errors will be signaled to DMU 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]
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 OpenSPARC T2)
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 DMU 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).

1.17.5 Error Cases

TABLE 1-58 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 DMU	One less credit id to use in DMU, no corruption but possible DMU hang.

TABLE 1-58 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	two 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.

1.17.6 IOMMU RAS

TABLE 1-59 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.

1.17.7 No Syndrome Register in DSN

Consider these cases:

1. **DMA write data parity error.**

   SII logs address, write completes to l2$ with byte enables off, SW can determine what the device was doing during the write from SII address syndrome.

2. **PIO rd cpl and interrupt data parity error.**

   rd cpl, data is passed back to core and load buffer and it would log the address for reads (precise trap).

   For interrupts the NCU logs

3. **DMA rd data return**

   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,**

   On ingress NCU will log

   On egress (DMA rd cpls) the DSN drops this cacheline and does not return the credit id and data to the DMU. Since the DMU orders the DRCs 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**

   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,**

   The DSN drops this ack back to DMU and the DMU 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**

   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.

1.18 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.

1.19 Content and Status Registers (CSRs)

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.

1.19.1 CSR Address Decoding

The DMU address decoding will be in three 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.

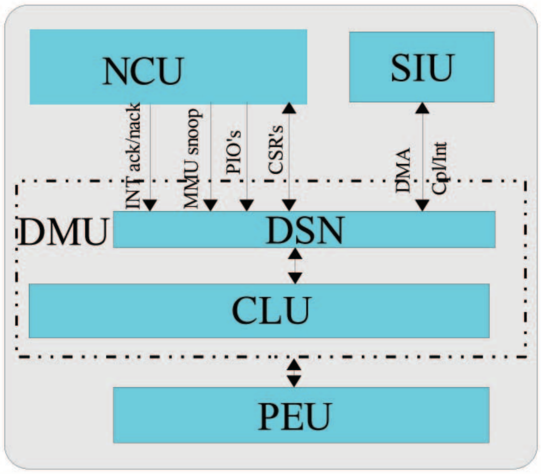
1.19.2 Content and Status Register (CSR) Related Pins

TABLE 1-60 Content and Status Register (CSR) Related Pins

Signal name	direction	Description
Ucb interface downs-tread		
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 ups-tread		
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

1.19.3 CSR Block Diagram

FIGURE 1-24 CSR Block Diagram



1.20 Transaction Ordering

This section describes the PIO and CSR ordering within the DSN and DMU blocks. The ordering between PIOs/CSRs 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 which is "tbd" cycles. Thus SW may determine ordering of invalidates and other PIO/CSRs.

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.

1.21 DEBUG Features

This will consist of three 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 DMU debug busses will continue to be used and new Signals from DSN will be sent to the DMU block and muxed out. New DSN Signals are listed in TABLE 1-62.

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.

1.21.1 Quiescent DMU/SII/SIO Interface

It is assumed that the NCU will be drained before the DMU is quiescent. Then the debug.v block will send a signal to the DMU to become quiescent. To manage this the DSN block will keep a 4 bit counter to track the number of outstanding DMA reads and writes, and a two 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.

1.21.2 Debug Busses

The DMU has existing A and B debug busses. These are eight 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 DMU debug select definitions (*OpenSPARC T2 Programmer's Reference Manual* registers DMU 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 three 1's and one 0 when the debug select busses are set to 0101.

1.21.3 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.

1.21.4 Debug Interface Signals

TABLE 1-61 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.

1.21.5 DSN Debug Signals

TABLE 1-62 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

TABLE 1-62 DSN Debug Signals *(Continued)*

Signal name	Bit number	Description
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 cpls
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]		
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 2

Miscellaneous I/O (MIO) Specification

This chapter contains the following sections:

2.1 Overview

This document describes OpenSPARC T2 MIO (Miscellaneous I/O) block which holds majority of non-SERDES I/Os of the chip. The I/Os 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 outputs 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.

2.1.1 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/Os 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. TABLE 2-1 shows all the I/Os 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/Os.

TABLE 2-1 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 2-1 MIO Pinlist *(Continued)*

Pin Name	I/O Type	Direction	Function	Share d	Description & Frequency	Src/Dest OpenSPARC T2 Block & Signal name(s)
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 2-1 MIO Pinlist *(Continued)*

Pin Name	I/O Type	Direction	Function	Share d	Description & Frequency	Src/Dest OpenSPARC T2 Block & Signal name(s)
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

TABLE 2-1 MIO Pinlist *(Continued)*

Pin Name	I/O Type	Direction	Function	Share d	Description & Frequency	Src/Dest OpenSPARC T2 Block & Signal name(s)
PWRON_RST_L	cmos 1.1v	Input	Reset	No	Power On Reset Level Signal	RST mio_rst_pwron_rst_l
FATAL_ERROR	cmos 1.1v	Output	Reset	No	Fatal Error has occurred 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 Burn-in 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 2-1 MIO Pinlist *(Continued)*

Pin Name	I/O Type	Direction	Function	Shared	Description & Frequency	Src/Dest OpenSPARC T2 Block & Signal name(s)
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 2-2 shows the sharing of pins between debug and other functionality.

TABLE 2-2 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 2-2 Sharing of Debug Pins with Other Pins *(Continued)*

Pin Name	Shared With Pin:/Pin Description	Select & Drive Enable	Src/Dest OpenSPARC T2 Block & Signal name(s)
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	Drive en = dbg1_mio_drv_en_muxtestpll_inp for 156:149 and 146: 103.	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 pins only NIU_DBG_DAT[22:21]	102:85, 148:147 have dbg1_mio_drv_en_op_only	
146:103	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]	Sel: 156: 124 - dbg1_mio_sel_niu_debug_mode	

TABLE 2-2 Sharing of Debug Pins with Other Pins *(Continued)*

Pin Name	Shared With Pin:/Pin Description	Select & Drive Enable	Src/Dest OpenSPARC T2 Block & Signal name(s)
102:91	Shared between output pins only PEU_DBG_BUS_A[2:0] PEU_DBG_BUS_B[7:0] PEU_DBG_CLK	123:91 - dbg1_mio_sel_pcix_debug_mode	
90:85	Not shared		
84	Input: PEU_CLK_EXT Scan Test Captures @ PEU	dbg1_mio_drv_en_muxtest_inp	**TCU**: mio_tcu_peu_clk_ext
83	Not shared	dbg1_mio_drv_en_op_only	
82:77	Input: NIU_CLK_EXT[5:0].Scan Test Captures @ NIU	dbg1_mio_drv_en_muxtest_inp	**TCU**: mio_tcu_niu_clk_ext[5:0]
76:75	Not shared		
74	Input: SCAN_EN	dbg1_mio_drv_en_muxtest_inp	**TCU**: mio_tcu_io_scan_en
73:43	Outputs: SCAN_OUT[30:0] Scan Out Dat	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 2-3 shows the functional categories and frequencies of the pins that are shared with the Debug Pins.

TABLE 2-3 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]
NIU_DBG_CLK[1:0]	Debug	Up to 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

TABLE 2-3 Shared Pins Functionality and Frequencies *(Continued)*

Pin Name	Functionality	Data Change rate
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)
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)

2.1.2 Internal Pullups/Pulldowns in MIO for Inputs

TABLE 2-4 shows the inputs in MIO that have pullups/pulldowns on them

TABLE 2-4 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

2.1.3 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 three categories:

1. Debug port signals from db1 module (166 wires @ cmp_clk launched off of io2x sync enables in db1).
2. Ramtest signals from tcu module (41 wires @ cmp clk /10 launched off of io2x sync enables in tcu).
3. Debug signals from DMU (16 wires @ iol2clk from db0 module).

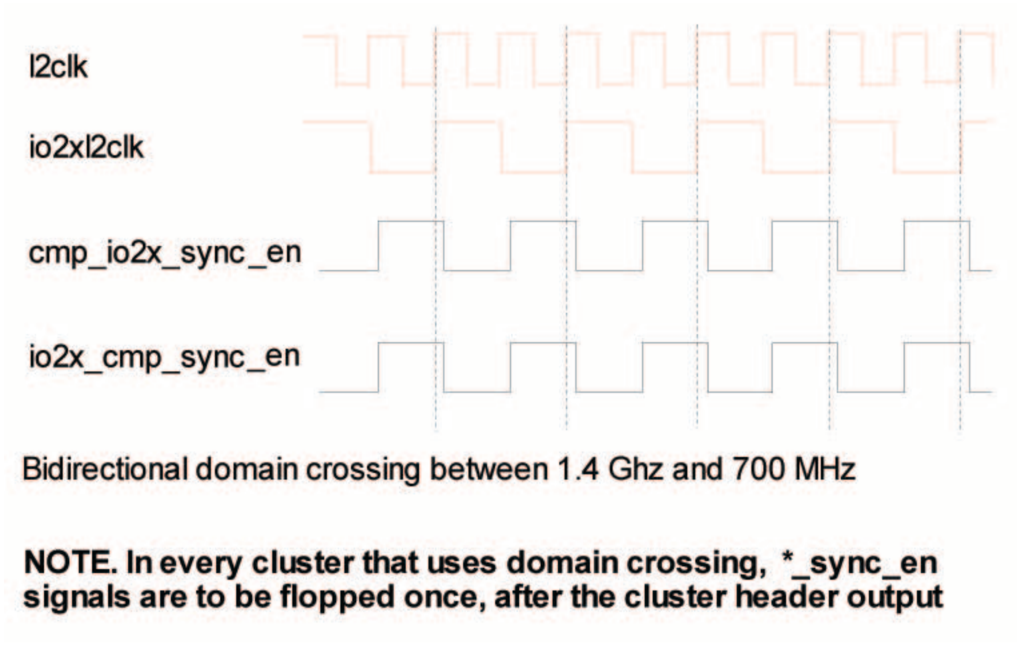
Each I/O cell in MIO that is bi-directional or output only will contain two flops both clocked by the cmp_clk generated by MIO's cluster header(s): one to latch the debug port signal on the io2x sync enable, the other to latch the ramtest signal on the io2x sync enable. Since the ramtest pins are shared 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. With respect to 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/Os. 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 2-1 IO2X Sync Enable Timing with respect to l2clk


l2clk
io2xl2clk
cmp_io2x_sync_en
io2x_cmp_sync_en
Bidirectional domain crossing between 1.4 Ghz and 700 MHz
NOTE. In every cluster that uses domain crossing, *_sync_en signals are to be flopped once, after the cluster header output

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/Os 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, one per cmp cluster header.

The FIGURE 2-1 shows io2x sync enables w.r.t l2clk (cmp_clk).

FIGURE 2-2 shows the global clocking and sync enable distribution (from CCU) to DB1/TCU and MIO blocks.

FIGURE 2-2 Global Clock and Sync Enable Distribution to DB1/TCU and MIO

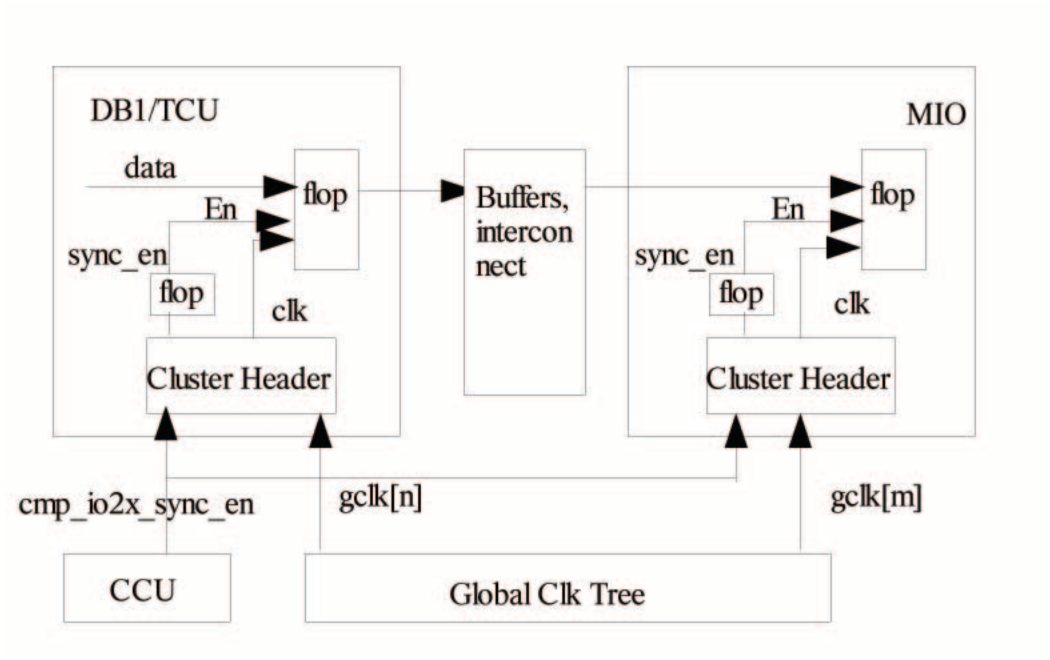
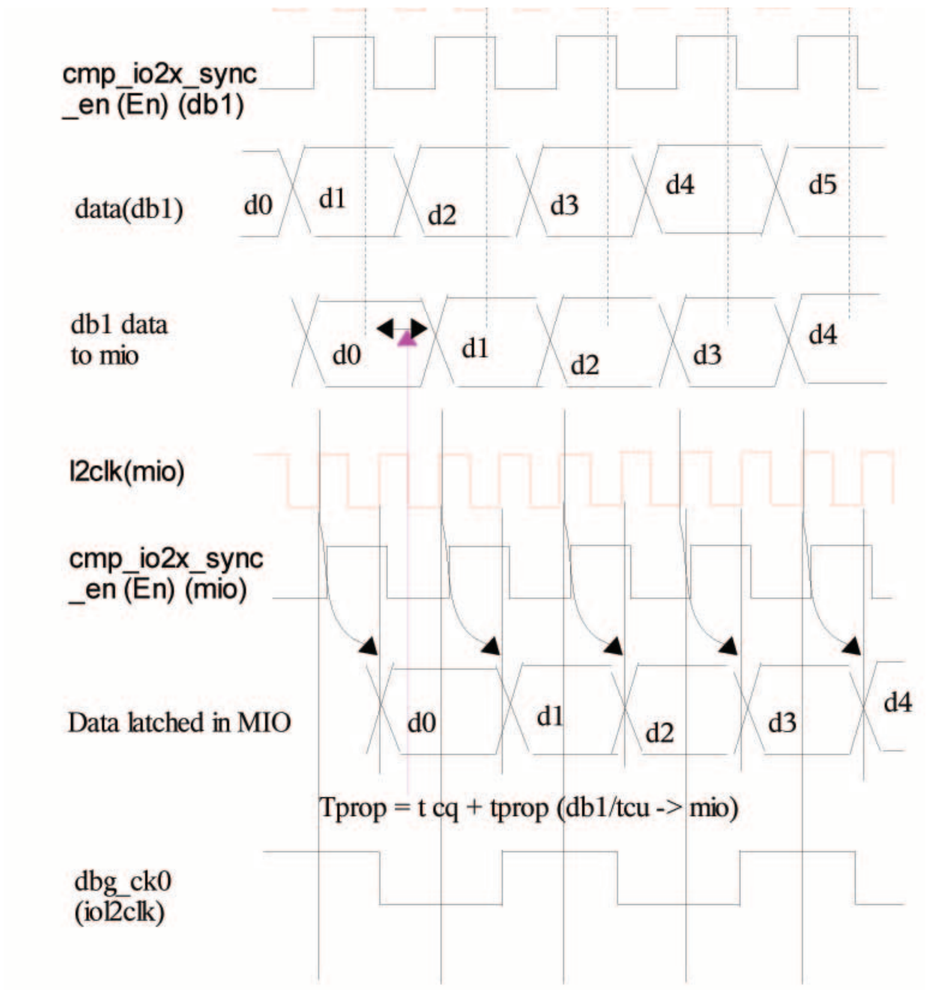


FIGURE 2-3 shows the scheme of launch of data from DB1/TCU off of cmp_clk with io2x sync enable and capture of same data in MIO I/O cell on cmp_clk with io2x sync_en.

FIGURE 2-3 Data Transfer from DB1 to MIO



Note – The membist data transfer mechanism from TCU to MIO is identical. 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 logic analyzer will be calibrated to account for this skew. Please refer to Timing Spec for Debug Port Signals for Reliable Logic Analyzer Sampling for description of the training sequence. This de-skewing in the logic analyzer has to be done only once and should hold valid across PVT variations as the propagation 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.

2.1.4 DFT Support for MIO

MIO implements the following DFT support for its I/Os:

Boundary Scan: All I/Os in 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

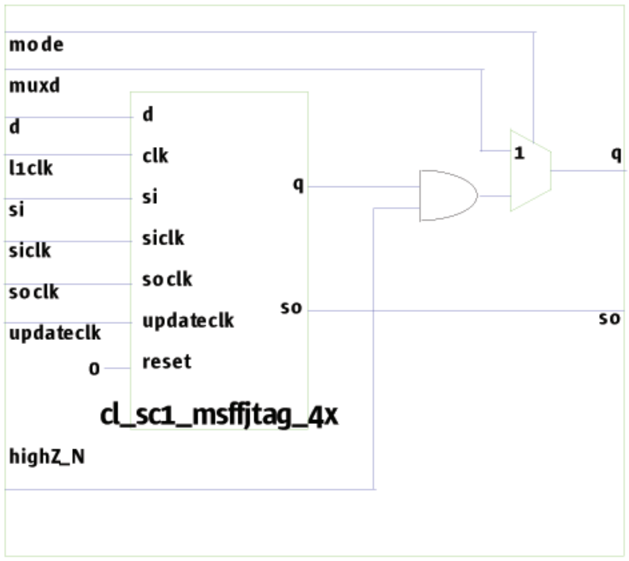
tcu_mio_bs_mode_ctl

All output only and bi-di I/Os of MIO that would be on Bscan chain would have Bscan cell on data out and output enable paths (as all output only I/Os in MIO would have tri-state control). Input only I/Os and bi-di I/Os 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.

FIGURE 2-4 shows the schematic for the cl_sc1_bs_cell2_4x cell which is the Bscan cell being used in MIO.

FIGURE 2-4 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 normal boundary 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 effects since TESTMODE==0, and PLL should block its shared pins with PLL_TESTMODE ==0.

Manufacturing/Automatic Test Pattern Generator (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`

2.2 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, the port would get used to monitor various other signals in OpenSPARC T2 in five 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 five 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 logic analyzer for sampling and aligning the data. In essence this is equivalent to data being driven on both edges of iol2clk. Commercially available logic analyzer's (like Tektronix) do have the ability to support DDR signal sampling with the Tektronix logic analyzer 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 associated with traditional memory multi-drop DDR2 interfaces. Also the Tektronix logic analyzer 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/Os. 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 FIGURE 2-5.

The I/Os in OpenSPARC T2 debug port can be thus broadly classified as falling under five categories:

1. 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/Os would get generated as:

   ```
   assign dbg_mio_drv_en_muxbist_op = debug_en  |
   tcu_dbg_jtag_membist_mode;
   ```

2. 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/Os would get generated as follows:

   ```
   assign dbg_mio_drv_en_muxtest_op = debug_en | mio_dbg_testmode;
   ```

3. I/Os 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/Os would get generated as follows:

   ```
   assign dbg_mio_drv_en_muxtest_inp = debug_en & ~mio_dbg_testmode;
   ```

4. I/Os 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/Os would get generated as follows:

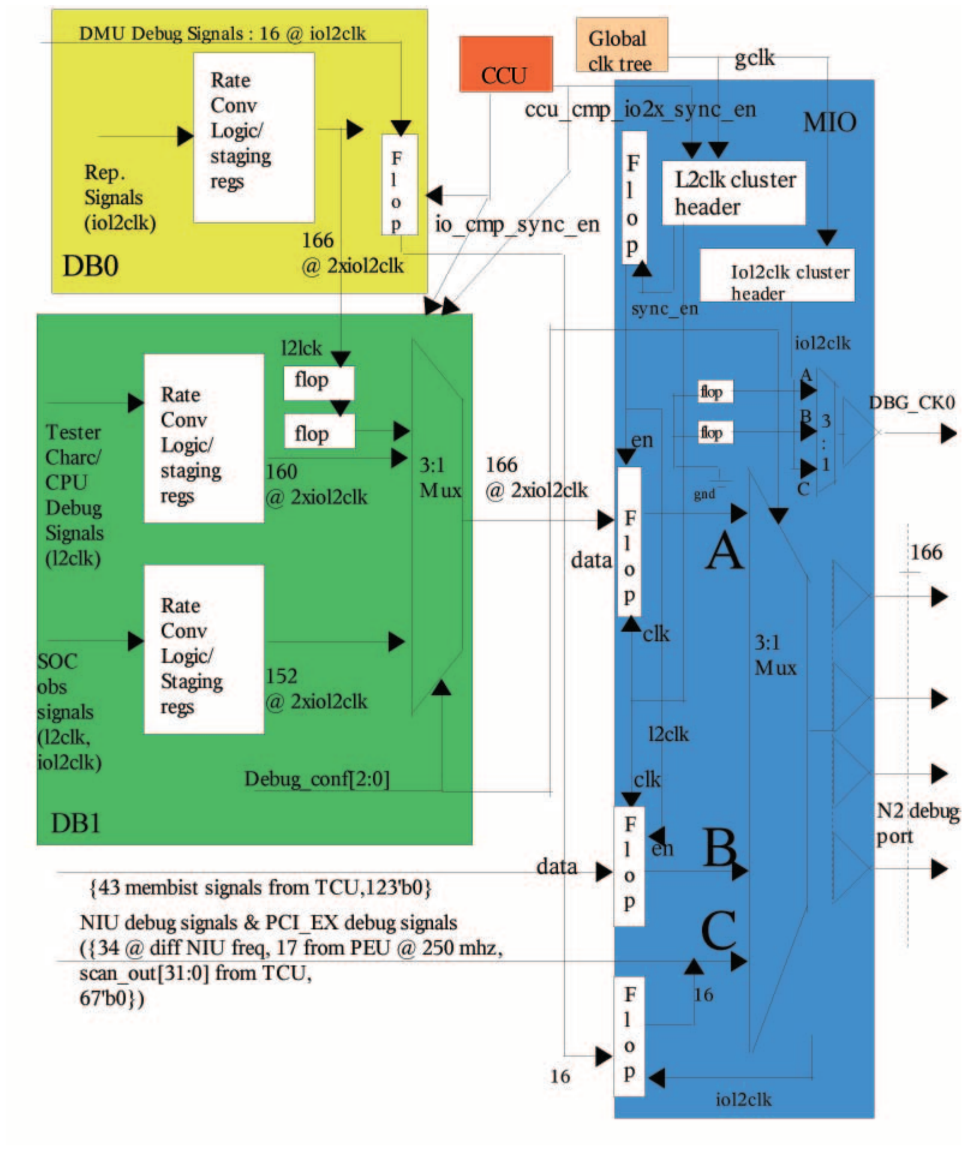
   ```
   assign dbg_mio_drv_en_muxtestpll_inp = debug_en &
   ~mio_pll_testmode;
   ```

5. I/Os which are always driven as outputs in the debug mode. For this group of signals, the Drive_en to the I/Os 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 2-5 OpenSPARC T2 Debug Port Layout across DBG0,DBG1 and MIO



DMU Debug Signals : 16 @ iol2clk
Rate Conv Logic/ staging regs
Rep. Signals (iol2clk)
Flop
166 @ 2xiol2clk
DB0
CCU
Global clk tree
gclk
ccu_cmp_io2x_sync_en
io_cmp_sync_en
MIO
Flop
L2clk cluster header
Iol2clk cluster header
sync_en
iol2clk
l2lck
flop
flop
Tester Charc/ CPU Debug Signals (l2clk)
Rate Conv Logic/ staging regs
160 @ 2xiol2clk
3:1 Mux
166 @ 2xiol2clk
SOC obs signals (l2clk, iol2clk)
Rate Conv Logic/ Staging regs
152 @ 2xiol2clk
Debug_conf[2:0]
DB1
en
data
Flop
clk
A
gnd
flop
flop
A
B
C
3:1
DBG_CK0
166
3:1 Mux
l2clk
clk
Flop
en
B
data
N2 debug port
{43 membist signals from TCU,123'b0}
NIU debug signals & PCI_EX debug signals ({34 @ diff NIU freq, 17 from PEU @ 250 mhz, scan_out[31:0] from TCU, 67'b0})
C
16
Flop
16
iol2clk

2.2.1 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.

2.2.2 Timing Spec for Debug Port Signals for Reliable Logic Analyzer Sampling

For the Tektronix P6860 logic analyzer, the 166 pin debug port of OpenSPARC T2 would be connected to (166/16) = 11 pods (where each pod has 32 data connections and two clock connections). With the data being driven @ 700 mhz on both edges of a 350 mhz clock, the logic analyzer 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 logic analyzer 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 bit basis based on a training sequence being sent out on the debug port. The training sequence would be a repetitive pattern of three one's, followed by 1 zero: this asymmetrical pattern would ease the alignment and de-skewing of the data bits in the logic analyzer in case the skew for some bits is as large as one cycle.

Note that once a calibration is done, the maximum cycle to cycle PVT skew that the logic analyzer 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 maintain 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.

2.3 MIO RTL Hierarchy

The MIO block (mio.v) would consist of the following design sub-blocks:

1. 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.

2. 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/Os 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/Os.

3. 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.

4. 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 inverters 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 five flavors of I/O cells are as follows:

1. Output Only (No Bscan). Module name: n2_mio_cell_out

   Pins driven: TDO,IMP_MON_PU,IMP_MON_PD

2. Input Only (No Bscan). Module name: n2_mio_cell_in

   Pins driven: TCK,PGRM_EN,BURNIN

3. 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]

4. 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]

5. Bidi (with Bscan). Module name: n2_mio_cell_bi_bscan

   Pins Driven: DBG_DQ[139:103],DBG_DQ[84:72],XAUI_MDIO

6. Input Only (No Bscan) with Pullup. Module Name: n2_mio_cell_in_pu

   Pins Driven: TDI,TMS,TRST_L

7. 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

8. 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])

9. 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 3

Debug

This chapter contains the following sections:

3.1 Overview

This document describes OpenSPARC T2 hardware (HW) features for post silicon debug ability which involves debugging any issues that interfere with early bring-up 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.

3.2 OpenSPARC T2 Debug Features

3.2.1 Observability

3.2.1.1 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 PLLs 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 PLLs.

TABLE 3-1 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 3-2 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

3.2.1.2 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) Configuration Register (CREG) access. The debug port can be configured into any one of five observability modes based on CSR bits (Dbg_conf[2:0]bits in OpenSPARC T2 Debug Port Configuration register: 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 four clocks. Eight clocks minimum before next set of four 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 five 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 three 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 four signals driven to debug.v and each L2 bank will have six signals driven to debug.v. All these signals will be at CMP clk frequency i.e. 1.4 GHz nominal. Since there are eight cores and eight 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 two 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 four wires are chosen as follows:

There are two pipes/core and two thread groups per core. Since each core has two thread grps, there can be two 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. it is possible to observe every instruction committed per cycle in each thread group. it is not necessary to know which thread each instruction belongs.

For each L2 bank, the six 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:

Five 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 five 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 three writes on any cycle on two 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 four MCUs, this will lead to a total of 84 wires to debug.v from all MCUs 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 two 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.

3.2.2 Repeatability

In order to effectively run processor tests in the post-silicon phase with or without the presence of I/O and debug them, it is necessary 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, and 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 push-out 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 Debug Appendix.

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 architecture unchanged for warm reset. Also contents of arrays (TLBs, L1/L2 caches, etc.) are unchanged. Please refer to Debug Appendix 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 microsecs).

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 repeatability, 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 common 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, and 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. 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 with respect to 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.

3.2.2.1 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 FIFOs and two synchronizers per FIFO (this way during debug/warm reset, the read and the write pointers constantly increment and are always offset by two: delay through the two 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 two new CSR bits for SW to control this feature. These two CSR bits are located in DRAM Debug Trigger Enable register. The two 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.

   (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:

   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 LFSRs 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 LFSRs 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.

3.2.2.2 I/O Quiescent in OpenSPARC T2 During Checkpoint

An inherent requirement for checkpoint/replay in OpenSPARC T2 is to stall I/O to the synchronous 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 OpenSPARC T2 I/O Quiesce Control Regs. 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 – 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.

3.2.3 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 communicate with the external Logic Analyzer to start or stop taking LA traces on the debug port (in any one of the four 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 repeatable by running the same code sequence) or occurrence of error events in different blocks in OpenSPARC T2.

3.2.3.1 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 four 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 (recorded 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.

3.2.3.2 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 eight 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 it is necessary to keep memory refresh running for DRAM and the PCI_EX and XAUI SERDES links running: Certain logic sections in MCU,PEU and NIU/MAC cannot be kept 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, and DMU, 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.

3.2.4 Joint Test Action Group (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 CPUs. 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 (address TBD) 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 two bit DECR in TCU (address TBD) 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 four 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 Clock Stop.

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 SPARC debug. More details are in OpenSPARC T2 Core Debug Features.

3.2.4.1 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.

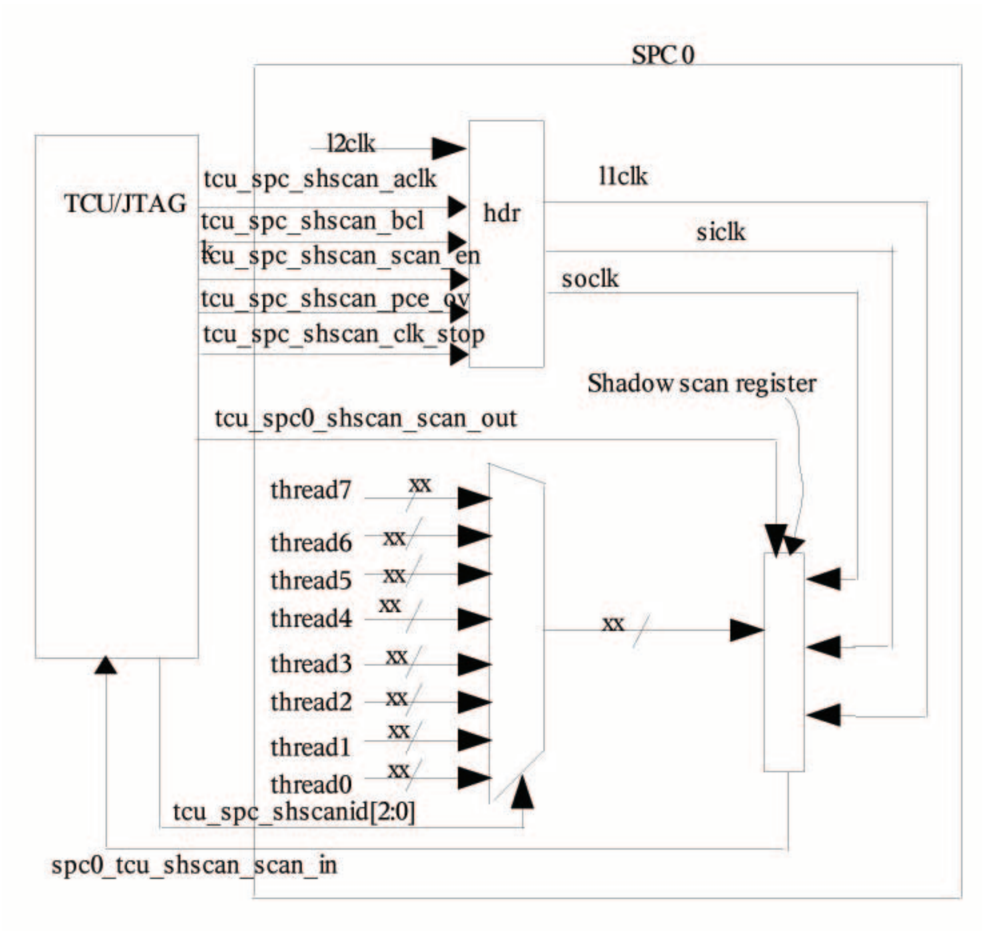
In-system scan is done through JTAG instructions. The scan chains are configured into a single long chain and placed between TDI and TDO. The chains used to construct the long chain will be configurable.

3.2.4.2 JTAG Shadow Scan

Shadow scan for the cores and L2 will be controlled via JTAG. The core shadow scan architecture is shown in FIGURE 3-1. 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 3-1 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 two-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 two cycles required for synchronization will not cause any overlapping.

All eight 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

3.2.4.3 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.

3.2.4.4 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 CSRs 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 one 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 with respect to 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 PIOs 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.

3.2.4.5 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 two 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 four encodings: Do Nothing, Hard Stop, Pulse Trigout, and Clock Stretch.

3.2.4.6 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 two 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 second 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. On OpenSPARC T2 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 two different stops are two 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 3-3. 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 3-2.

FIGURE 3-2 TCU Clock Stop Logic

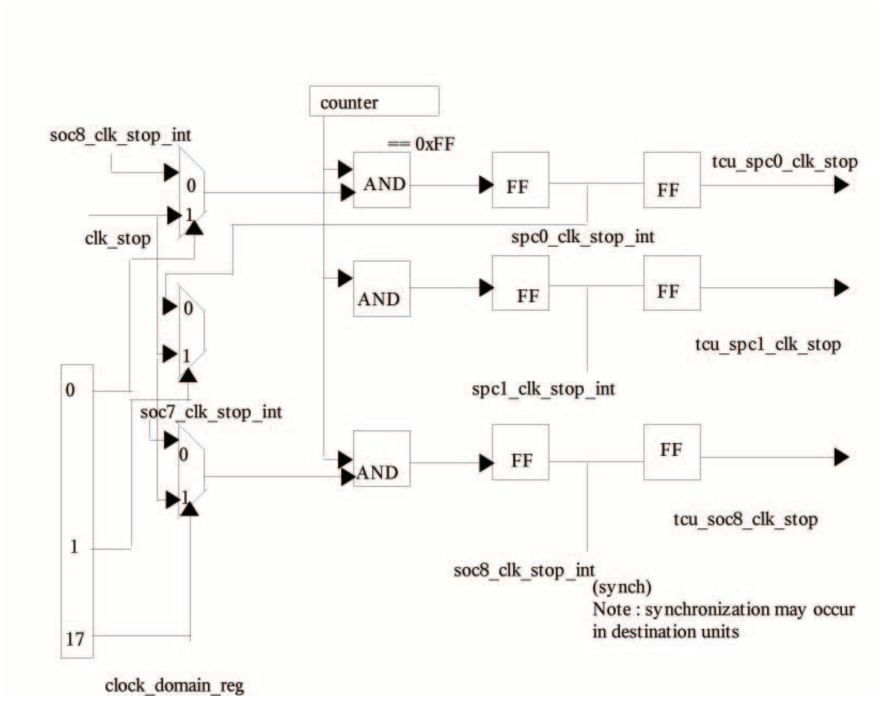
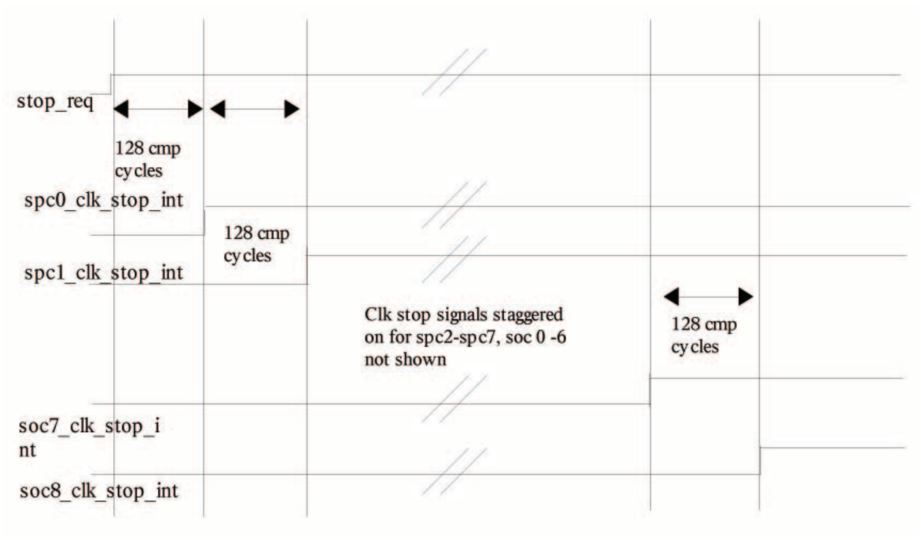


FIGURE 3-3 Clock Stop Sequencing through Clock Domains



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.

3.2.4.7 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 OpenSPARC T2 Core Debug Features.

3.2.5 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, and 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).

3.2.6 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.

3.2.7 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 3-3 OpenSPARC T2 DTM Modes

Serdes_DTM 1	Serdes_DT M2	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: Don't 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 six 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 3-4 and FIGURE 3-5 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 3-4 DTM Mode 1 Configuration for db1,mio in OpenSPARC T2

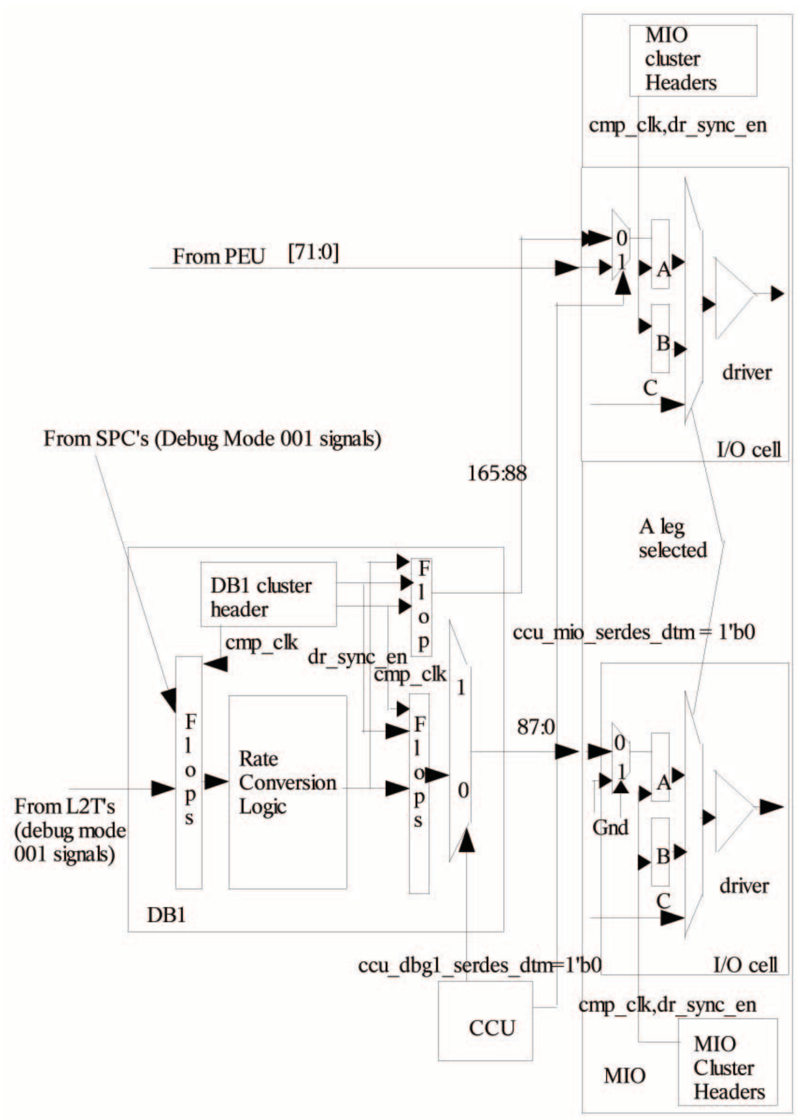
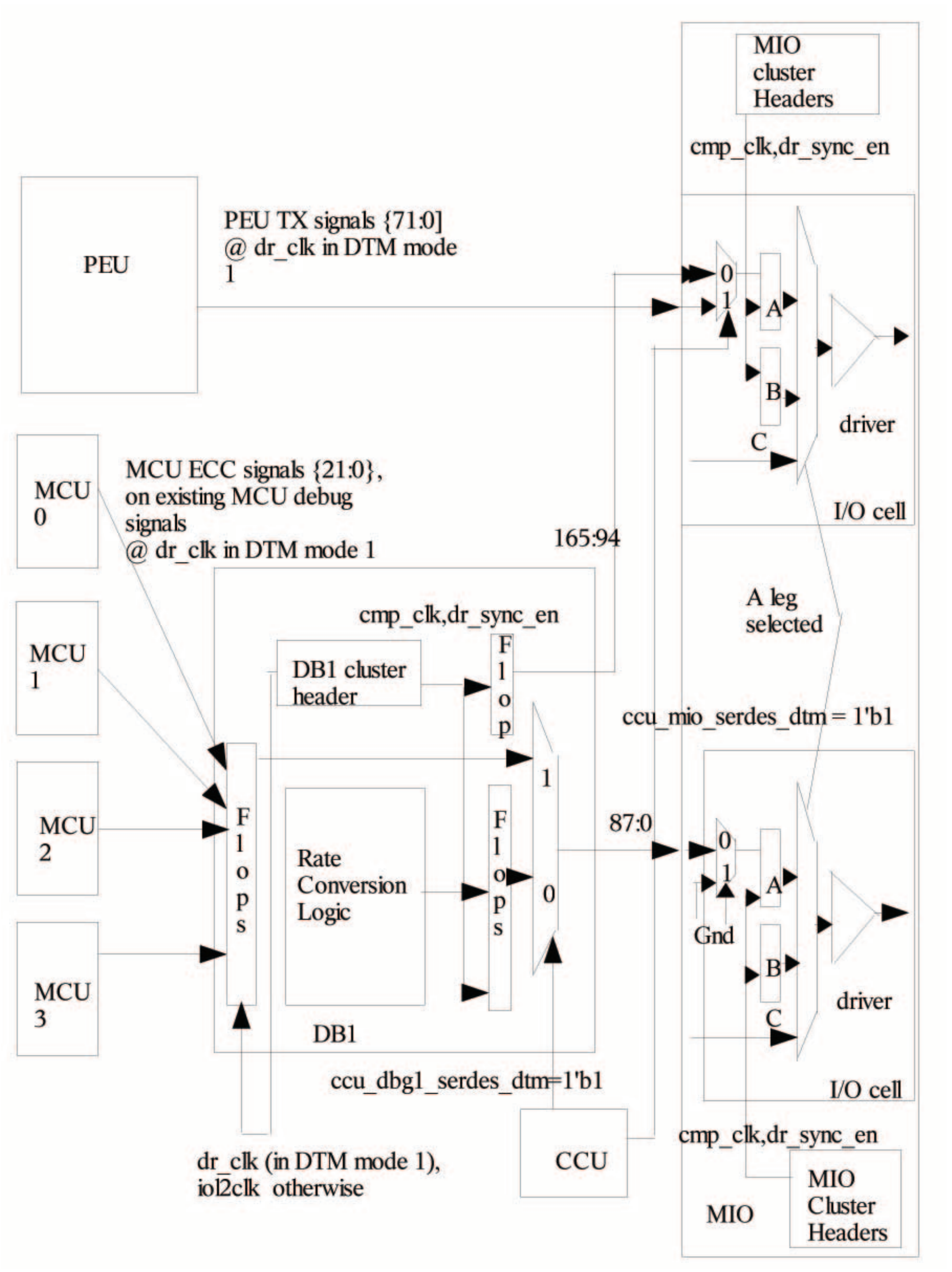


FIGURE 3-5 DTM Mode 2 Configuration for db1,mio in OpenSPARC T2



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 with respect to cmp_clk and also the mechanism of OpenSPARC T2 entering DTM modes would all be covered in the CCU MAS.

3.2.7.1 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 two southbound FBD channels, and each channel has 22 bits of CRC per frame. The 22 bits from each channel are bit wise XORed to provide 22 bits total to the debug port.

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]}
```

3.3 OpenSPARC T2 Core Debug Features

From a system and business perspective, the features goal is to minimize time-to-revenue by providing means to speed silicon and system bring-up 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 three 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 (three 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 three 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.

3.3.1 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. Existing architecturally visible debug capabilities. Existing means specified by V9/JPS1 or other SPARC processors, such as Millennium 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) or pulses an external pin.

   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.

3.3.2 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.

With respect to 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 TBD 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 one of eight cores, all other debug events are disabled in core and SoC DECRs. 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 cannot be used for a 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 determine the critical path.

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 Joint Test Action Group (JTAG) Access.

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 pipe-lining 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 previously. 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:

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 TBD register indicating Cycle Stepping done and stops the clocks to the cycle stepped cores.
8. User reads this TBD 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 – 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.

3.3.3 Details of the OpenSPARC T2 Core Debug Features

This section details the OpenSPARC T2 core debug features.

3.3.3.1 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 inTABLE 3-5. All bits are initialized to 0 at POR. Reserved bits read as zeroes and are ignored on writes.

TABLE 3-4 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.

3.3.3.2 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 3-6i.

TABLE 3-5 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 3-7. 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 3-6 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 Programmer's Reference Manual*. 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.

3.3.3.3 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.

3.3.3.4 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 Core Interface with the TCU.

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 Debug Appendix. 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.

3.3.3.5 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 pipe-lining 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.

3.3.3.6 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 SPU 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 – 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.

3.3.3.7 Shadow Scan

OpenSPARC T2 core shadow scan provides access to the PC, HPSTATE, PSTATE, TL, TT, TPC, and 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.

3.3.3.8 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 Debug Appendix.

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 SPU 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.

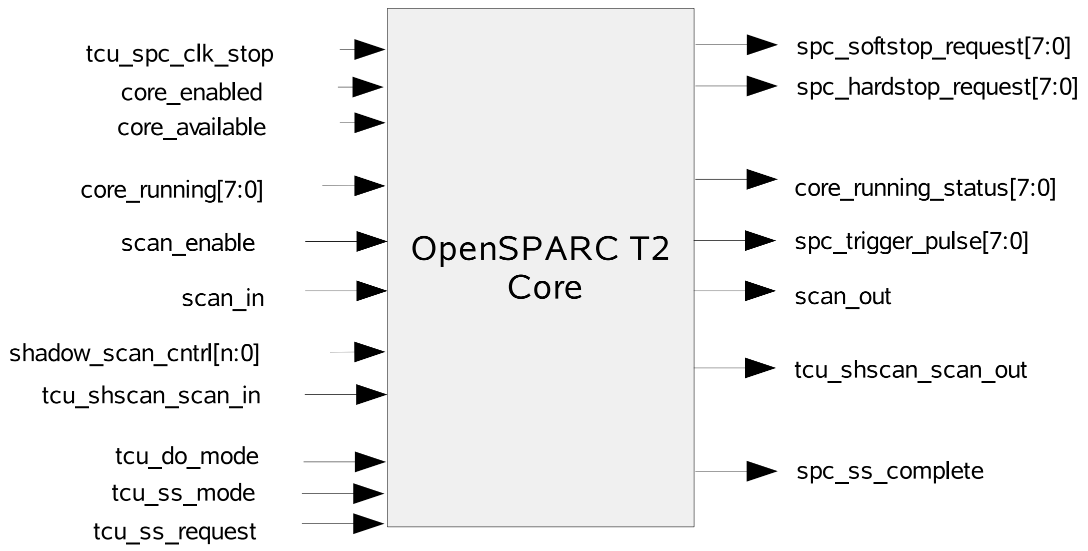
3.4 Core Interface with the TCU

This section outlines the interface between the OpenSPARC T2 Core and the TCU for the purposes of describing debug functions. FIGURE 3-6 shows a high-level diagram of the relevant interface signals (per core).

3.4.1 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 3-6 OpenSPARC T2 Core to TCU Debug Interface



3.4.1.1 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.

3.4.1.2 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.

3.4.1.3 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.

3.4.1.4 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.

3.4.1.5 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, SPU is idle) 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. The OpenSPARC T2 Core will not quiesce until the SPU has completed any pending operations.

3.4.2 Debug Event Interface

This group of core outputs are used to signal either an error or that a debug trigger event has occurred.

3.4.2.1 spc_trigger_pulse[7:0]

This is a bus from the core to TCU covering the eight 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.

3.4.3 Scan Interface

Not all signals relevant to the scan interface are detailed here (e.g., not all the scan clocks and controls are listed).

3.4.3.1 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".

3.4.3.2 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.

3.4.3.3 Shadow_scan_in

This is the scan-in for the shadow-scan string.

3.4.3.4 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 eight 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

3.4.3.5 Shadow_scan_out

This is the scan-out of the core's shadow-scan string.

3.4.4 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:

1. **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.**
2. **User issues a JTAG command to do a Single Step (TAP_single_step)**
3. **TCU parks all threads to the core(s) enabled for single step by deasserting core_running[7:0] to the core(s).**
4. **All threads indicate they are parked via core_running_status[7:0] to TCU.**

5. **TCU asserts tcu_ss_mode to the core(s).**
6. **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.**
7. **TCU pulses tcu_ss_request for one CMP clk.**
8. **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.**
9. **TCU sets a bit in a TBD register visible to JTAG indicating Single Step done.**
10. **User reads this register through JTAG to know that Single Step done.**
11. **For a sequence of N single steps, execute steps 2:10 N-1 times**
12. **User issues a TAP_CLOCK_HSTOP to hard stop the core(s) that were single stepped.**

Note – Hard stop can be used because the cores that were being single stepped have all threads parked/idle, so can be hard stopped.

---

13. **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).**
14. **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.**
15. **User writes to ASI_overlap_register to put the core(s) back to normal mode of operation.**
16. **User reads the ASI_Overlap_register to know that core(s) have been put back to normal mode.**
17. **User issues a TAP_STOP_SINGLE_STEP command to TCU to get the core(s) out of single step mode.**
18. **TCU deasserts core_running[7:0] to the core(s).**
19. **Core(s) indicate all threads parked on core_running_status[7:0] to TCU.**
20. **TCU deasserts tcu_ss_mode to the core(s) taking the core(s) out of single step mode.**

21. **TCU unparks enabled threads by asserting respective bits in core_running[7:0] bus.**

22. **Core resumes execution on all enabled threads in normal mode.**

Note – Data integrity may be lost unless all cores are run in single step in unison.

3.4.5 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:

1. **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.**

2. **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.**

3. **User issues a JTAG command to do a Disable Overlap (TAP_disable_overlap).**

4. **TCU parks all threads to the core(s) enabled for disable overlap by deasserting core_running[7:0] to the core(s).**

5. **All threads indicate they are parked via core_running_status[7:0] to TCU.**

6. **TCU asserts tcu_do_mode to the core(s).**

7. **TCU asserts core_running[7:0] to the core(s), unparking all the enabled threads in the core(s).**

8. **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.**

9. **TCU counter decrements to zero indicating end of disable overlap.**

10. **TCU parks all threads by deasserting core_running[7:0] to the core(s).**

11. **Core(s) indicate all threads parked on core_running_status[7:0] to TCU.**
12. **TCU sets a bit in a TBD register visible to JTAG indicating Disable Overlap done.**
13. **User reads this register to know that Disable Overlap done.**
14. **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.**
15. **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).**
16. **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.**
17. **TCU turns on the clocks to the core(s) that were hard stopped.**
18. **User writes to ASI_overlap_register to put the core(s) back to normal mode of operation.**
19. **User reads the ASI_Overlap_register to know that core(s) have been put back to normal mode.**
20. **User issues a TAP_STOP_DISABLE_OVERLAP command to TCU to get the core(s) out of disable overlap mode.**
21. **TCU deasserts tcu_do_mode to the core(s) taking the core(s) out of disable overlap mode.**
22. **TCU unparks enabled threads by asserting respective bits in core_running[7:0] bus.**
23. **Core resumes execution on all enabled threads in normal mode.**

Note – Data integrity may be lost unless all cores are run in single step in unison.

3.5 Debug Block Interface Signals

TABLE 3-7 Debug Block Interface Signals

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
PEU					
peu_mio_debug_bus_a[7:0]	I	8	PEU	pcl2clk	Debug Bus A from PEU to MIO

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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 up to two 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					
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)

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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
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.

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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)
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

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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)
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)

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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
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

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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
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

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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
mio_tcu_io_bclk	O	1	TCU	100 – 200 MHz (tester)	B clock during manufacturing scan

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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 from 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	Independent 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
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

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
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 Committed in Thread Group 0 for SPC 0
spc0_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC0	L2clk	Instruction Committed in Thread Group 1 for SPC 0
spc1_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC1	L2clk	Instruction Committed in Thread Group 0 for SPC 1
spc1_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC1	L2clk	Instruction Committed in Thread Group 1 for SPC 1
spc2_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC2	L2clk	Instruction Committed in Thread Group 0 for SPC 2
spc2_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC2	L2clk	Instruction Committed in Thread Group 1 for SPC 2
spc3_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC3	L2clk	Instruction Committed in Thread Group 0 for SPC 3
spc3_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC3	L2clk	Instruction Committed in Thread Group 1 for SPC 3
spc4_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC4	L2clk	Instruction Committed in Thread Group 0 for SPC 4
spc4_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC4	L2clk	Instruction Committed in Thread Group 1 for SPC 4
spc5_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC5	L2clk	Instruction Committed in Thread Group 0 for SPC 5
spc5_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC5	L2clk	Instruction Committed in Thread Group 1 for SPC 5
spc6_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC6	L2clk	Instruction Committed in Thread Group 0 for SPC 6
spc6_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC6	L2clk	Instruction Committed in Thread Group 1 for SPC 6

TABLE 3-7 Debug Block Interface Signals *(Continued)*

Signal Name	I/O	Size	From/ To	Clk Dmn	Description
spc7_dbg1_instr_cmt_ grp0[1:0]	I	2	SPC7	L2clk	Instruction Committed in Thread Group 0 for SPC 7
spc7_dbg1_instr_cmt_ grp1[1:0]	I	2	SPC7	L2clk	Instruction Committed 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

3.6 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.

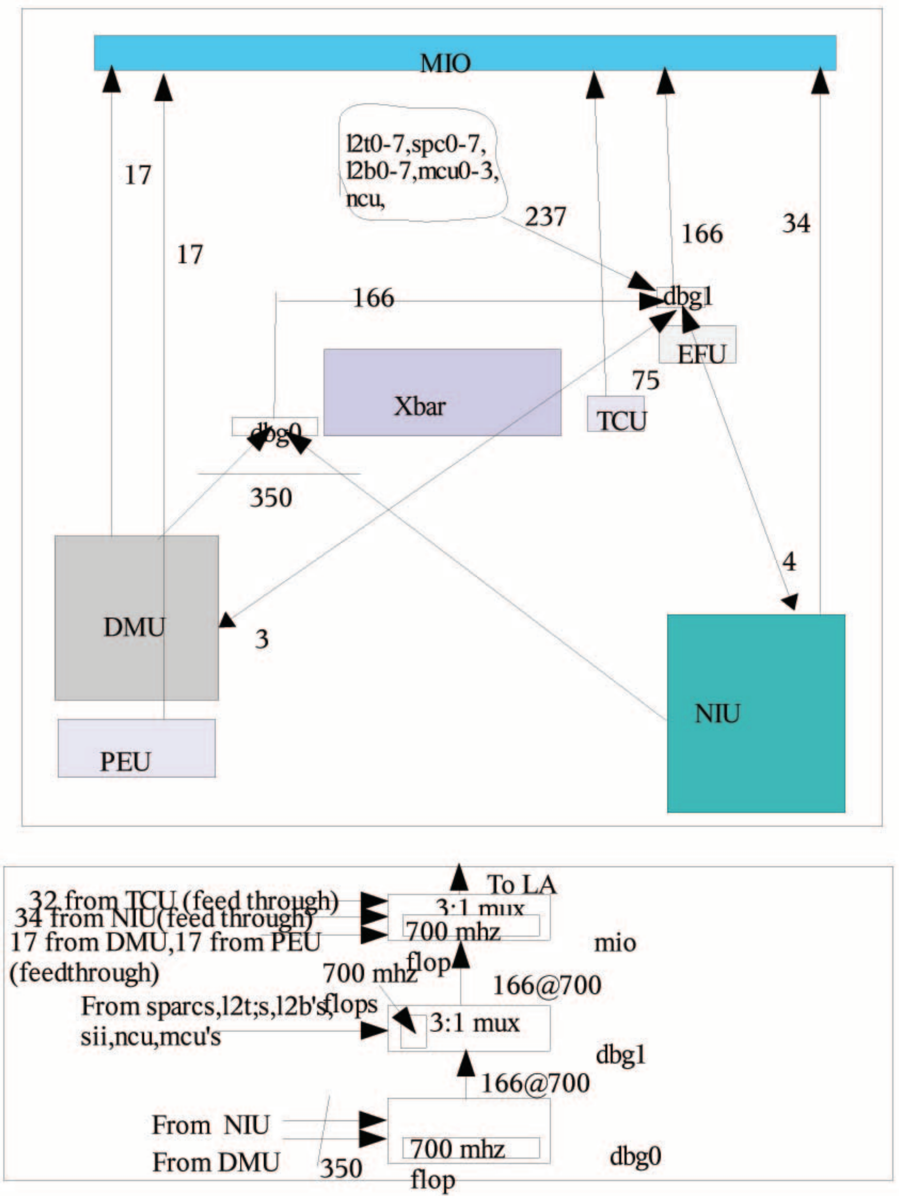
Hosts I/O mapped SoC DECR register to assert Hard Stop or pulse TRIGOUT request to TCU based on various SoC debug events.

Hosts I/O mapped CSR to configure debug port in any one of five 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 Jobs 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 3-7 DBG0 and DBG1 in OpenSPARC T2 Floorplan



3.6.1 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 (Repeatability), the port will get used to monitor various other signals in OpenSPARC T2 in four 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 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/Os. 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 FIGURE 3-8.

The I/Os in OpenSPARC T2 debug port can be thus broadly classified as falling under five 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/Os 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/Os will get generated as follows:

```
assign dbg_mio_drv_en_muxtest_op = debug_en | mio_dbg_testmode;
```

I/Os 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/Os will get generated as follows:

assign dbg_mio_drv_en_muxtest_inp = debug_en & ~mio_dbg_testmode;

I/Os which are always driven as outputs in the debug mode. For this group of signals, the Drive_en to the I/Os 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.

FIGURE 3-8 OpenSPARC T2 Debug Port layout across DBG0,DBG1 and MIO

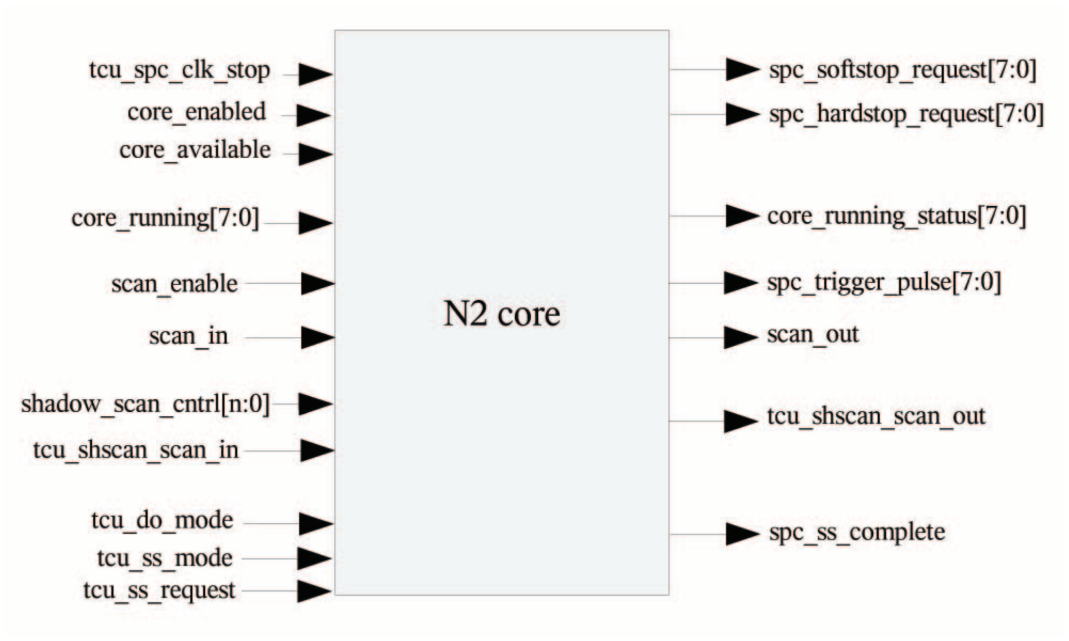


FIGURE 3-9 Rate Conversion from iol2clk to io2xclk

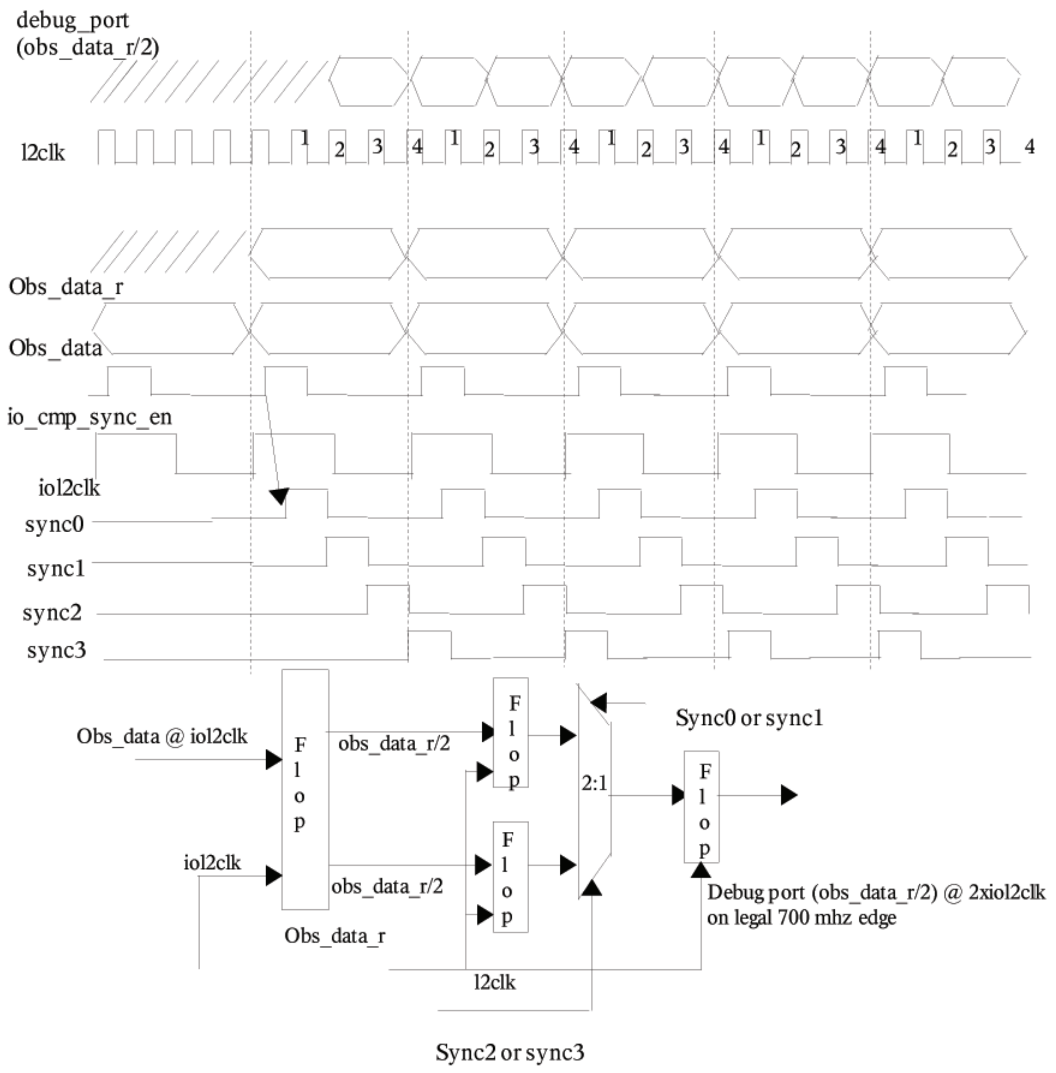


FIGURE 3-10 Rate Conversion from l2clk to io2xclk

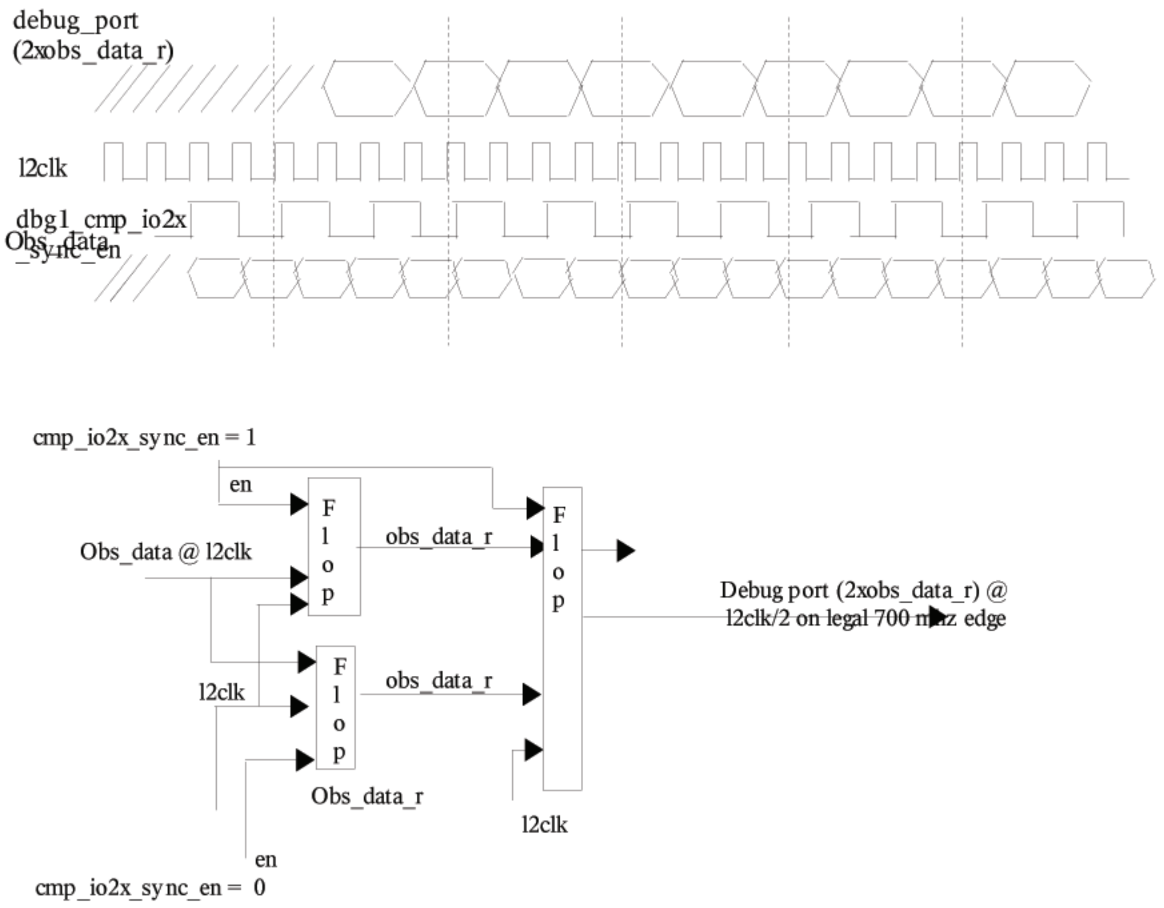


TABLE 3-8 Mapping

Name	Field	Description
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 3-8 Mapping *(Continued)*

Name	Field	Description
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 3-8 Mapping *(Continued)*

Name	Field	Description
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]}

3.6.2 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 Debug Appendix. This module will be operating at iol2clk frequency and will communicate with NCU, TCU, DMU and NIU.

It will have a four 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).

With respect to 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.

With respect to NIU and DMU it will support separate interfaces to control I/O quiescing of NIU and DMU individually to complement checkpoint replay.

3.7 Debug Appendix

3.7.1 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 (full list TBD)

** 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

3.7.2 SW Visible State Lost on Debug Reset

TABLE 3-10 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 3-9 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)
TT[TL	TT[TL	1	1

TABLE 3-9 State that Loses Value over debug_reset (excluding NIU and PCI_EX) *(Continued)*

Name	Fields	POR	WMR/DBR
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_CWQ_HEAD		0	0
ASI_CWQ_TAIL		0	0
ASI_CWQ_FIRST		0	0
ASI_CWQ_CSR		0	0
ASI_SPU_MA_CTL		0	0
ASI_SPU_MA_PA		0	0
ASI_SPU_MA_NP		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

TABLE 3-9 State that Loses Value over debug_reset (excluding NIU and PCI_EX) *(Continued)*

Name	Fields	POR	WMR/DBR
ASI_SPARC_PWR_MGMT		0	0
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
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

3.7.3 Registers to Support Debug

3.7.3.1 Debug Port Configuration Register

This register is used to enable and configure the debug port in any one of six modes. It is located in debug.v module at location 0x86-0000-0000. The format of this register is shown in:TABLE 3-11

TABLE 3-10 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
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 : SoCSoC 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

3.7.3.2 RESET_GEN Register

The reset generation register, TABLE 3-12, is provided to allow software to generate XIR resets to all processors specified in the ASI_XIR_STEERING register or a chip wide warm or debug reset.

TABLE 3-11 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.

3.7.3.3 RESET_SOURCE Register

The reset source register, TABLE 3-13, allows software to identify the source of a reset. The bits in this register are write-one to clear.

TABLE 3-12 Reset Source Register RESET_SOURCE (0x89-0000-0818)

Field	Bits	Reset Name	Reset Value	Type
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

TABLE 3-12 Reset Source Register RESET_SOURCE (0x89-0000-0818)

Field	Bits	Reset Name	Reset Value	Type
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.

3.7.3.4 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 shown in TABLE 3-14.:

TABLE 3-13 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 0xFFFFFFFFF0000020. Value preserved across warm reset and debug reset.

3.7.3.5 MCU Channel Read Latency Register

This register is at location 0x84_0000_08B8. The format is shown in TABLE 3-15.

TABLE 3-14 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.

3.7.3.6 MCU Sync Frame Frequency Register

This register is at location 0x84_0000_08B0. The format is shown in TABLE 3-16.

TABLE 3-15 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.

3.7.3.7 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 3-16 Subsystem Reset Register

Field	Bit Position	Initial Value	R/W	Description
RSVD1	63:5	0	RO	Reserved
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.

3.7.3.8 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 3-18:

TABLE 3-17 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.

3.7.3.9 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 TABLE 3-19.

TABLE 3-18 ASI_DECR Format

Field	Bits	Reset Name
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

TABLE 3-18 ASI_DECR Format *(Continued)*

Field	Bits	Reset Name
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 shown in TABLE 3-20.

TABLE 3-19 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

3.7.3.10 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 TABLE 3-21. This register will be physically located in the Debug block (debug.v).

TABLE 3-20 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 shown in TABLE 3-22.

TABLE 3-21 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

3.7.3.11 L2 Mask Register

This register will be located at address 0xAF-0000-0000 within l2t.sv. The format is as shown in TABLE 3-23.

TABLE 3-22 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

3.7.3.12 L2 Compare Register

This register will be located at 0xBF-0000-0000 within l2t.sv. The format is as shown inTABLE 3-24.

TABLE 3-23 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

3.7.3.13 DMU 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 3-24 DMU Core and Block Interrupt Enable register Format

Field	Bit Position	Initial Value	R/W	Description
DMU	63	0x0	R/W	The enable bit to enable all operations from the DMU 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.

3.7.3.14 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 shown in TABLE 3-26.

TABLE 3-25 DRAM Debug Trigger Enable Register

Field	Bit Position	Initial Value	R/W	Description
RSVD	63:3	0x0	R/W	Reserved
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.

3.7.3.15 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 3-26 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

3.7.3.16 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 3-27 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.

3.7.3.17 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 3-28 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 3-28 ASI_OVERLAP_MODE Register *(Continued)*

Field	Bit Position	Initial Value	R/W	Description
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

3.7.3.18 PEU Debug Select A Register

The PEU debug select register selects the output on PEU debug bus A.

TABLE 3-29 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

3.7.3.19 PEU Debug Select B Register

The PEU debug select register selects the output on PEU debug bus B.

TABLE 3-30 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

3.7.3.20 DMU Debug Select Register for DMU Debug Bus A

The DMU debug select register A selects the output on DMU debug bus A.

TABLE 3-31 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 Messenger 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

3.7.3.21 DMU Debug Select Register for DMU Debug Bus B

The DMU debug select register selects the output on DMU debug bus B.

TABLE 3-32 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 Messenger 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 4

Electronic Fuse Unit (EFU)

This chapter contains the following sections:

4.1 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 to TABLE 4-3) 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 three 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.

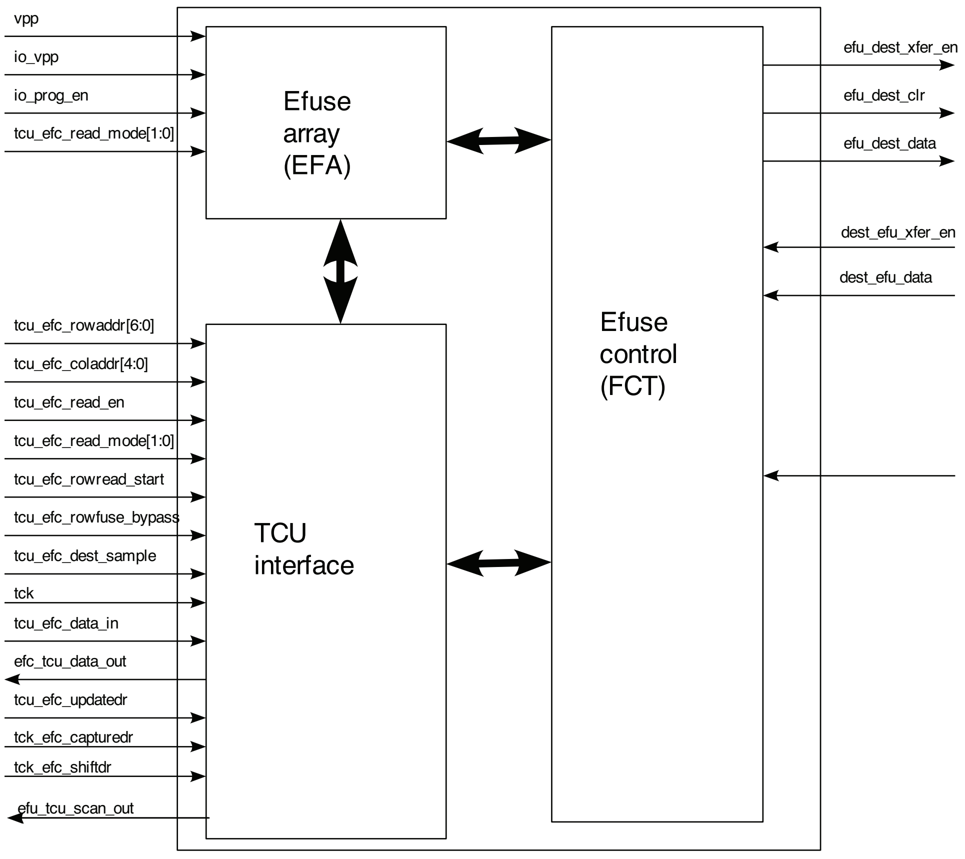
4.1.1 Definitions of Terms Used

TABLE 4-1 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.

4.2 EFU Block Diagram

FIGURE 4-1 EFU Top Level Diagram



4.2.1 Unit Functional Description of EFU

The eFuse unit (EFU) contains:

- eFuse array (EFA)
- eFuse controller (FCT)
- TCU interface

4.2.1.1 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 fields shown in TABLE 4-2.

TABLE 4-2 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.

TABLE 4-3 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

TABLE 4-3 Truth Table of EFA Programmed 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.

4.2.1.2 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 four 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 4-2) 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 eFuse Modes of Operations . Unloading of the shift registers can be to any legal block ID TABLE 4-5 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. 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 determine if 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] 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.

4.2.1.3 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 to Section 4.5, "Miscellaneous/Multiple Clock Domains" on page 4-41 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.

TABLE 4-4 lists all the commands, which are used by TCU to program EFU behavior.

TABLE 4-4 TAP Private Instructions for Fuse Functionality

Field	Bits	Reset Name
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
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

4.3 EFU Logical Implementation

4.3.1 eFuse Modes of Operations

TCU can configure eFuse in five different modes. The following are the various modes of operation of EFU:

4.3.1.1 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 4-2 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 4-2 group B).

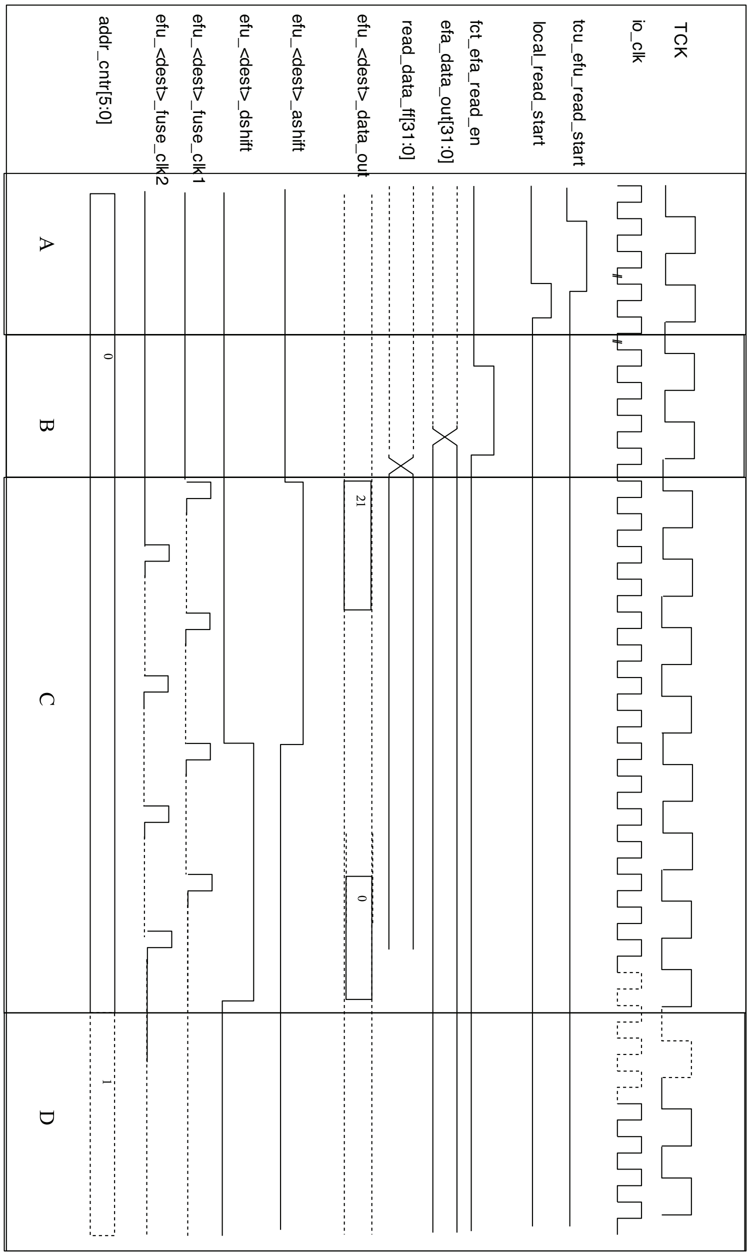
EFU determines if the row is valid and error free (refer to TABLE 4-3). If an error is encountered then the corresponding bit is set in the rslt_status_ff[63:0].

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 4-2 group C)

Upon completion of processing a row, addr_cnt_ff[5:0] is incremented (refer FIGURE 4-2 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 4.3.2.3, "EFU to NCU Interface:" on page 4-19.

FIGURE 4-2 Timing Diagram showing Power On Reset Read Mode



4.3.1.2 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 two 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 4-2 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 4-2 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:

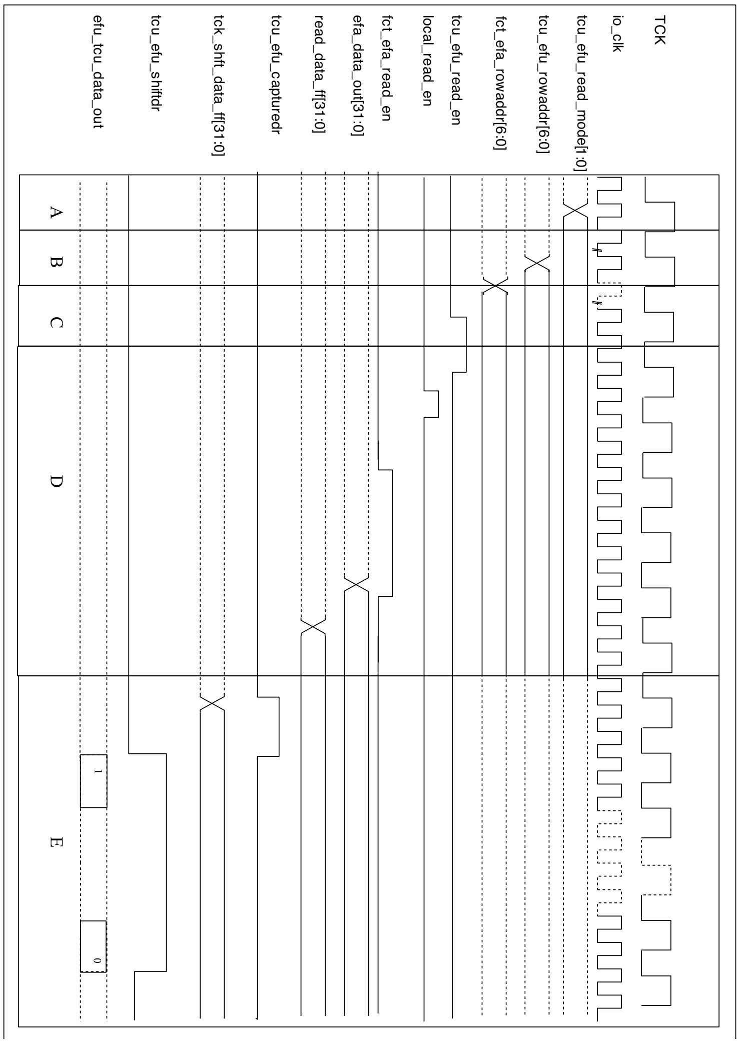
efa_fuseout[31:0] = {29'b0,vddc_ok,vddo_ok,vpp_ok

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 4-3 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 4-3 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 4-3 group E) The readback data shifted out first is bit 31 to tcu, followed by the subsequent lower significant bits.

FIGURE 4-3 JTAG Read Access Timing Diagram.



4.3.1.3 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

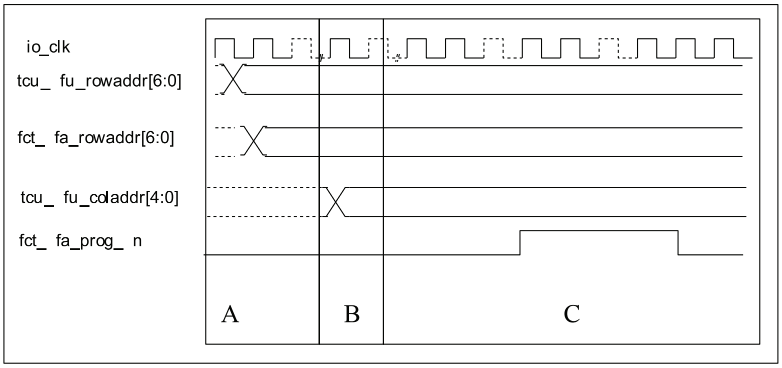
TAP_FUSE_ROW_ADDR instruction gets issued to the TAP controller through the JTAG port. TCU decodes and supplies the row address as efu_tcu_rowaddr[6:0]. The row address is synchronized from TCK to iol2clk domain as fct_efa_row_addr[6:0]. (refer FIGURE 4-5 group A)

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 4-5 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 4-5 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 4-4 Fuse Programming Mode Timing Diagram.



4.3.1.4 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.

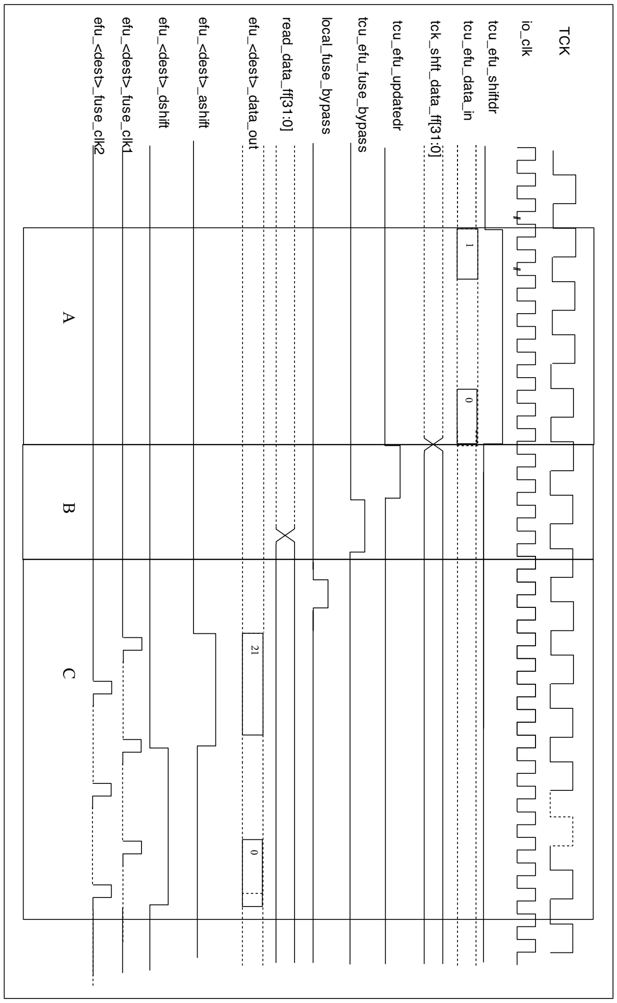
TAP_FUSE_BYPASS instruction gets issued to the TAP controller through the JTAG port. TCU decodes and asserts tcu_efu_fuse_bypass valid for one TCK clock. tcu_efu_fuse_bypass is synchronized into iol2clk domain as local_fuse_bypass.

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 4-4).

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 4-6 group C)

FIGURE 4-5 JTAG Fuse Bypass Mode



4.3.1.5 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

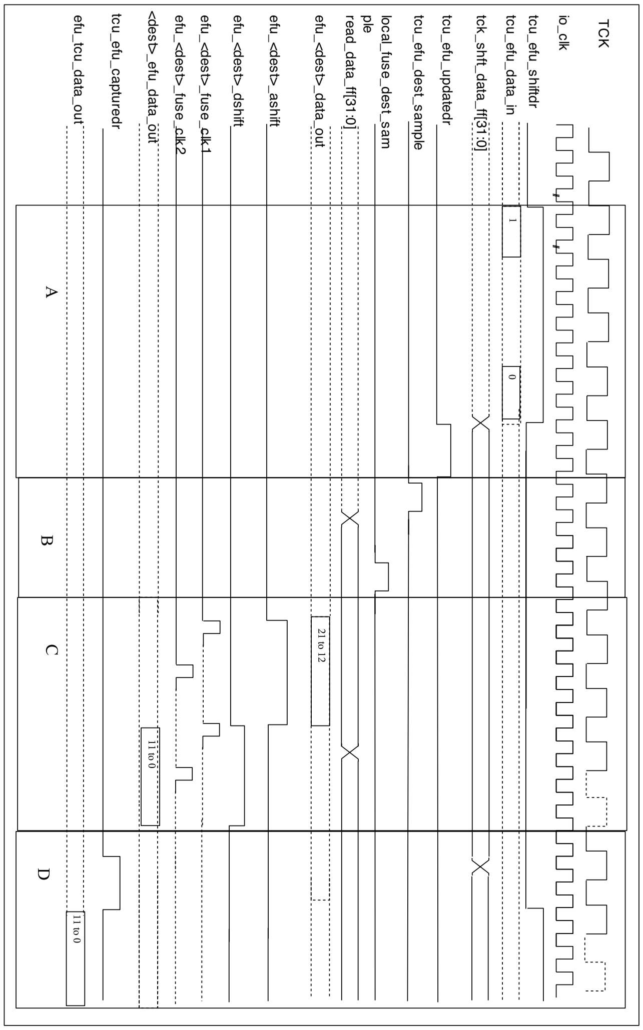
3. TAP_CAPTUREDR
4. TAP_SHIFTDR

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 4-6 group D). Bit 31 of the tck 32 bit register, tck_shft_data_ff[31:0], will be shifted out first.

FIGURE 4-6 Destination Sample Mode Timing Diagram



4.3.2 Interface with NCU, SRAM Header Flops and TCU Destinations

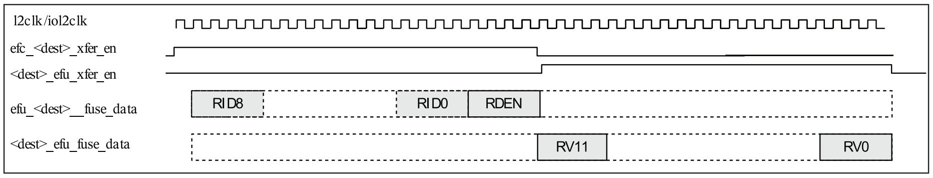
4.3.2.1 EFU to SRAM Header Flops

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 FIGURE 4-7.

4.3.2.2 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 appropriate SRAMs with the correct sync_en signals.

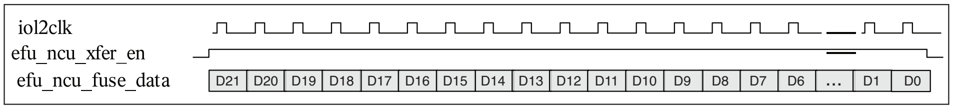
FIGURE 4-7 SRAM to EFU Data Transfer Timing Diagram



4.3.2.3 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 4-8 EFU to NCU Interface Timing Diagram



4.3.2.4 TCU to EFU Transfers

The protocol for transferring 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.

4.3.2.5 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]).

4.3.3 Register Formats

4.3.3.1 RV REGISTER CLEAR ID

The following are the seven bit rv register clear ID. When bit seven is high the clear function is enable. When bit seven is low the clear function is disable. When all seven bits are high all rv clear signals are active.

TABLE 4-5 Seven Bit Block ID for Memories

Field	Bits	Reset Name
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 4-5 Seven Bit Block ID for Memories *(Continued)*

Field	Bits	Reset Name
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)

TABLE 4-5 Seven Bit Block ID for Memories *(Continued)*

Field	Bits	Reset Name
niu_ram_clr	1100001	Clear all bits in the rv registers (TDS TDMC)
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

4.3.3.2 Block ID

TABLE 4-6 shows the six-bit block IDs for destination:

TABLE 4-6 Six Bit Block IDs for Memories

Field	Bits	Reset Name	Reset Value
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

TABLE 4-6 Six Bit Block IDs for Memories *(Continued)*

Field	Bits	Reset Name	Reset Value
9	Core4 I$	001000	SPARC core 4 Icache repair information
10	Core4 D$	001001	SPARC core 4 Dcache repair information
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

TABLE 4-6 Six Bit Block IDs for Memories *(Continued)*

Field	Bits	Reset Name	Reset Value
46		101101	
47		101110	
48		101111	
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	

4.3.3.3 SRAM Redundancy Register Formats:

There are four 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.

4.3.3.4 L2 Data Array EFA Entry Definition

For the L2 Data array, the EFA entry is stored in the format shown in TABLE 4-7.

TABLE 4-7 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 eight 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 4-8 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 for row repairs in TABLE 4-9.

TABLE 4-9 L2 Tag Array RID/RV Field Description

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.)
[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 for column repairs in TABLE 4-10.

TABLE 4-10 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 4-11 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

4.3.3.5 L1 INSTRUCTION CACHE (ICD) EFA Entry Definition

For the L1 ICD, the RID/RV fields are defined for column repairs in TABLE 4-12

TABLE 4-12 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

4.3.3.6 L1 data cache array redundancy register (DCD) definition

For the L1 DCD, the RID/RV fields are defined as column repairs in TABLE 4-13.

TABLE 4-13 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)

TABLE 4-13 L1 DCD RID/RV Field Descriptions for Column Repair *(Continued)*

Bits	Size	Description
[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)

4.3.3.7 Core Available

TABLE 4-14 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.)

4.3.3.8 L2 Bank Available

TABLE 4-15 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.)

4.3.3.9 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)
```

4.3.3.10 DMU DATA Registers

TABLE 4-16 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 4-17 DMU Read Data Format

Bits	Size	Description
[21:4]	18	ignored
[3:0]	4	Readback fuse data; when clear the data is 4'b0010

4.3.3.11 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 two 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 shown in TABLE 4-18,TABLE 4-19, and TABLE 4-20.

TABLE 4-18 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 4-19 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 eight 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 two or three 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 format is shown in TABLE 4-20,

TABLE 4-20 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.

4.4 Unit-Level Interface Signals

TABLE 4-21 Unit-Level Interface Signals

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
tcu_efu_capturedr	Input	1	Shift data register captures read register value

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
spc1_efu_fuse_ixfer_en	Input	1	SPARC core 1 I$ read transfer enable
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

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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
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

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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
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

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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
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

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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
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)

TABLE 4-21 Unit-Level Interface Signals *(Continued)*

Signal name	Direction	Size	Description
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
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

4.5 Miscellaneous/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
```

4.6 eFuse Array Specification

4.6.1 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.

4.6.2 eFuse Array Functions

Supports the following two functions:

1. Read access: There can be two 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.

4.6.3 Timing Diagrams

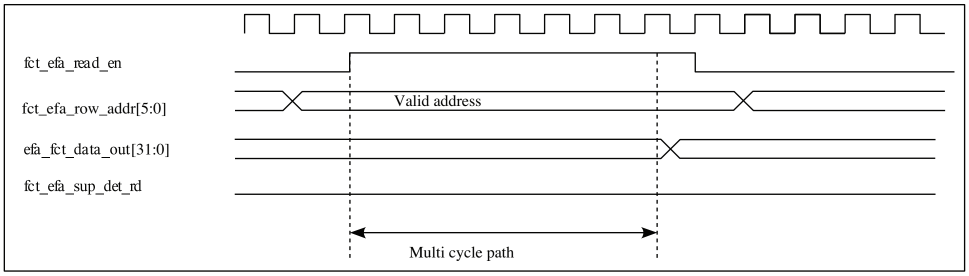
Read access:

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.

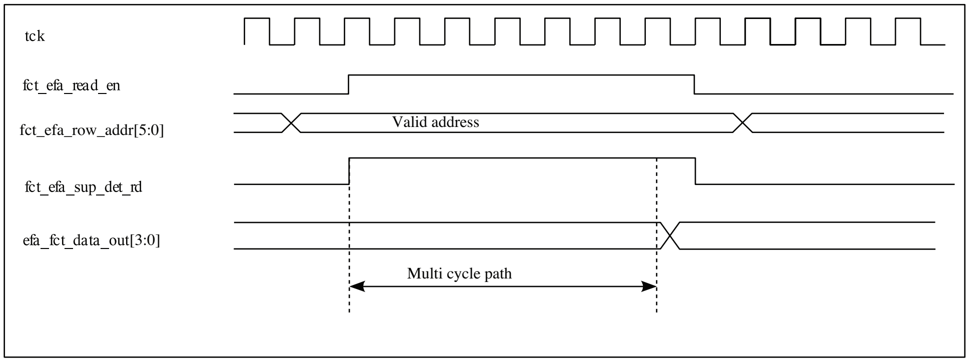
FIGURE 4-9 EFA Row Read Access



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 4-10 EFA Supply Detect Access



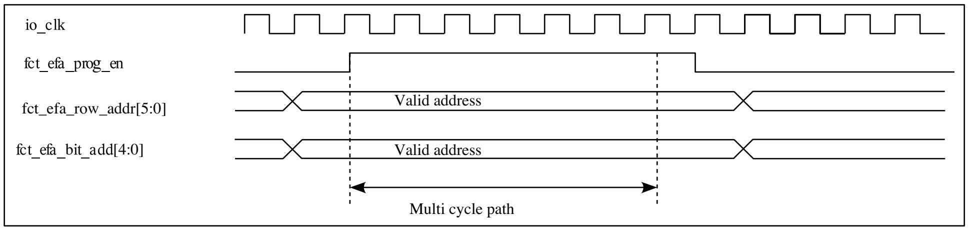
Program access:

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 4-11 EFA Program Access



4.6.4 Interface Table

TABLE 4-22 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 5

Reset Unit Specification

This chapter contains the following sections:

5.1 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 5-1 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

5.2 Reset Overview

5.2.1 Goals

The Reset Unit asserts signals that cause other units to immediately revert to the initial state defined by the *OpenSPARC T2 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.

5.2.2 Nomenclature

In the specifications relevant to OpenSPARC T2, the term *reset* is used in many ways. TABLE 5-2 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, 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 5-2 presents reset functions by configuration.

TABLE 5-2 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	

5.2.3 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 5-3. Priority 1 traps, which are resets other than POR, are processed in the following order: XIR(3) > WDR(2) > SIR(4) > RED(5).

Reset priorities from highest to lowest are:

POR(1)>WMR(1)>XIR(3)>WDR(2)>SIR(4).

TABLE 5-3 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 5-4 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 TABLE 5-3.
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.

5.2.4 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. eFuse Array.
   b. SRAM array cores.

A 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.) A 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.) A 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.

5.2.5 eFuse destination Flops and Latches

eFuse OpenSPARC T2 Micro-Architecture Specification lists the destinations of information from the EFU, as shown in TABLE 5-5.

TABLE 5-5 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 5-5 Destination of Information from the EFU *(Continued)*

Block ID	Destination	Information
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 eFuse Unit, before the transfer starts, using flash (synchronous) reset. A 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.

5.2.6 Latches

TABLE 5-6 lists each kind of latch, the agent that sets it to its initial value, and method.

TABLE 5-6 Latch Kind

Latch	Agent that initializes value	Method of initializing
SRAM redundant array Repair Value valid bit latches	TCU via eFuse Unit, before the transfer starts	Flash (synchronous) reset
Other SRAM redundant array Repair Value latches	eFuse Unit	Write to latch. (These bits have no effect if the valid bit is cleared.)
Other SRAM latches	(Not resettable.)	(See Flip-Flops Outside of SRAMs)
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 Types of Reset.

5.2.7 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 5-7 lists each kind of flip-flop, except for SRAM input and output flops:

TABLE 5-7 Types of Flip-Flops

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 5-7 Types of Flip-Flops

Flip-flops outside of SRAMs	Agent that initializes value	Method of initializing
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.

5.2.8 SRAM Input Flops

TABLE 5-8 lists each kind of SRAM input flop:

TABLE 5-8 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

5.2.9 SRAM Output Flops

TABLE 5-9 lists each kind of SRAM output flop:

TABLE 5-9 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.

5.2.10 Core Array Contents

TABLE 5-10 lists each kind of core array contents:

TABLE 5-10 Core Array Contents

Core array contents	Agent that initializes value	Method of initializing
eFuse Array contents	Factory	Fuse blow
eFuse 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 Types of Reset.

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 eFuse-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.

5.2.11 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.

5.2.12 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.

5.2.12.1 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.

   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.

5.2.12.2 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.

5.2.12.3 Two Signals Require Asynchronous 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.

5.3 Types of Reset

5.3.1 TRST_

TRST_ only involves the TCU, and not RST. The *IEEE 1149.1 JTag Specification* 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 five clock cycles.

5.3.2 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.

5.3.3 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.

5.3.4 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. See Reset Signals Asserted for each Kind of Reset.
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 A Fatal Error causes a WMR.)

5.3.4.1 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 eight 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.

5.3.4.2 Conflicting Demands placed on WMR

Warm reset serves two purposes:

1. Test

2. Fatal error

Test involves hundreds of functional vectors. For example, TABLE 5-11 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.

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.

TABLE 5-11 Chip Reset

Chip reset step	Jclks	Percent of diags
POR/PLL	34800	27
EFA	8000	6
WRM	4000	3
BISI	2600	2

TABLE 5-11 Chip Reset *(Continued)*

Chip reset step	Jclks	Percent of diags
SSI	27800	22
Total reset steps	77200	60
Total for 145 diags	127200	100
Diag portion other than reset	50000	40

5.3.5 WMR Trap and SPARC-V9 POR Trap

The WMR trap generates a SPARC-V9 POR trap, which has a trap type of 1.

5.3.5.1 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.
2. That virtual core then starts each of the others with an interrupt.

From the *OpenSPARC T1 Programmer's Reference Manual*:

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.

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."

5.3.5.2 How OpenSPARC T2 Starts its Virtual Cores at Reset

1. For OpenSPARC T2, the eFuse Cluster scans out the eFuse 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.

“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 set, 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 is located at an implementation-dependent address referred to as RSTVaddr.

"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.

5.3.6 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.

5.3.6.1 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."

5.3.7 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 OpenSPARC T2, a WDR also can result from a fatal error condition detected by on-chip error logic. A WDR only affects the strand which created it. When a WDR is recognized, instruction fetching begins at RSTVaddr || 0x40.

5.3.7.1 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."

5.3.7.2 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 a individual virtual core and are not propagated to the other virtual cores on a CMP."

5.3.8 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 TABLE 5-11, 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.”

5.4 Machine State after Each Kind of Reset

TABLE 5-12 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 5-12 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 s 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.

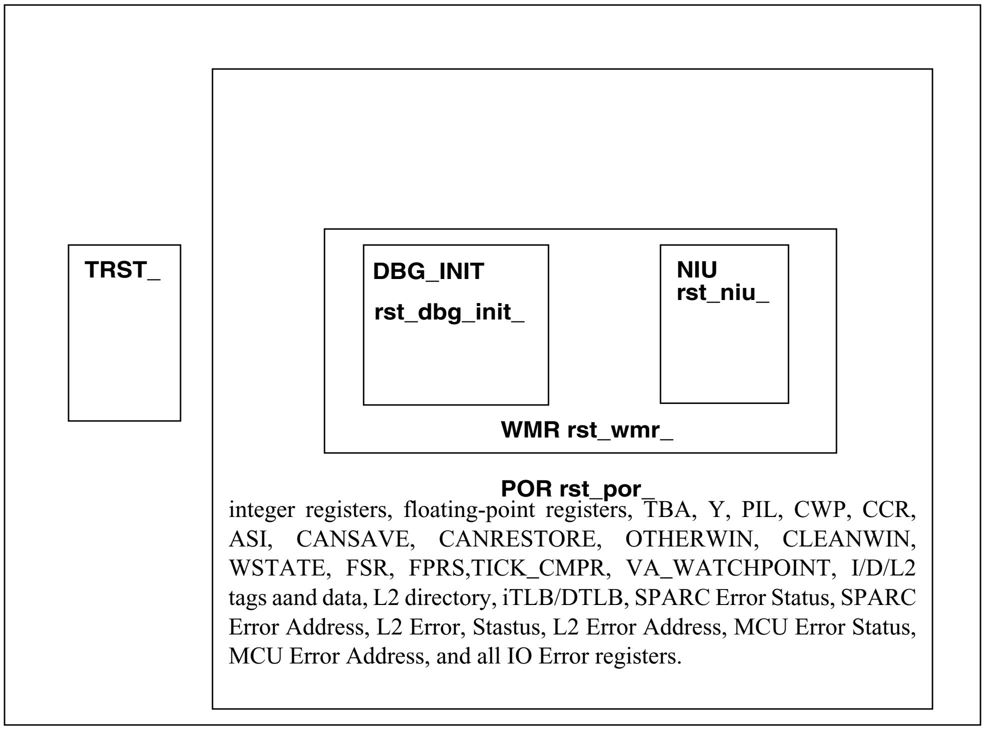
TABLE 5-13 Cleared Arrays

	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.

5.4.1 Venn Diagram

Rectangles in FIGURE 5-1 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 5-1 Venn Diagram


TRST_
DBG_INIT
rst_dbg_init_
NIU
rst_niu_
WMR rst_wmr_
POR rst_por_
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 aand data, L2 directory, iTLB/DTLB, SPARC Error Status, SPARC Error Address, L2 Error, Stastus, L2 Error Address, MCU Error Status, MCU Error Address, and all IO Error registers.

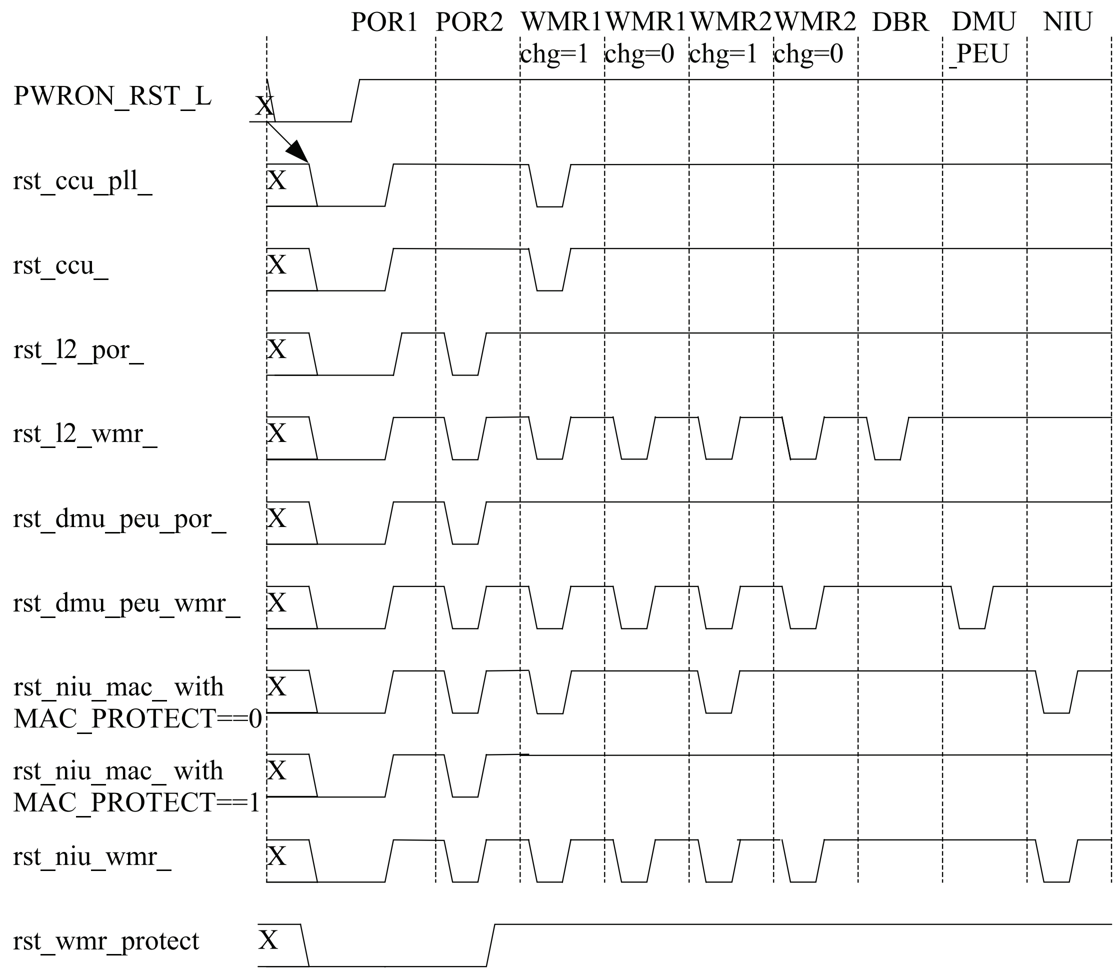
5.4.2 Reset Signals Asserted for each Kind of Reset

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 5-2, 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.

FIGURE 5-2 Reset Signals



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 5-2 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.

5.4.3 POR Clears the Valid Bits in the L2T Directory of L1 Tags CAM

To guarantee coherency and correct functionality, initialize the arrays shown in TABLE 5-14 before enabling the L2 cache:

TABLE 5-14 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];
```

This 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.

Note – 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.

Note – 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 5-15 shows the state of the cache lines.

TABLE 5-15 CPU State after Reset and in RED_state

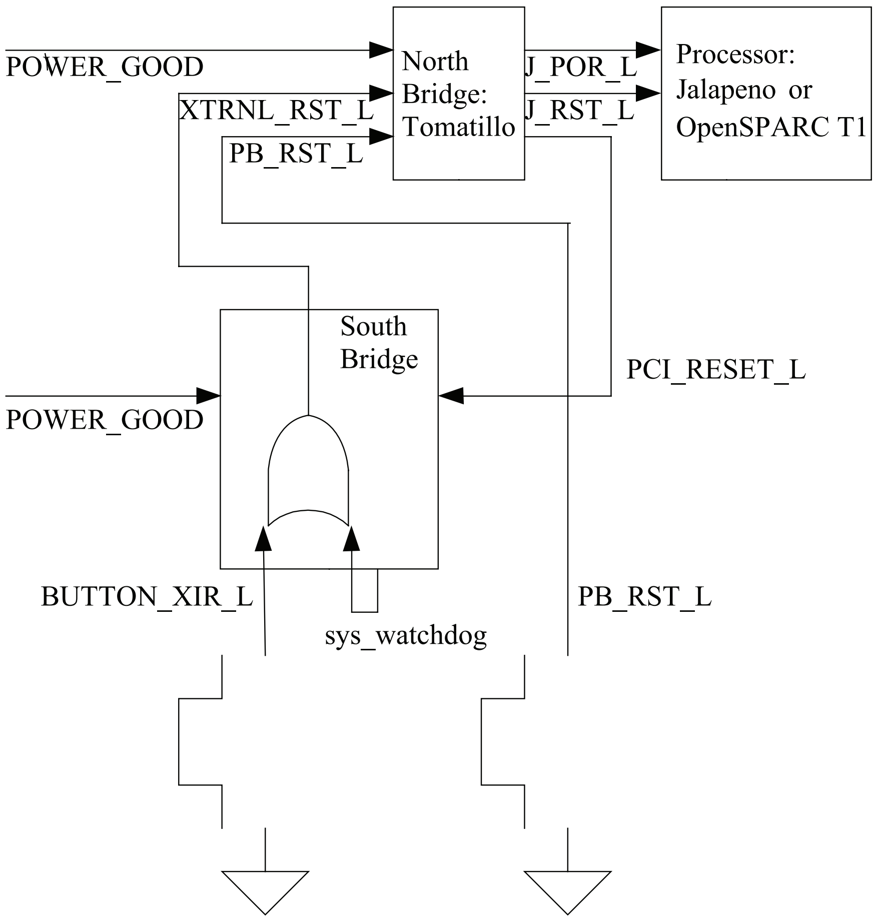
Structure that holds state	POR	WMR
I/D cache tags	All invalid	Unchanged if BISI not run, else invalid
L2 tags and data	Unknown	
L2 directory	All invalid	

5.5 OpenSPARC T2 is a System On a Chip

5.5.1 System On a Board

FIGURE 5-3 shows a possible configuration for an 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.

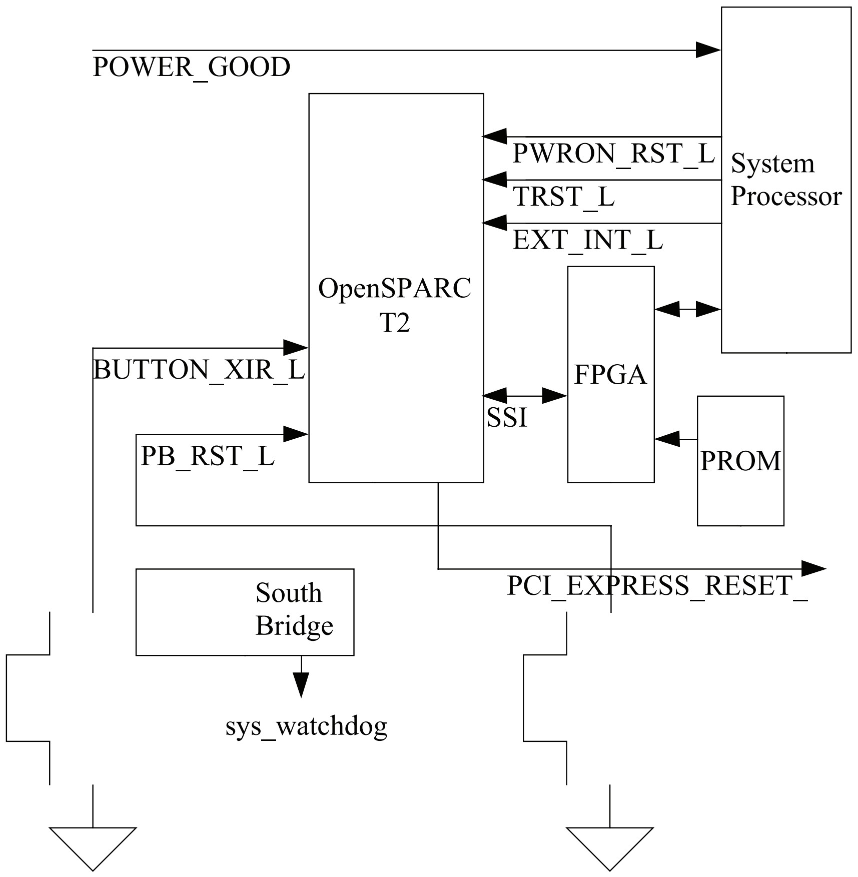
FIGURE 5-3 System On a Board



5.5.2 System On a Chip

FIGURE 5-4 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.

FIGURE 5-4 System On a Chip



5.5.3 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
2. 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 OpenSPARC T2 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, OpenSPARC T2 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 OpenSPARC T2 design team that OpenSPARC T2 needs to indicate to the FPGA on a pin when it should ignore the SSI_SCK and SSI_MOSI outputs from OpenSPARC T2 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 OpenSPARC T2. By this point OpenSPARC T2 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.

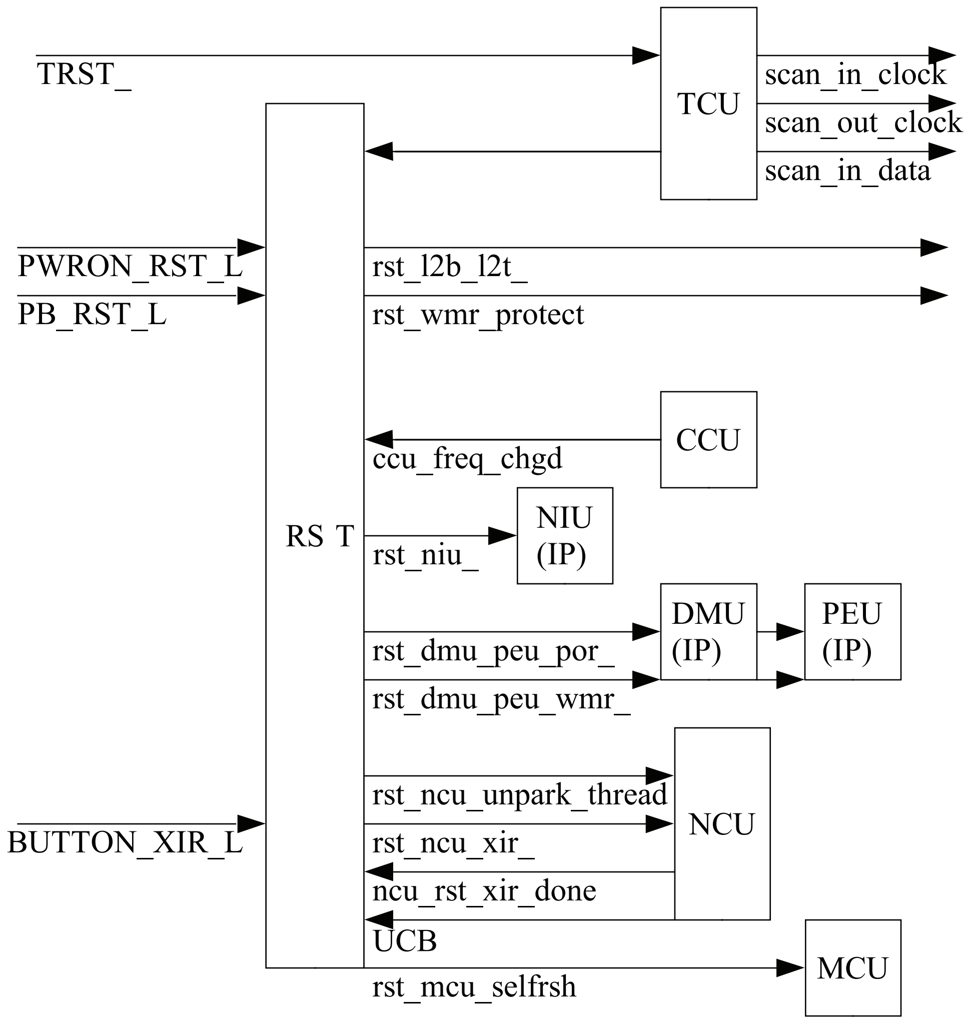
Assert after flush reset, but before rst_ncu_unpark_thread.

Deassert before flush reset of NCU.

5.5.4 Connections between RST and Other Clusters

FIGURE 5-5 shows some connections between RST and other clusters. See the Reset Unit Verification Test Plan for a more complete depiction.

FIGURE 5-5 Connections between RST and Other Clusters



5.6 Registers

5.6.1 (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 5-16 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. “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. The section entitled "POR & Warm Reset Initialization", implies that POR and Warm Reset are two different things. Also, the Fire Power-On, Reset and BIST document says, "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.

5.6.2 (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 5-17 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.

TABLE 5-17 Reset Source Register *(Continued)*

Field	Bit Position	Initial Value	R/W	Description
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.

5.6.3 (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 5-18 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.)
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.

5.6.4 (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. OpenSPARC T2 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

Register Base Address 1 IOBMAN – 0x98-0000-0000

The chip reset status, shown in TABLE 5-19 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.

TABLE 5-19 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
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.
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.

5.6.5 (0x89-0000-0820) Fatal Error Enable Register, RESET_FEE

Each bit of this register allows the l2t*n*_rst_fatal_error signal, 0 equal or less than *n* equal or less than 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 5-20 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.
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

5.6.6 (0x89-0000-0860) Clock Control Unit Time Register, CCU_TIME

TABLE 5-21 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.

5.6.7 (0x89-0000-0870) Lock Time Register, 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 eFuse 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.

TABLE 5-22 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

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 eFuse Cluster scans the eFuse Array, and the ddr_pll and NIU PLLs lock. [eFuse 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 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.

5.6.8 (0x89-0000-0880) Propagation Time Register, PROP_TIME

TABLE 5-23 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 Network Interface Unit (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. These 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/comp
iled/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");
      }
```

Taking the entries in the middle column and the middle rows of the cell_fall table and the cell_rise table, the following values are obtained, in nanoseconds:
0.161900
0.166500
0.158600
0.165100

IWith an original estimate of 250 ps per stage, a 50 percent margin over the slowest of these values is provided, in addition to the 22 percent margin already provided.

5.6.9 (0x89-0000-0890) NIU Time Register, NIU_TIME

TABLE 5-24 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).

5.7 Power-On Reset Sequence Overview

This section summarizes what the following sections lay out in detail.

TABLE 5-25 shows the major types of OpenSPARC T2 structures that hold state (see OpenSPARC T2 Structures that Hold State). 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 eFuse Unit, "efu",
- has taken on a value, either 0 or 1, that is deterministic and repeatable, "det".

TABLE 5-25 Types of Structures

	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 5-7 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. A reset after BISI, and another after BIST were added. 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.

5.7.1 Power-On Reset Duration in a System

TABLE 5-26 sums up the duration of each step of the Power-On Reset sequence, in which OpenSPARC T2 is in a system:

TABLE 5-26 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_stabl e	5	0.714	3.57
Deassert Asic clk_stop	tcu	ccu_rst_sync_sta ble	tcu_rst_flush_stop _ack	2*128 = 256	0.714	182.86
niu pll	rst	tcu_rst_flush_sto p_ack	tcu_rst_flush_stop _req	NIU_TIME = 1,600	5.000	8,000.00
Deassert SunV clk_stop	tcu	tcu_rst_flush_sto p_req	tcu_rst_flush_stop _ack	22*128 = 2,816	0.714	2,011.43
efu1	efu	tcu_rst_flush_sto p_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14
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_ini t_req	tcu_rst_flush_init_ ack	24*128 = 3,072	0.714	2,194.29
por2	rst	tcu_rst_flush_ini t_ack	tcu_rst_flush_stop _req	PROP_TIME = 3,000	5.000	15,000.00
Deassert clk_stop	tcu	tcu_rst_flush_sto p_req	tcu_rst_flush_stop _ack	24*128 = 3,072	0.714	2,194.29
efu2	efu	tcu_rst_flush_sto p_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14
	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

TABLE 5-26 Power-On Reset Sequence Duration *(Continued)*

POR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
SSI pll locks	ncu	-	-	3FFFF = 256k = 262,144	11.428	2,995,931.42
	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End POR sequence	-	-	-	-	-	3,175,439.98

5.7.2 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 5-27 sums up the duration of each step of this minimal-delay Power-On Reset sequence:

TABLE 5-27 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
Deassert SunV clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	22*128 = 2,816	0.714	2,011.43
efu1	efu	tcu_rst_flush_stop_ack	tcu_rst_efu_done	64*62 = 3,968	2.857	11,337.14

TABLE 5-27 Power-On Reset Duration on Tester *(Continued)*

POR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
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
Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
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
	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End POR sequence	-	-	-	-	-	179,508.56

5.7.3 Warm Reset Duration in a System

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 5-28 sums up the duration of each step of this maximum-delay Warm Reset sequence:

TABLE 5-28 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
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
Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
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
Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
	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
	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End WMR sequence	-	-	-	-	-	19,894,314.27

5.7.4 Warm Reset Duration on a Tester

Finally, 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 5-29 sums up the duration of each step of this minimal-delay Warm Reset sequence:

TABLE 5-29 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_init_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_stable	5	0.714	3.57
Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
bist2	tcu	tcu_rst_efu_done	tcu_bisx_done	0	0.714	0.00
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
Deassert clk_stop	tcu	tcu_rst_flush_stop_req	tcu_rst_flush_stop_ack	24*128 = 3,072	0.714	2,194.29
	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

TABLE 5-29 Minimum Warm Reset Duration *(Continued)*

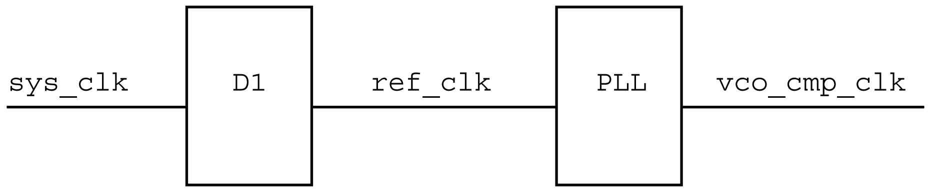
WMR step	Who	Start	End	Cycles	Clock period (ns)	Duration (ns)
SSI PLL locks	ncu	-	-	0	11.428	0.00
	ncu	Request from spc to ncu	Data on SSI bus	7	2.857	20.00
End WMR sequence	-	-	-	-	-	38,822.85

5.8 Deterministic Behavior

Any sequence of actions on OpenSPARC T2 is required to be deterministic and 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 5-6 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 re times 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.

5.9 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 5-7, FIGURE 5-8, FIGURE 5-9, and FIGURE 5-10 show the entire Power-On Reset sequence. Numbers on the figures indicate a step or steps in the reset sequence and a step, in this section, will correspond to each number. A solid arrow from one step to another indicates that the completion of the first step causes the second step to begin. By contrast, there are three dashed arrows:

1. From the assertion of POWER_GOOD to deassert TRST_.
2. From deassert TRST_ to After PWRON_RST_L.
3. From PLLs lock to After PWRON_RST_L, via CCU.

These indicate not cause and effect, but rather the ordering of these steps that the System Processor will impose.

There are six sections describing POR:

1. During PWRON_RST_L (including POR 1).
2. After PWRON_RST_L (including POR 2).
3. Pre-WMR boot code.
4. During WMR 1.
5. After WMR 2.
6. Post-WMR boot code.

FIGURE 5-7 Sequence - Start of POR1

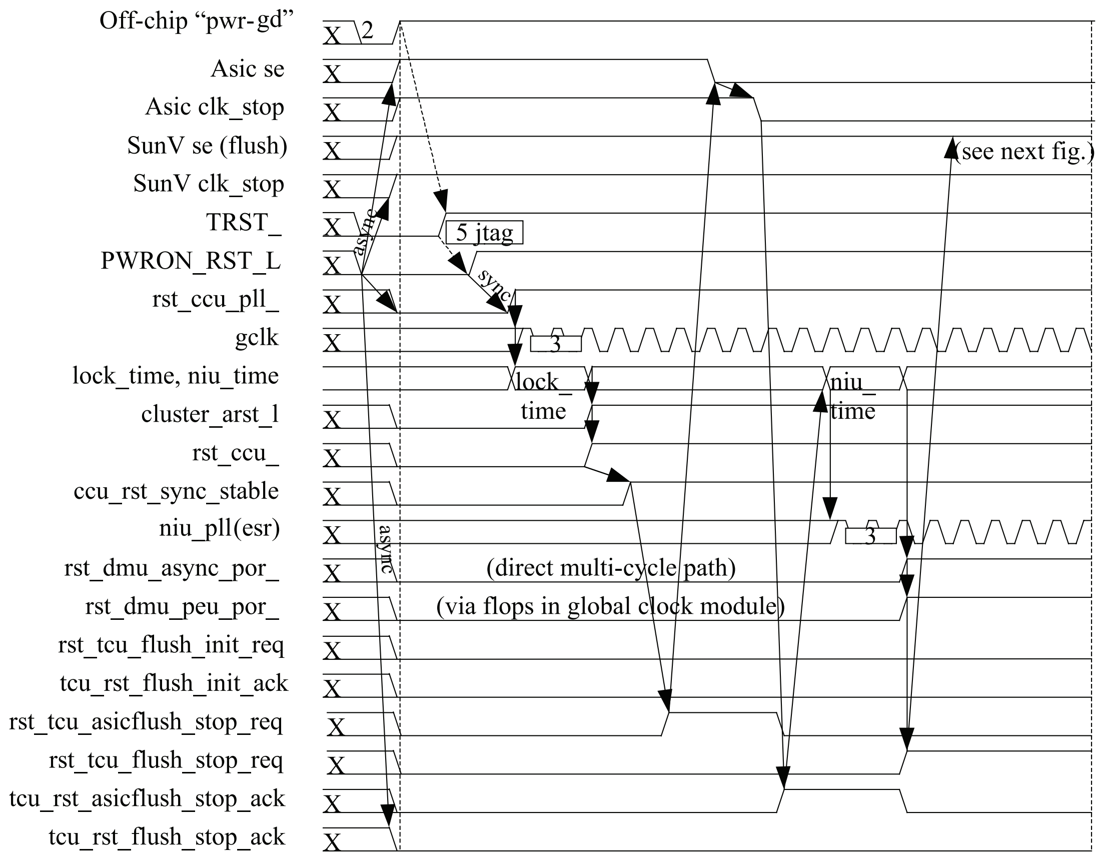


FIGURE 5-8 Sequence - End of POR1

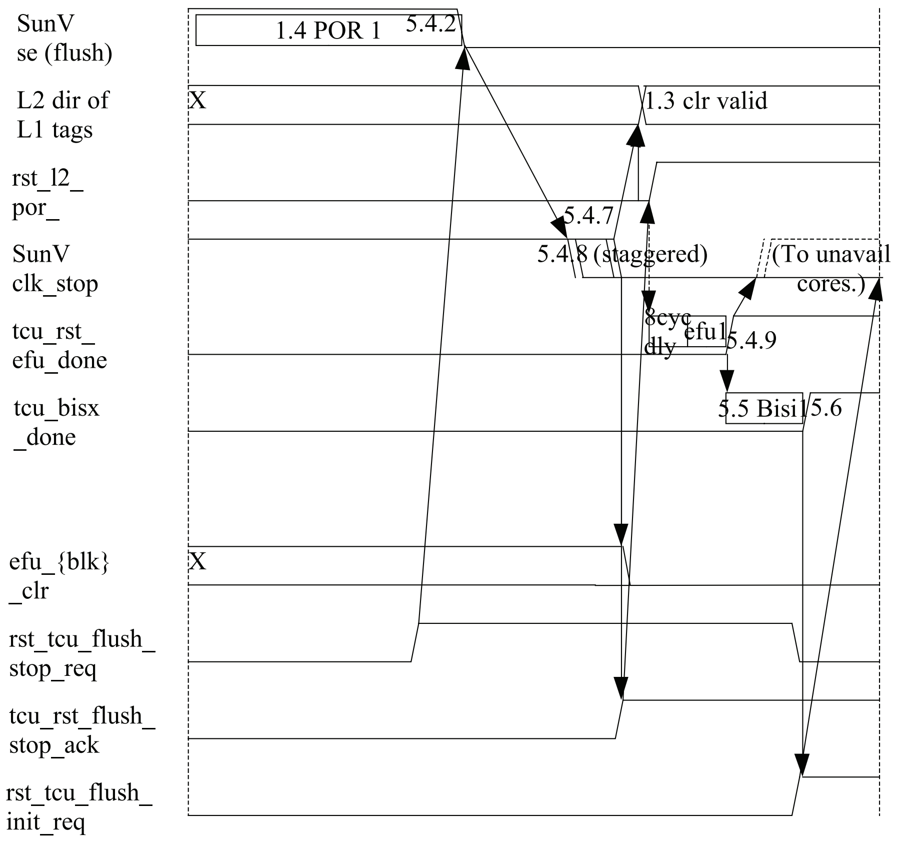


FIGURE 5-9 Reset Sequence - POR2

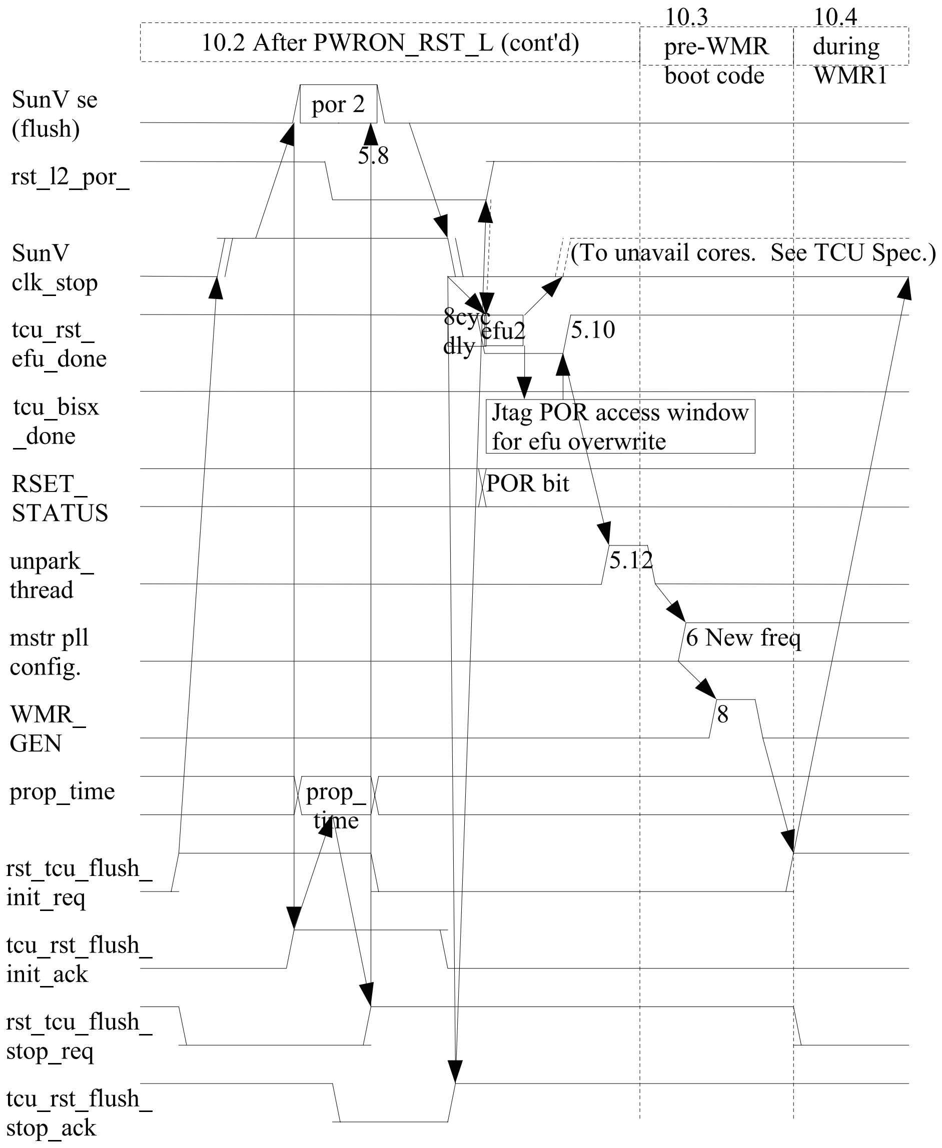
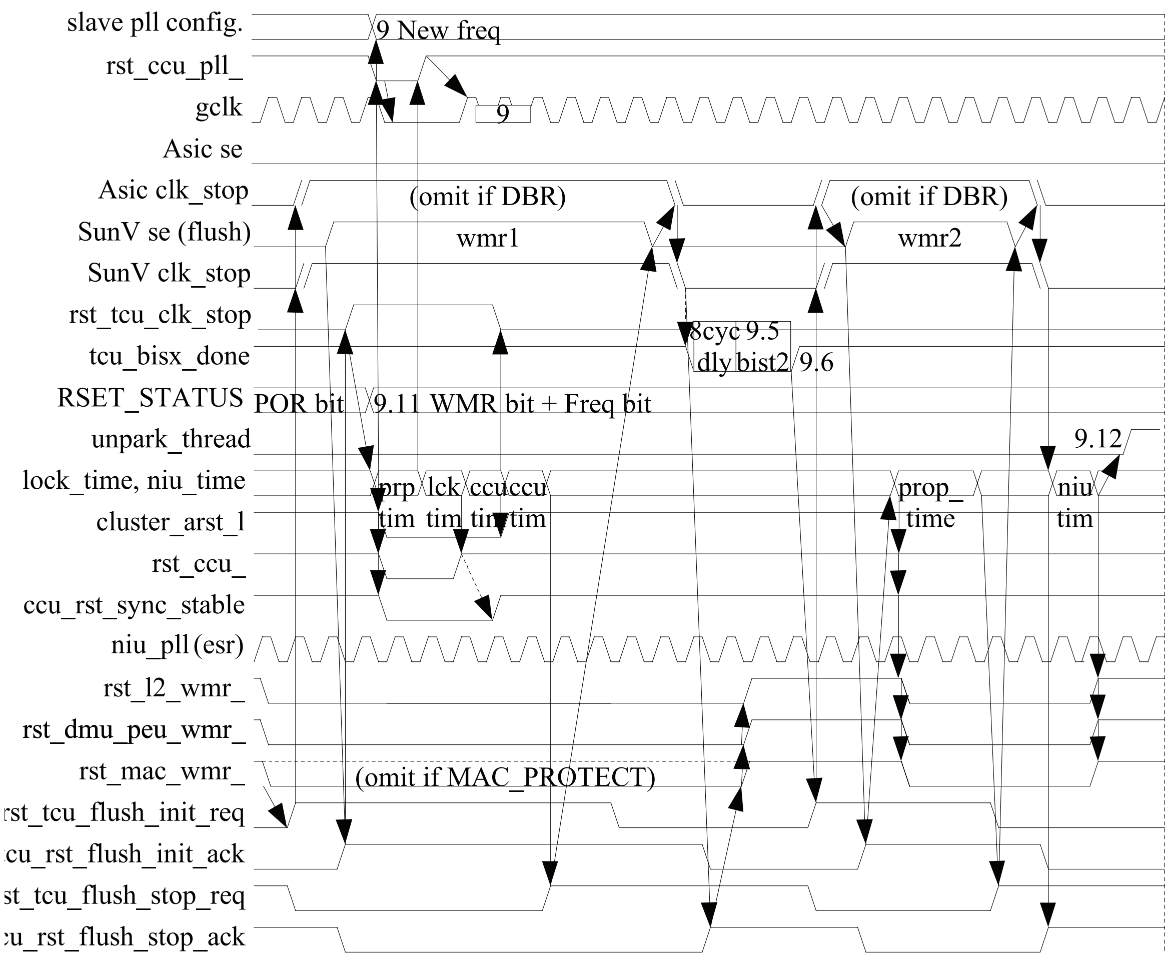


FIGURE 5-10 Reset Sequence - Warm Reset: WMR1+WMR2



5.9.1 During PWRON_RST_L (including POR1)

1. Service processor asserts TRST_ and PWRON_RST_L. See FIGURE 5-7.

a. TRST_ event will initialize the PLL config. register to a default setting.

Note – 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 5-8 shows this as TCU asserting se, scan enable, during the interval labeled “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 During WMR1 correspond to steps After WMR in During WMR1 and After WMR, starting with PLLs lock. An exception is that the portions of After PWRON_RST_L (including POR2) 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.

5.9.2 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_.

5.9.3 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 eFuse 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: eFuse Unit, EFU, scans out eFuse 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: eFuse Unit, EFU, scans out eFuse Array, EFA. EFU asserts efu_done. This restores values that POR 2 cleared. Now that the eFuse 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).

5.9.4 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.

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.

5.9.5 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."

These steps correspond to During PWRON_RST_L (including POR1), starting with "PLLs lock."

a. PLLs lock.

b. 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.

5.9.6 After WMR

c. 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.

d. 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."

e. TCU asserts tcu_bisx_done.

f. 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 5-10 shows this sequence as "WMR 2". RST asserts rst_niu_, and rst_dmu_peu_wmr_, and TCU asserts se.

g. After lock_time clock cycles, TCU deasserts se.

h. 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.

i. (eFuse Unit, EFU, remains idle during WMR.)

j. RST sets the WMR bit of the RSET_STATUS register. Since the frequency changed, it also sets the FREQ bit.

k. RST asserts rst_unpark_thread to NCU. NCU asserts core_running to Trap Unit of lowest-numbered enabled SPC.

5.9.7 Post-WMR Boot Code

l. 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."

m. Post-WMR boot code starts by reading RSET_STAT register, which indicates WMR, with clock change.

10. TCU already launched BIST on caches, in Power-On Reset Sequence - End of POR1.

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. Enable error detection on L1 and L2 caches.

13. Enable L1 and L2 caches.

14. Post-WMR boot code continues as outlined in *Programmer's Reference Manual*.

5.10 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.

5.10.1 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 Pre-WMR Boot Code.
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.

5.10.2 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 5-10 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.

1. PLLs lock.

2. 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.
3. 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.
4. The Reset Unit waits for a number of cycles of the reference clock. The Lock Time register holds that number.

5.10.3 After rst_wmr_

1. 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.
2. TCU asserts tcu_bisx_done.
3. 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 5-10 shows this sequence as “WMR 2”. RST asserts rst_wmr_, and TCU asserts se.
4. After lock_time clock cycles, TCU deasserts se.
5. 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.
6. eFuse Unit (EFU) remains idle during WMR.
7. RST sets the WMR bit of the RSET_STATUS register. In addition, if the frequency changed, it also sets the FREQ bit.
8. RST asserts rst_unpark_thread to NCU. NCU asserts core_running to Trap Unit of lowest-numbered enabled SPC.

5.10.4 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 After WMR)

Check local error logs.

Post-WMR boot code continues as outlined in *Programmer's Reference Manual*.

5.11 Reset Sequence for DBG

DBG is the same as WMR, except that the Reset Unit does not reset DMU, PEU, nor NIU.

5.12 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.

5.13 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.

5.14 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
3	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.

5.14.1 tcu_rst_clk_stop

```
module rst...

The Reset Unit gates tcu_rst_clk_stop with tcu_rst_scan_mode:

  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.

5.14.2 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.

5.15 Reset Unit Ports

5.15.1 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 5-30 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.

TABLE 5-30 Inport Ports Clocks *(Continued)*

Source	Clock	Input port	
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	
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	

5.15.2 Output Ports

TABLE 5-31 lists the output ports of the Reset Unit.

TABLE 5-31 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	FIGURE 5-10
	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 5-31 Output Ports Clocks *(Continued)*

Sink	Clock	Ultimate clock	Output port	
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. Eight 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

5.16 Appendices

5.16.1 OpenSPARC T1 Thread Suspension Differs from CMP Suspend

At first, one might suspect that OpenSPARC T1thread suspension could have something to do with the reset unit.

Each OpenSPARC T1thread is in one of three states:

1. halt,
2. idle, or
3. active.

The *OpenSPARC T1 Programmer's Reference Manual* refers to halt and idle as "inactive" states. The *OpenSPARC T1 Programmer's Reference Manual* section describing these states has as its title "Thread Suspension", but that is the only place the *OpenSPARC T1 Programmer's Reference Manual* uses the term "suspension"

The two-bit TYPE field of the OpenSPARC T1 INT_VEC_DIS register can take on values named interrupt and reset, but these differ from all other interrupts and resets. Neither the SPARC Architecture Manual, nor the Sun SPARC Spec., nor the Sun Microsystems Standard CMP Programming Model, mentions INT_VEC_DIS.

Halt, idle, active, inactive, and suspension, as used in the *OpenSPARC T1 Programmer's Reference Manual*, differ from the disable, enable, suspend, park, and run states described in Sun Microsystems Standard CMP Programming Model.. Due to scheduling pressures, the OpenSPARC T1 project received an exemption from the corporate requirement to conform to the CMP specification. The OpenSPARC T1 states are controlled by INT_VEC_DIS, whereas the CMP states are controlled by the registers described in the next section, CMP Disabling and Parking of Virtual Cores.

TABLE 5-32 summarizes the meaning of each OpenSPARC T1 thread suspension term:

TABLE 5-32 Thread Suspension

Will respond to interrupts	**Execute instructions**	
	No Inactive	Yes
No	Idle	(No such status)
Yes	Halt	Active

In conclusion, OpenSPARC T1 thread suspension has nothing to do with the reset unit.

5.16.2 CMP Disabling and Parking of Virtual Cores

At first, one might suspect that CMP disabling and parking of virtual cores could have something to do with the reset unit.

OpenSPARC T1 has a CORE_AVAIL register, set from the eFuse, EFU. It does not have ASI_CORE_AVAILABLE, nor any of the following registers and state definitions for disabling and parking virtual cores.

Shared Machine State registers TABLE 5-33, all at ASI# = 0x41, have one instance shared among the virtual cores.

TABLE 5-33 CMP Shared Machine State

Register ASI name	VA	Access	SP?	Note
ASI_CORE_AVAILABLE	0x00	RD only	Yes	-
ASI_CORE_ENABLE_STATUS	0x10	RD only	Yes	-
ASI_CORE_ENABLE	0x20	RD/RW	Yes	Takes effect after reset
ASI_XIR_STEERING	0x30	RD/WR	Yes	General access
ASI_CMP_ERROR_STEERING	0x40	RD/WR	Yes	-
ASI_CORE_RUNNING_RW	0x50	RD/RW	Yes	General access
ASI_CORE_RUNNING_STATUS	0x58	RD only	Yes	-
ASI_CORE_RUNNING_W1S	0x60	W1S	No	Write 1 to set bit(s)
ASI_CORE_RUNNING_W1C	0x68	W1C	No	Write 1 to clear bit(s)

ASI_CORE_AVAILABLE is located in the CMP unit, but takes inputs from the eFuse unit. As discussed in the section on nomenclature, XIR and the ASI_XIR_STEERING register involve the CMP and SPG units, and not the reset unit.

TCU may choose to implement debug features for SoC units analogous to CMP disabling and parking of virtual cores, but these features would not involve reset.

The following registers have an instance per core, and so are best implemented in each SPARC Gasket block, SPG.

ASI_SCRATCHPAD_0_REG
ASI_SCRATCHPAD_1_REG
ASI_SCRATCHPAD_2_REG
ASI_SCRATCHPAD_3_REG
ASI_SCRATCHPAD_4_REG
ASI_SCRATCHPAD_5_REG
ASI_SCRATCHPAD_6_REG
ASI_SCRATCHPAD_7_REG
ASI_INTR_ID
ASI_CORE_ID

The CMP spec uses the terms in TABLE 5-34.

TABLE 5-34 CMP Specification Terms

CMP term	Definition
disable	Will complete at next system or power-on reset.
disabled	Execute no instructions (suspended, parked) nor maintain cache coherency.
enable	Will complete at next system or power-on reset.
enabled	Maintain cache coherency.
suspend, park	Execute no instructions. If enabled, maintain cache coherency.
running, unparked	Execute instructions and maintain cache coherency

TABLE 5-35 summarizes the meaning of each CMP term:

TABLE 5-35 CMP Term Meaning

Maintain cache coherency	Execute instructions	
	No Suspend, park	Yes
No	Disabled (implicitly suspend disabled, park disabled)	(No such status)
Yes, enabled	Suspend enabled, park enabled	Run (implicitly run enabled)

In conclusion, CMP disabling and parking of virtual cores has nothing to do with the reset unit.

5.16.3 OpenSPARC T1 Reset Sequence

OpenSPARC T2 is not strictly compliant with the letter of the JBUS spec, in regards to reset, and specifically cold power-on reset. The "problem" is that OpenSPARC T2 receives only J_POR_L, and not the more basic POWER_OK signal, so OpenSPARC T2 starts the PLL lock sequence on the deassertion of J_POR_L, and starts propagating clocks to internal blocks after PLL lock is achieved. This means that most of OpenSPARC T2 , including the JBI, does not get any clocks for microseconds after J_POR_L deassertion.

The effects of this is that OpenSPARC T2 's JBI output is undefined for those microseconds after POR deassertion until PLL lock. OpenSPARC T2 correctly disables its outputs during J_POR_L assertion. After J_POR_L deassertion, OpenSPARC T2 JBUS outputs are undefined until OpenSPARC T2 's internal PLL locks, and the CTU starts distributing clocks. Once clocks are being distributed, OpenSPARC T2 will correctly drive IDLE transactions, until the end of J_RST_L assertion. Since J_RST_L was being asserted while OpenSPARC T2 's outputs were undefined, and since this is only at cold power-on, other chips should safely ignore OpenSPARC T2 's undefined behavior.

A second caveat on reset is that OpenSPARC T2 assumes that it will drive the first JBUS transaction, long after J_RST_L deassertion. All non-PLL initialization of OpenSPARC T2 occurs after J_RST_L deassertion, and OpenSPARC T2 will not be ready to receive a transaction until that initialization is completed. Thus, OpenSPARC T2 would not work in a system where, for example, Fire was initialized by a PCI-Express device, which then started a DMA transaction before OpenSPARC T2 was ready for it.

We plan to work within the reset timing specified in the JBUS spec. This means POWER_OK to POR deassertion of 5 msec, and POR deassertion to RST deassertion of 2 msec, for cold poweron; and 5 msec from RST assertion to deassertion for warm reset.

5.16.4 Glossary

ASI	Address Space Identifier
ASR	Ancillary State Register
MISR	Multiple-Input Signature Register. Used in LBIST
RClk	Regional Clock

5.16.5 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.”

5.16.6 Promotion among Core Available, Enable, and Status registers

Before the first instruction executes, TABLE 5-36 shows registers that must contain their correct values.

TABLE 5-36 Register Abbreviations

Register ASI name	Abbreviation in this appendix
ASI_CORE_AVAILABLE	CA
ASI_CORE_ENABLE	CE
ASI_CORE_ENABLE_STATUS	CES

TABLE 5-37 shows the sequence of events during Power-On Reset. The NCU accepts the initial, and only, value from the eFuse 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 5-37 Power-On Reset sequence of Events

	POR1	EFU1	NCU1	MBisi1	POR2	EFU2	NCU2	Soft ware	NCU pre-compu te
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 5-38 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 5-38 Warm reset Sequence of Events

	WMR1		NCU1	MBist2	WMR2		NCU2		

TABLE 5-38 Warm reset Sequence of Events

CA	CA								
CE	CE2								
CES	CESpre -> CES2		CE2 -> CES2	CES2	CES2		CE2 -> CES2		

CHAPTER **6**

Network Interface Unit (NIU)

This chapter contains the following sections:

- Meta Interface Microarchitecture Specification
- Interrupt Microarchitecture Specification
- Debug Microarchitecture Specification
- N2 NIU Design for Test
- SMX Microarchitecture

6.1 Introduction

The Networking Interface Unit (NIU) is an Ethernet to host bus/interface bridge. The design supports up to four Ethernet ports. The NIU is designed to provide scalable, high performance packet processing, optimized for Sun's throughput computing and networking architecture.

The primary goal of the NIU core is to provide a cost effective high performance interface with advanced network processing functions, such as packet classification for load balancing, checksum/CRC off loading and channelized and locatable DMA support. The NIU core is designed to run at frequencies up to 375 MHz @ 65nm , based on selected process technologies that were studied at the time of product definition. It includes an on chip 128 X 200bit TCAM, a 4K VLAN ID table shared by all ports, store-and-forward multiple max size packets per port in each RX and TX direction, a per port classification results FIFO, a per port TX Reorder FIFO, TID table, retry buffer and segmentation buffers etc. using on chip SRAMs. MAC Hash Table and DMA caches are implemented as registers due to their smaller size.

For network connectivity the, NIU includes up to two Ethernet ports, which are grouped as two Quad speed (10/100/1000/10000) Ethernet Ports (IEEE 802.3z,802.3ae). The Physical layer I/O interfaces provided are dual XAUI interfaces. Supported Port configurations include but not limited to dual 10/1 Gigabit Ethernet.

The NIU can connect to the system via various host busses using the patented on-chip 128bit wide, full duplex, system error aware, host agnostic Meta Interface bus

OpenSPARC T2 NIU supports the System Interface Unit (SIU).

6.1.1 Context for OpenSPARC T2

OpenSPARC T2 is Sun's third generation Chip Multi Threading (CMT) multi core CPU that will incorporate a dual 10/1 Gigabit version of the NIU core. Having direct network interfaces built in to the multi thread, multi core CPU will effectively double the network bandwidth of the T2 CPU, as well as reduce latency, greatly enhancing performance.

T2 platforms will provide PCI-E slots which will also provide early realization of the architecture.

6.1.2 Features and Requirements

1. Packet Processing
   - Supports IEEE 802.3/Ethernet packets at Layer 2, VLAN (802.1q) packets at Layer 2, LLC SNAP at Layer 2, AH/ESP for security, IPv4/IPv6 packets at Layer 3, TCP/UDP packets at Layer 4 etc.
   - Layer 1 – 4 Classification and flow identification
   - Internal 128 x 200 TCAM
   - True Store and Forward Architecture on RX and TX
   - Hardware checksum for Rx and Tx paths (TCP/IP)
   - Jumbo frame support (9216 bytes)
   - IP Multicast
2. Packet Movement
   - Support for 16 Receive and 16 Transmit DMA channels
   - Support for Transmit Gather of up to 15 descriptors
   - Jumbo frame support (9216 bytes)
3. System
   - Hypervisor virtualization and partitioning
   - T2 System Interface
   - Interface loop backs (internal and external)
   - Support UCB interface to NCU

6.1.3 Design Goals

1. Maximum aggregate throughput per direction (ingress, egress) of up to 20 Gbps for Network receive and 20 G (for jumbo packets at 9216B).
2. Store and Forward architecture for Rx and Tx paths.
3. Buffering for 9K bytes Jumbo packets in store and forward mode.
4. Buffering in 2 port active mode for latencies and delay in round robin arbitration and data movement into system memory.
5. Non blocking behavior for Posted Writes, Non Posted Writes and Read Requests.
6. Host bus MTU agnostic.
7. Thirty-two Non Posted Writes for updates of shadowed state for software coherency
8. Thirty-two split transaction for Read Request, with relaxed ordering rules.
9. Handle system level errors by encompassing them into split transaction time-outs, used by NPW and Read Request only.
10. Single code base with high level of reuse for different host connectivity.
11. Architecture and design independent of physical FIFO sizes.
12. Synchronous active low resets, with buffer tree optimization at layout.
13. No multi cycle paths for design blocks.
14. ECC and parity mechanism for protection of RAM data.
15. Little Endian design for control, data and software pios.

6.1.4 Buffering Analysis

NIU's buffering analysis assumes the following; 12 bytes Inter Packet Gap (IPG) and 100 ps/bit. Data FIFO size is normalized for 32K bytes (2048x128), which is 2048 (16 byte) entries. The control header for every packet is 16 bytes, and will be added for the analysis.

6.1.5 Single 10 Gigabit Port Active

64 bytes packet case: Data plus control header is 80 bytes, which uses five entries in the data FIFO. 2048/5 equals approximately 409 (64 byte packets) @ 67 ns per packet, which gives us an overflow time in the order of 27+ usec.

Jumbo packets case: 9500+ (9600) bytes used in data FIFO, worst case uses

600 entries. 2048/600 equals approximately three (9600 byte packets) @ 7.68 us per packets, which gives us an overflow time in the order of 23 usec.

Therefore for 32k on ingress buffering would give us 23+-27+ usec of overflow time per port.

6.1.6 All 10G and 1G Ports Active

The worst case for all ports active is the jumbo packet case, since we are a single threaded write architecture, we need to finish a complete write request before initiating the next write request.

For this case, the 1G ports will receive jumbo frames every 76.8 usec. We will assume that they were already there, when the 10G port completely receive their respective jumbo frames.

In addition, we will constraint the analysis to the host bus having the same bandwidth as the ingress port, and use a derating factor for speeds lower than that.

Port zero will get serviced every five jumbo frame the first time around, and then every third jumbo frame after that, then back to five and so on and so on.

This gives us a buffering time of 23 usec (3x7.68) to 38.4 usec (5x7.68) requirement. This buffering requirement decrease as the host bus speed increasesTherefore the buffering of 23-27 usec should be sufficient for T2 implementations.

6.1.6.1 Transaction Time-Outs and System Errors

The NIU core uses a transaction time-out mechanism for all system level errors that may be detected from the host bus. This enables the NIU core to validate its behavior independent of the host bus connectivity. This is important, since the semantics and error mechanism of the varied host buses can now be folded into a simple error recovery scheme.

The NIU architecture specifies that errors are grouped into following groups:

1. Detectable and Uncorrectable Error - Recoverable

2. Detectable and Uncorrectable Error - Fatal
3. Detectable and Correctable Errors - Recoverable

Each group mandates that the engines within the NIU core are non-blocking in nature. For example DMA 0 detects case 2, which causes DMA 0 to go dead. This should not affect any other DMA. However, in the event a Network Port goes dead, multiple DMA could be affected but not other ports.

Transaction Time-Outs

Transaction time-outs apply to Split Transaction Read Request that do not complete within a specified time interval. This time interval could be different on the multiple host buses that NIU could bolt to.

Since the NIU core uses "The Meta Interface" to communicate with the host buses, all host bus semantics are abstracted out. This enable the NIU core to use an abstracted semantics for movement of data from or too system memory. For example Read and Write Request of 4K Bytes.

To get into the specifics of transaction time-outs, we explore Split Transactions and Transaction IDs (TIDs).

The NIU core uses TIDs as handle(s) for Split Transaction Read Request and Non Posted Writes. These handles are unique per transaction and are recycled from a TID bin. The depth of the bin could vary between host buses.

In the event a split transaction does not complete within the transaction time-out period, that transaction is considered as expired or timed-out. This is indicated to the NIU core using the TID as the handle. The NIU core will take the appropriate action as to which engine was the originator of that transaction.

Since, we don't know whether we will receive any data for this expired or timed-out transaction. This transaction ID needs to be marked dirty and not used any more until told to do so by software. Further any data received for this transaction ID is to be dropped by the host bus logic, since there is no originator any more for this transaction.

The previous explanation covers at a high level the expected behavior of what the host bus logic would need to support for end to end behavior. However given the differences in host bus MTU and Read Request sizes. The host bus logic would need to take care of the multiple corner cases that could arise.

The following are some of the examples that could occur. Please note that these example are not meant to be taken as complete specification or as an implementation directive.

Case 1: Host bus and Meta Interface MTU and Read Request Size are the same.

In this example the TIDs on the Meta and the host bus could use a one-to-one mapping, with responses on the host bus cross referencing the same ID on the Meta side. Any time-outs detected, would result in the same TID on the host bus and Meta being marked dirty.

Case 2: Host bus and Meta Interface MTU and or Read Request sizes are different.

In this example, the TIDs on the Meta and the host bus cannot use a one-to-one mapping, but a one-to-many mapping (segmented reads), with responses on the host bus cross referencing the same ID on the Meta side. In the event any time-outs are detected, all of the many segmented mappings have to be marked as dirty on the host side, and the mapping removed for the Meta side.

Many examples of TIDs in flight and processing that would need to be quantified. Two case scenarios are presented.

All of the segmented reads have been dispatched prior to responses coming back.

Some of the segmented reads have been dispatched and responses have started coming back. Listing two of the possible cases.

1. Host system memory very fast.

2. No more credits for Split Transaction Requests.

In each case all the TIDs, dispatched or not, need to be marked dirty. This is very implementation specific, because of the decision regarding whether to even dispatch pending read segment requests if a time-out occurs.

In two cases time-out could occur when are receiving response:

- Some segment not received, since it dropped along the way.
- System level Error handling mechanism for NIU core. In the event a segment that fails CRC is received, that packet is dropped and then falls back to a transaction time-out behavior.

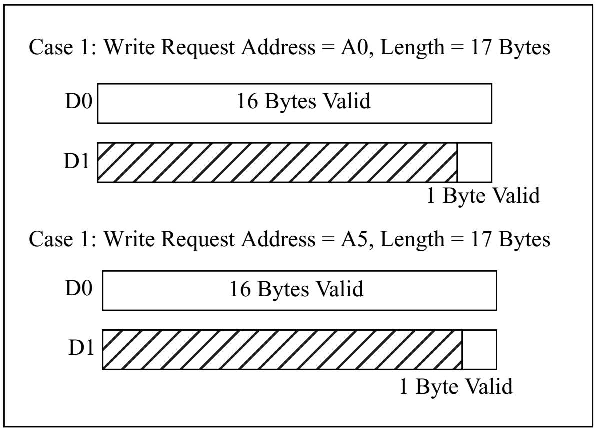
6.1.7 Data Alignment Format for Internal Datapath

FIGURE 6-1 and FIGURE 6-2 show the alignment of the internal data path of the NIU design for Request and Response data.

6.1.7.1 Request Data Format

In the case of Requests, data and address are byte aligned, with byte enables only on the last data phase. Where D0 and D1 are the two successive data phases, with D1 being the last data phase.

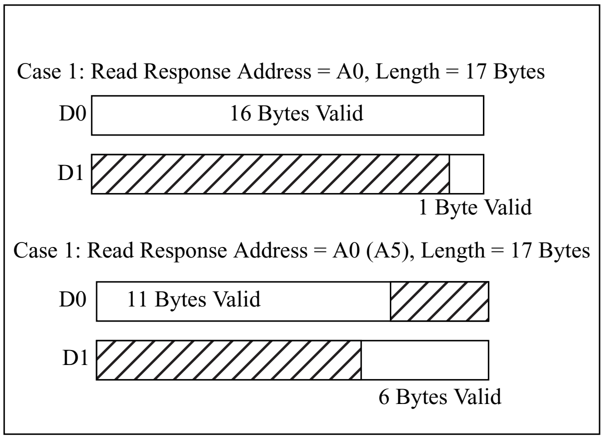
FIGURE 6-1 Request Data Format


Case 1: Write Request Address = A0, Length = 17 Bytes
D0
16 Bytes Valid
D1
1 Byte Valid
Case 1: Write Request Address = A5, Length = 17 Bytes
D0
16 Bytes Valid
D1
1 Byte Valid

6.1.7.2 Response Data Format

In the case of Response, data is 16 byte aligned with the appropriate byte enable driven, as shown in FIGURE 6-2. Where D0 and D1 are the two successive data phases.

FIGURE 6-2 Response Data Format


Case 1: Read Response Address = A0, Length = 17 Bytes
D0
16 Bytes Valid
D1
1 Byte Valid
Case 1: Read Response Address = A0 (A5), Length = 17 Bytes
D0
11 Bytes Valid
D1
6 Bytes Valid

6.2 Chip Overview

The NIU core abstracts out the properties associated with the physical host bus used (SIU) via Meta interface. The Meta Interface Specification details the microarchitecture, behavior and implementation.

The NIU architecture is designed with pipeline stages that are all interlocked, each of which meets the requirement for maximum performance as stated in the design goals section.

Clocking: The highest NIU clock frequency is 375MHz for OpenSPARC T2. The Ethernet sub system has multiple clock domains. For details see the MAS clock chapter. For reference some of the key frequencies are listed below:

XAUI Serdes: 156MHz (125MHz)/62.5MHz for 10G/1G

Glue: ~312MHz/125Mhz for 10G/1G (Rx and Tx clock domain)

MAC: XMAC: 2.5MHz/25MHz/125MHz/156MHz

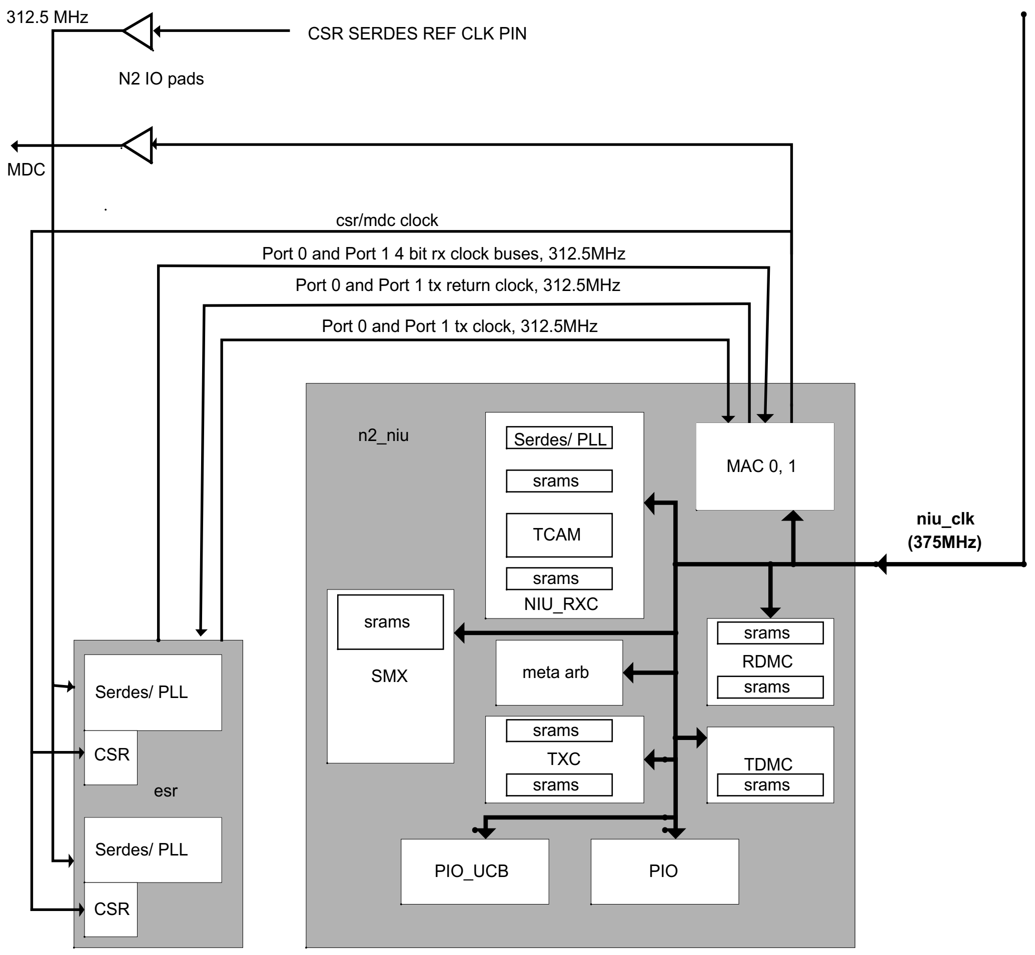
Receive & Transmit Datapath: 375MHz

TCAM: 375 MHz

Reset: Full chip reset is asynchronous upon assert and synchronous upon de-assert. IO flip flops are asynchronously resettable. Software reset is supported.

The system reset gets synchronized to each sub block's clock domain.

FIGURE 6-3 NIU Top Level Block Diagram



6.3 Configuration and Modes of Operation

Programming option for the XMAC will allow 1G operation also. Therefore a 10G, can be changed to 1G by software. The system is free to populate one or more ports at hardware or software level, which can be configured as 10G or 1G. TABLE 6-1 describes the various supported port configurations.

TABLE 6-1 NIU-OpenSPARC T2 Configurations

NIU-T2	Ethernet Ports	Remarks
X	2x10G	Fiber
X	2x1G	Copper
X	1x10G	Copper
X	1x10G and 1x1G	Fiber, Copper (Low Priority)
X	1x10G or 1x1G	Fiber, Copper (Low Priority)
	1 10G	Copper
	2x10G	Fiber

6.4 Effective Performance Targets for various Host Bus variants of NIU

The performance target is for one direction only with numbers listed for the transmit path. The bottle neck is governed by host bus performance.

TABLE 6-2 NIU-OpenSPARC T2 Tx Performance

Packet Size Back to Back	NIU-T2	Remarks
64B	15M	OpenSPARC T2: NIU is the bottle neck currently
512		OpenSPARC T2: Networks is the bottleneck
1518	823K	OpenSPARC T2: Networks is the bottleneck.

6.5 Theory of Operation

MACs are the Ethernet Media Access Controllers that support the Ethernet protocol. They contain the layer 2 protocol logic, statistic counters, address matching and filtering logic. The output from the MACs contain information on the destination address, whether it is one of the programmed individual addresses or an accepted group address, and the index associated with the address in that category.

Frames from different physical ports are stored temporarily in a per port receive FIFO. While they are being stored into the FIFO, the frame or first 128B will also be copied to the packet classification and checksum engines. The classification logic will determine which Receive DMA Channel (RDC) Group the packet belongs to and an offset into the RDC Table where the final RDC is determined. There are a total of eight RDC Tables.

The Layer 2 parser processes the Ethernet header to determine if the received frame contains a VLAN Tag or LLC/SNAP header. For VLAN tagged packet, the VLAN ID is used to lookup into a VLAN table to determine the RDC Table number for the packet. Hardware will also lookup the MAC address table to determine a RDC Table number based on the destination MAC address information. Software can program which of the two groups to use in subsequent classification. The output of the Layer 2 parser together with the resulting RDC Table number will be passed to the Layer 3/4 parser.

The Layer 2/3/4 parser will examine the EtherType, TOS/DSCP field and the Protocol ID/Next Header field to determine if the IP packet needs further classification. The L3/4 parser is hardwired to recognize some fixed protocol such as TCP or UDP. It also supports a number of programmable Protocol IP number. If the packet needs further classification, it will generate a Flow Key and a TCAM Key.

The TCAM key is sent to the TCAM unit for an associative search. If there is a match, the result may override the RDC Table selection from L2 and/or contain an offset into the Layer 2 RDC Table and ignore the result from the Hash Unit..

The TCAM result will determine if a hash lookup based on the Flow Key is needed. Using the RDC Table number supplied by the TCAM logic, which determines a partition of the external table the Hash unit can search, an lookup is launched and either an exact match or an optimistic match is performed. If there is a match, the result contains the offset into the RDC Table and the User Data.

The output from the Hash unit and the TCAM unit will be merged, and used to lookup in the RDC Table to determine a RDC to enqueue the received frame. The output of the Classification Unit is stored into the Control FIFO.

Hardware supports checksum off load and CRC-32c off load for TCP/SCTP payloads.[1] Hardware will simply compare the calculated value with the CRC value. The result will be sent to software through the completion status. No discard decision is made based on the CRC result. Note that checksum/CRC errors do not affect the L3/4 classification results. Similarly, the error status will be sent to software through the completion status.

The datapath from the two MACs is time shared. The Receive FIFO is logically organized per physical port. Layer 2/3/4 error information has to be logically synchronized with the classification result of the corresponding frame.

Logically there are 16 Receive DMA Channels. The datapath engine is common across the channels. It is also used to prefetch the Receive Blocks or update the Completion Ring of the RDCs.

Each Receive DMA Channel (RDC) has a Receive Buffer Ring (RBR), a Receive Completion Ring (RCR) and the state associated with the RDC. Physically, they are allocated as ring buffers in system memory. To support partitioning, each RDC supports two logical pages. All the addresses posted by software, such as the configuration of the ring buffers, buffer block addresses, are translated to physical addresses when used to reference system memory.

Software posts buffer Blocks into the RBR. The size of each block is programmable, but fixed per channel. Software can specify up to three sizes of packet buffer hardware can partition a block. Each block can only contain packet buffers of the same size.

To reduce the per packet overhead, hardware maintains a pre fetch buffer for the RBR and a tail buffer for the RCR. When the RBR prefect is low, a request will be issued to the system memory to retrieve a cache of Block addresses from the ring. The prefect requests may be issued as bypass queue read requests.[2] Or if the RCR tail buffer needs to be updated, a non posted writes request will be issued. The RCR non posted write request will be issued as an ordered queue posted write request for host buses that do not support non posted writes. When a completion acknowledgments from system is returned, the software visible states will be updated Since there is no completion acknowledgment for posted writes, the host bus interface logic must return and acknowledgment when the posted write has been dispatched to the ordered domain, where no ordered transaction can bypass this write. The RDC control scheduler will maintain the fairness among the RDCs.

The Port Scheduler examines if there are any frames available from the Receive FIFO and the Control FIFO, and decides which port to service first. A Deficit Round Robin scheduler is implemented. From the control header, the scheduler determines which RDC to check for congestion and retrieve a buffer to store the frame. Congestion is determined by a WRED algorithm applied on the Receive Buffer Ring and the

1. For TCP over Ethernet packets, the last four bytes of the packet is assumed to be a CRC value.

2. This is safe if there is only one outstanding request.

Receive Completion Ring. If the RDC is not congested, a buffer address is allocated according to the packet size. Packet data requests are issued as posted writes to the bypass/relaxed order queue.

The RCR buffer will be updated after issuing the write requests for the entire packet. However, software visible states will not be updated at this time. When updating the RCR buffer to system memory, an ordered write request is issued. When the acknowledgment is returned, the software visible states will be updated.

The datapath engine will fairly schedule the requests from the Port Scheduler and the RDC Control Scheduler and issue the requests to the DRAM.

The DMA status registers will be updated every time the RCR buffer is updated. Software may poll the DMA status registers to determine if any packet has been received. When the RCR queue length reached a threshold or a time-out occurred since the first arrival to an empty RCR, hardware may update the RCR buffer and at the same time, write the DMA status registers to a software defined mailbox. These writes are issued as ordered writes. When the acknowledgments are returned, the software state will be updated, and an interrupt may be issued to the NCU directly. Note that the CSR mailbox update and the interrupt can be enabled independently.

On the transmit side, there are 16 Transmit DMA Channels total. The following Figure shows the logical view of the transmit hardware. Each channel is comprised of a Transmit Ring, a set of control and status registers. Similar to the receive side, each transmit channel supports two logical pages, different from the RX page. Addresses in the Transmit Ring and configuration registers are subjected a translation to convert to physical addresses.

The Transmit Ring is built from a ring buffer in system memory. Software posts packets into the Transmit Ring, and signals the DMA hardware that packets have been queued. Each packet is built as a gather list. When the Transmit Ring is not empty, hardware will prefetch the Transmit Ring into a per channel buffer.

Any DMA channel can be bound to one of Ethernet ports (2 in OPenSPARC) by software. This is controlled by a mapping register at the per port DRR scheduler. The DRR scheduler may switch to a different channel on packet boundary. This guarantees there will be no packet interleaving from different DMA channels. The scheduler will first acquire an available buffer tag for that port. If it is available, a memory request will be issued. The buffer tag is needed because acknowledgment (with the packet data) may return out of order. This tag, which is linked to the request/ack ID, is used to reorder the data from memory system into the packet order.

The Ethernet ports are serviced in round robin order, and requests from different ports may be interleaved.

6.6 Receive Datapath

The receive data path is responsible for providing advanced network processing functions such as packet classification, checksum off loading, and channelized DMA for movement of ingress network traffic to system memory. The receive datapath is partitioned into two main sub-modules, Receive Controller (RXC) and Receive Data Management Controller (RDMC), and includes all the functionality for moving data from the Ethernet ports to system memory.

The RXC sub-module is partitioned into two engines, Data and Control/Classification. The data partition is responsible for all aspects of packet checking, while the control/classification selects the DMA resource responsible for packet transfer to system memory.

The motivation to separate the data and classification paths, is to abstract the behavior and latency of the classification engine, and for supporting line rate traffic at 20 G bits/sec.

The Data partition comprises of the following sub-modules, Input Packet Processor (IPP) with Header Parser interface, Checksum/CRC engine and data FIFO.

6.7 Input Packet Processor (IPP)

The IPP interfaces to the 10 G and 1G MACs via 64 bit data bus, and converts 64 bit ingress data to an internal 128 bit data path. Following the conversion, the data is moved into a pre-buffer for IPP processing. The IPP waits till the pre-buffer indicates that either of the two conditions are valid:

1. Complete Packet
2. 128 Bytes of data stored

When either condition is valid, the IPP extracts the data and informs the Header parser that there is a valid header for this port. Since the header parser is shared between all ports (pure round robin), the pre-buffer is sized (1kbytes), to support the worst case arbitration delay.

When the header parser indicates that it is ready to process the header. The IPP forwards a copy of the data to the header parser, and waits for checksum/crc calculation information. The header parser is required to respond to the IPP within 4 system cycles with the information needed to compute/verify checksum/crc.

The IPP and checksum/crc engines provide support for the following:

1. TCP and UDP full checksum
2. CRC32 on TCP, UDP and SCTP

The IPP now starts moving the ingress packet from the pre-buffer via the checksum/crc engine, to the main data fifo. The IPP encapsulates the packets in the data fifo with SOP with control header(128 bits) and EOP tags. The control header consist of packet length (calculated by the IPP) MAC crc status and checksum/sctp status. This information is used by the RDMC.

The IPP is responsible for padding MAC data when the conversion of 64bit to 128bit datapath has a partial result from odd packet sizes. For example:

a. if a MAC packet data is an odd set of 64-bits

   XMAC: {41'h0,mac_status[22:0],mac_data[63:0]}

b. if a MAC packet data is an even set of 64-bits

   XMAC: {41'h0,mac_status[22:0],41'h0,mac_status[22:0]}

Runt packets smaller than 64 bytes are silently dropped and recorded in IPP status registers.In addition the MAC will also record dropped and runt packets.

The Data Fifo to RDMC interface is implemented using a simple request-ack protocol (dmc_request, ipp_ack). For each dmc_request (read operation) there is an associated ipp_ack (data Valid). The “ipp_full_pkt” flag signals the RDMC, that at least one complete packet is in the FIFO. The RDMC asserts dmc_request signal for reading data out of the Data FIFO. Data returned by the read operation is qualified by a data valid (ipp_ack).

The Checksum unit supports 1's complement 128 bit full checksum and CRC_32c. The overhead of the checksum engine is 4 cycles.

6.8 Header Parser and Classification Engine (FFLP)

The Header Parser is responsible for extracting header information from the ingress packet. It provides the IPP with offset information necessary to compute TCP/IP checksum. The Header Parser also creates the TCAM search keys for flow identification.

The Classification Engine is responsible for VLAN lookup’s, TCAM searches and updates, computation of Hash function for DMA RDC table offset calculation etc.

This stage is grouped as one of the logical pipeline stages and must perform its operations to meet 20 Gbps line rate performance.

6.9 Receive DMA Engine

The Receive I/F connects to both the Rx Packet FIFO and the Rx Control FIFO. Based on Rx Control FIFO parsing results, the DMU receive path is responsible for transferring packets between the Rx Receive FIFO and the Meta Interface.

The Port Scheduler determines the next port from which to transfer data based on the status of the Rx Packet FIFO, the Rx Control FIFO, a Deficit Round Robin algorithm is used to determine the availability of the selected DMA channel.

Congestion is determined by a WRED algorithm applied on the Receive FIFO and the Receive Completion Ring. When a channel is determined to be congested, the Port Scheduler will drop the packets randomly from the existing queues. In the TCP flags are available, preference may be given to existing connections. The preference is to drop randomly from the existing queue rather than packets from new connections.

6.10 Transmit Datapath

The transmit data path is responsible for providing checksum computation, DMA support for moving egress network traffic from system memory to the Ethernet Ports. The transmit datapath is partitioned into two main sub-modules, and Transmit Data Management Controller (TDMC) and Transmit Controller (TXC), and includes all the functionality for supporting a gather list of up to 15 descriptors.

6.11 Transmit DMA Controller

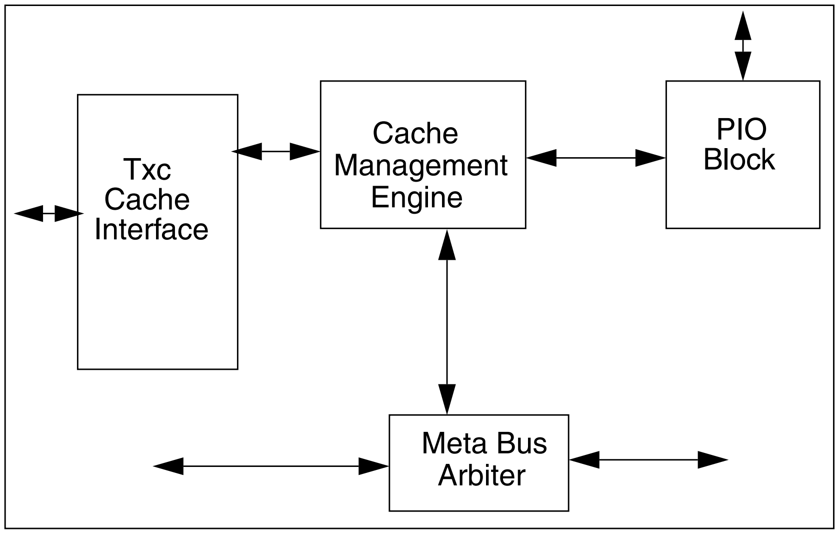
This block is the interface between the Transmit Control Engine (TXC) and Meta Bus Interface (which interfaces to the system memory).This block is responsible for managing the Transmit Descriptors. It also has the mechanism for caching descriptors and providing these to TXC upon request.

The TDMC is partitioned into four sub-blocks as follows:

1. PIO Block.
2. Cache Fetch and Management Engine.
3. Transmit Controller Interface block.
4. Arbiter for Meta Bus (Shared between Rx and Tx DMA Engines)

The structure of various sub-blocks is shown in FIGURE 6-4.

FIGURE 6-4 TXDMA Sub Blocks



6.11.1 PIO Block

This block interfaces with NIU's PIO control block. It manages all the Control and Status Registers (CSRs) required for correct functioning of the DMU block. Some of the registers, added for ease of diagnosis, are also managed within this block.

6.11.2 Cache Management Engine

The cache fetch engine is responsible for scheduling descriptor fetch requests to the System Memory and to cache descriptors in its cache memory. There are up to 16 DMA channels supported in NIU. Per DMA Channel, the cache engine can fetch up to a maximum of two cache-line worth of descriptors and store these in the cache. The cache line is assumed to be 64 Bytes long.

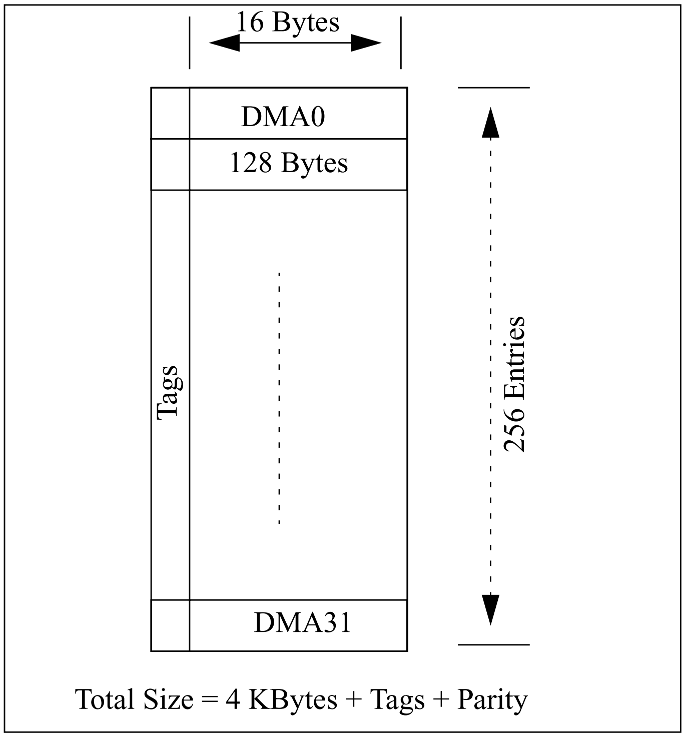
The descriptors held in the cache are processed by the TXC and hence the read pointers are controlled by the TXC Block.

6.11.3 Tx DMA Cache RAM

The cache is organized as 16 logical FIFOs, one for each DMA channel. Each entry in this cache stores two descriptors (8 Bytes each). Up to 16 descriptors (128 Bytes – Two 64 Byte Cache Lines) can be stored per DMA Channel. There is a set of tags associated with each entry. This is used by the Exacted Interface to keep track of any gather list.

Although the worst case latency for read request is system dependent, NIU expects the worst case latency to be less than 800ns calculated from the acceptance of the read request by the host bus logic. FIGURE 6-5 illustrates the image of the descriptor cache.

FIGURE 6-5 Diagram Of Descriptor Cache



6.11.4 Tx DMA Cache Fetch Engine

The cache fetch engine manages the heuristics associated with cache fetches from system memory. A fetch request to the memory can be initiated by any of the following conditions.

1. Receipt of a new kick for any DMA channel. This channel gets added into a list of active DMAs. This condition is detected when software clears the stall bit in the CSR.

2. Availability of at least one cache line worth of space in the cache of any active DMA. This can happen as TXC keeps servicing the descriptors. The fetch engine always tries to fill up the cache so that the descriptors are available when requested by TXC.

3. Appending of descriptors to an active DMA by the software. This happens when software moves the tail pointer in the CSR. A fetch request for that DMA will be scheduled under the following conditions.

4. The Descriptor Cache is empty (For this DMA Channel).

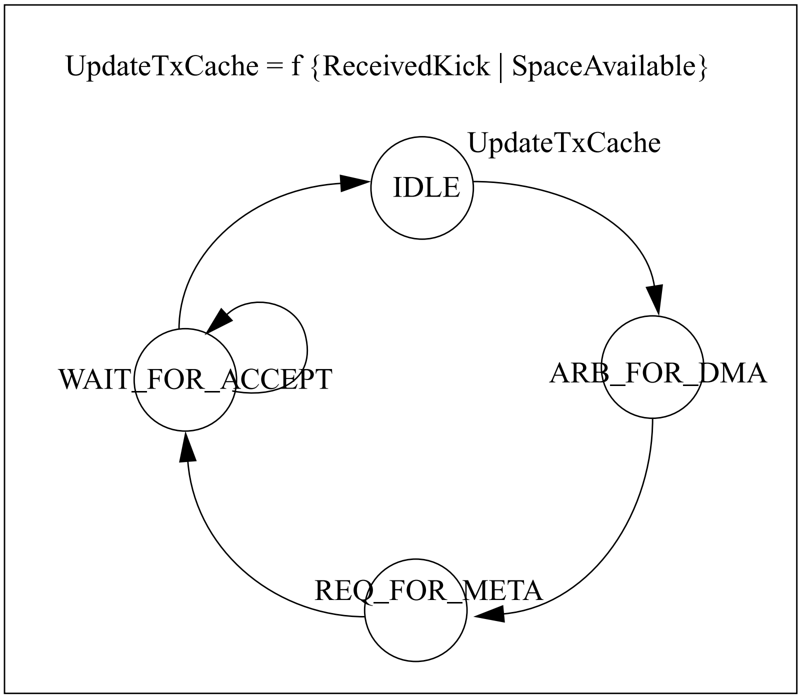
5. The Descriptor Cache is partially full and has space available for at least one cache line.

6. If the Cache is full, appending a new set of descriptors should not initiate any new descriptor fetches until enough space is available.

The extent of fullness of the cache is determined using a set of shadow pointers. These pointers get updated as soon as a request is issued to the system memory. These pointers are used as an anchor to determine the state of outstanding requests to System Memory on a per DMA channel basis.

Once the request state machine s triggered to fetch descriptors, it arbitrates among all the available DMAs. An arbitration algorithm is used to choose a DMA from the list of available DMAs in a round robin fashion. The algorithm also checks for space availability in the cache of each DMA and sends read requests to system memory accordingly. A logical view of the request state machine is shown in FIGURE 6-6.

FIGURE 6-6 Request State Machine



6.11.5 Tx DMA Cache Write Engine

The Cache write engine is responsible for writing the cache response data back into the SRAM. It writes the data into the appropriate logical offset using the DMA number returned by the Meta Interface. This engine reorders any out of order responses from memory and updates the appropriate pointers, which are used by the TxCache interface block.

6.12 Tx Controller Interface

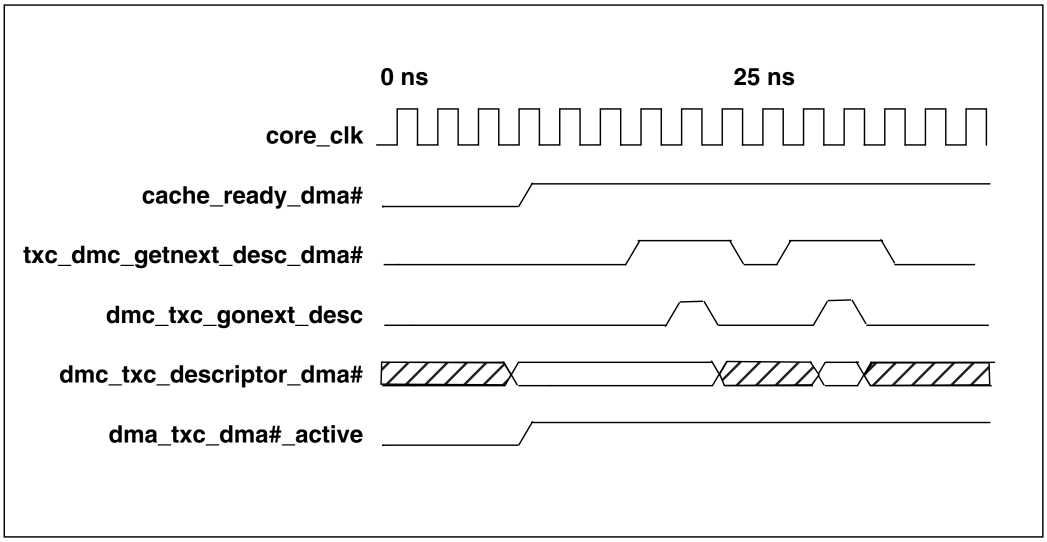
This block reads the cache and presents the descriptors to TXC. As soon as the descriptors are ready and read out from the cache FIFO, it send a cache_ready signal to the TXC. This causes the TXC state machines to be kick started.

Each popped entry has two flags associated with it. These are used to indicate if the corresponding entry is valid or not. The validity of an entry is determined by the following conditions:

1. Comparison of the current head and tail pointers of the descriptor ring to determine if the end of the list has been reached.
2. If the entry is the first descriptor of a gather list and at least one descriptor of that gather list is unavailable.
3. Any errors associated with the data integrity.

Based upon these conditions, only valid entries are sent to the TXC block for further processing. The timing diagram for the interface between TXC and Tx DMA block is shown in FIGURE 6-7.

FIGURE 6-7 TXDMA To TXC Interface Timing Diagram



6.13 Transmit Controller

The Transmit Controller Engine (TXE), FIGURE 6-8, consist of the following functional blocks.; The DRR Scheduler, Data Fetch State Machine and per port re-order/realigner buffer. The engine is designed as a true store and forward, such that latencies on the host bus are abstracted from any temporal issues during transmit. The Scheduler and Data State machine are shared by all the ports in the Tx path.

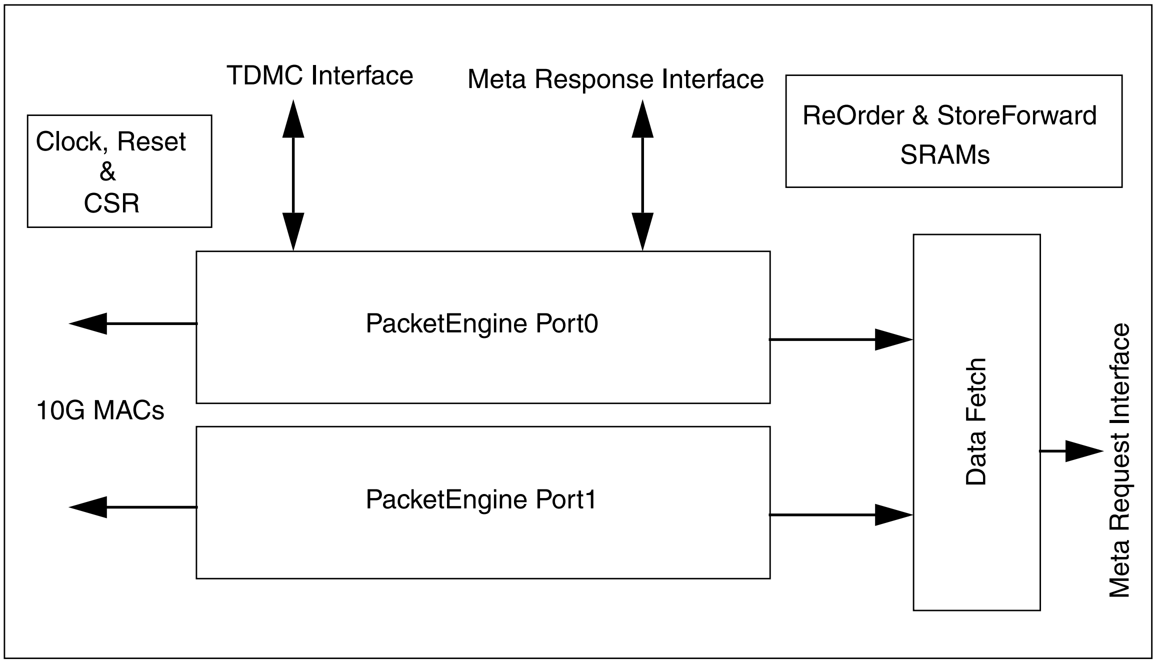
Software posts transmit packets into a transmit DMA channel where each packet may be made up of a gather list. The DMA controller fetches the descriptor and informs the DRR scheduler of active DMA channels and their corresponding descriptor information. The Scheduler coupled with the DRR information and re-order state, dispatches data request to the Data fetch state machine. This engine is now responsible for fetch and re-ordering data from the host system and storing the response data with the appropriate alignment information in the re-order buffer. Upon completion of the request the state machine causes the re-aligner to process the stored data in the buffer. The Realigner reads data from the buffer, aligns the data into16 byte-aligned chunks and pushes the data into the Store and Forward FIFO, while at the same time primes the checksum engine if enabled. On the completion of the packet the re-aligner updates the checksum information and forwards the packet for transmission. The data state machine interface with the DMA controller for update to the completion ring and next descriptor fetches.

The Transmit Controller Engine (TXE) interfaces to host system through the DMA Controller & Cache via the Meta interface. However, the use of the Meta interface abstracts out the physical MTU of the host bus.

The Transmit data requests and the prefetch requests share the same datapath to memory system. The returned acknowledgment is first processed to decide whether it is a prefetch or a Transmit data. Transmit hardware also supports checksum off load and CRC-32C off load. This logic is embedded in the Reorder and Transmit FIFO logic.

When the entire packet has been received into the Transmit FIFO, the transmission of the packet is considered to be completed and the state of the DMA channel will be updated through the associated status register. A 12-bit wrap around counter, initialized to 0, is used to keep track of packets transmitted. Software may poll the status registers to determine the status. Alternatively, software may MARK a packet so that an interrupt (if enabled) may be issued after the transmission of the packet. Similar to the receive side, hardware will update the state of the DMA channel to a predefined mailbox before issuing an interrupt.

FIGURE 6-8 TX Controller Block Diagram

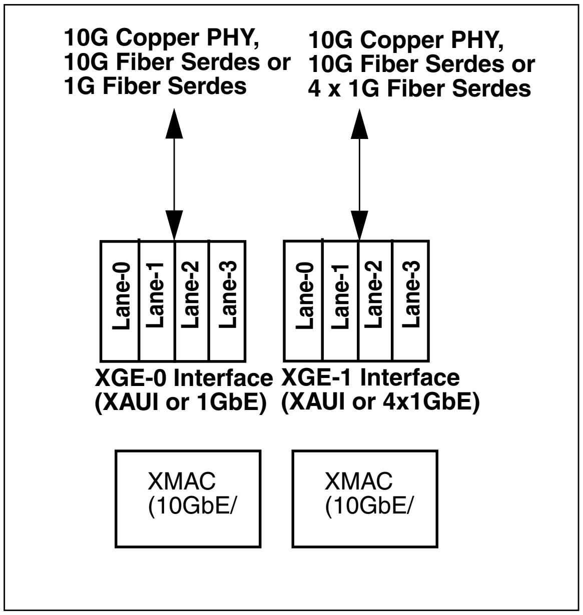


6.14 Ethernet MicroArchitecture Specification (MAC,MIF)

6.14.1 MAC Network Connections

6.14.1.1 T2 Network Interface Connections

FIGURE 6-9 Network Interface Connections



6.14.2 Ethernet Port Configuration Table

TABLE 6-3 T2 MAC Port Configuration

configuration	port0	port1	mode setting port0	mode setting port1

2x10G	10G (XAUI)	10G (XAUI)	xgmii_mode0 = 1
			gmii_mode0 = 0
			mii_mode0 = 0
			pcs_bypass0 = 0
			(XAUI)
				xgmii_mode1 = 1
				gmii_mode1 = 0
				mii_mode1 = 0
				pcs_bypass1 = 0
				(XAUI)

1x10G + 1x1G	10G (XAUI)	1G (ch0) (fibre)	xgmii_mode0 = 1
			gmii_mode0 = 0
			mii_mode0 = 0
			pcs_bypass0 = 0
			(fibre)
				xgmii_mode1 = 0
				gmii_mode1 = 1
				mii_mode1 = 0
				pcs_bypass1 = 0
				(fibre)

2x1G	1G (fibre)	1G (ch0) (fibre)	xgmii_mode0 = 0
			gmii_mode0 = 1
			mii_mode0 = 0
			pcs_bypass0 = 0
			(fibre)
				xgmii_mode1 = 0
				gmii_mode1 = 1
				mii_mode1 = 0
				pcs_bypass1 = 0
				(fibre)

6.14.3 Ethernet Port Loopback Mode Configuration Table

6.14.4 T2 MAC Loopback Mode

The corresponding T2 MAC port configuration table, TABLE 6-4, should be setup to do the loopback static timing analysis.

TABLE 6-4 T2 MAC Port Configuration - Loopback Analysis

configuration	port0	port1	mode setting port0	mode setting port1

2x10G	10G (XAUI)	10G (XAUI)	loopback0 = 1
			xpcs_loopback0 = 0
				loopback1 = 1
				xpcs_loopback1 = 0

XMAC loopback				

2x10G	10G (XAUI)	10G (XAUI)	loopback0 = 0
			xpcs_loopback0 = 1
				loopback1 = 0
				xpcs_loopback1 = 1

XPCS loopback				

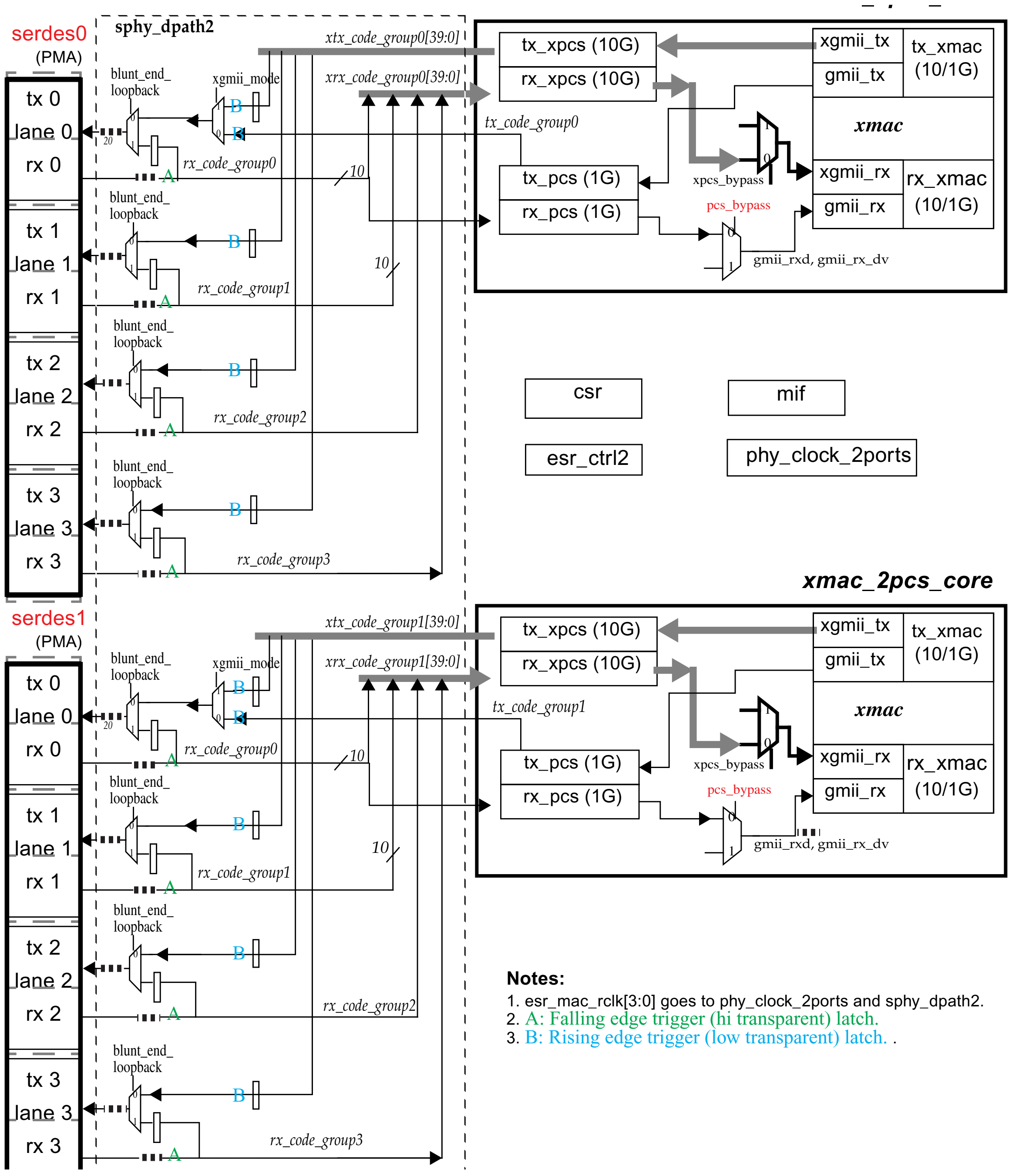
10G	10G (XAUI)	10G (XAUI)	loopback0 = 0
			xpcs_loopback0 = 0
				loopback1 = 0
				xpcs_loopback1 = 0

Blunt Loopback				blunt_end_loopback = 1

1G	1G (fibre)	10 (XAUI)	loopback0 = 0
			xpcs_loopback0 = 0
				loopback1 = 0
				xpcs_loopback1 = 0

Blunt Loopback				blunt_end_loopback = 1

FIGURE 6-10 T2 MAC Top Level Architecture



Notes:

1. esr_mac_rclk[3:0] goes to phy_clock_2ports and sphy_dpath2.
2. A: Falling edge trigger (hi transparent) latch.
3. B: Rising edge trigger (low transparent) latch. .

6.14.5 Serdes - MAC Interface Signals

TABLE 6-5 MAC Interrupts Info Table

Signal Name	Width	Description
from serdes to mac		
esr_mac_los_0	[3:0]	Port 0 serdes loss of signal from each lane
esr_mac_los_1	[3:0]	Port 1 serdes loss of signal from each lane
esr_mac_oddcg0_0		Port 0, lane 0 odd code group
esr_mac_oddcg0_1		Port 1, lane 0 odd code group
esr_mac_oddcg1_1		Port 1, lane 1 odd code group
esr_mac_oddcg2_1		Port 1, lane 2 odd code group
esr_mac_oddcg3_1		Port 1, lane 3 odd code group
esr_mac_rclk_0	[3:0]	Port 0 per lane receive clock
esr_mac_rclk_1	[3:0]	Port 1 per lane receive clock
esr_mac_rxd0_0	[9:0]	Port 0 lane 0 receive data
esr_mac_rxd1_0	[9:0]	Port 0 lane 1 receive data
esr_mac_rxd2_0	[9:0]	Port 0 lane 2 receive data
esr_mac_rxd3_0	[9:0]	Port 0 lane 3 receive data
esr_mac_rxd0_1	[9:0]	Port 1 lane 0 receive data
esr_mac_rxd1_1	[9:0]	Port 1 lane 1 receive data
esr_mac_rxd2_1	[9:0]	Port 1 lane 2 receive data
esr_mac_rxd3_1	[9:0]	Port 1 lane 3 receive data
esr_mac_serdes_rdy_0	[3:0]	Port 0 serdes lanes have detected and sync up with comma character (K28.5) and ready to receive packet.
esr_mac_serdes_rdy_1	[3:0]	Port 1 serdes lanes have detected and sync up with comma character (K28.5) and ready to receive packet.
esr_mac_tclk_0		Port 0 lane 0 transmit clock; output of tx PLL
esr_mac_tclk_1	[3:0]	Port 1 lane 0,1,2,3 transmit clock; output of tx PLL
from mac to serdes		
mac_esr_txd0_0	[9:0]	Port 0 lane 0 transmit data
mac_esr_txd1_0	[9:0]	Port 0 lane 1 transmit data
mac_esr_txd2_0	[9:0]	Port 0 lane 2 transmit data
mac_esr_txd3_0	[9:0]	Port 0 lane 3 transmit data

TABLE 6-5 MAC Interrupts Info Table *(Continued)*

Signal Name	Width	Description
mac_esr_txd0_1	[9:0]	Port 1 lane 0 transmit data
mac_esr_txd1_1	[9:0]	Port 1 lane 1 transmit data
mac_esr_txd2_1	[9:0]	Port 1 lane 2 transmit data
mac_esr_txd3_1	[9:0]	Port 1 lane 3 transmit data

Note – oddcg: odd code group; per IEEE 802.3z clause 36.2.4.2; Used to identify comma code group (K28.5) which should reside in even code group;

6.15 NIU_RXC_TOP Microarchitecture Specification

6.15.1 NIU_RXC_TOP Overview

NIU_RXC consists of three sub-modules:

1. IPP: Input Packet Process unit,
2. FFLP: Flow Forwarding and Learning Process unit,
3. ZCP: Zero Copy Process unit.

IPP takes packet data from MAC, sends packet's header to FFLP;

FFLP sends packet's checksum information back to IPP;

IPP then puts the packets into the ipp_data_fifo with checksum results for RDMC to transfer MAC packets into system memory.

FFLP parses headers of the packets for IPP's checksum and ZCP's packet control information buffering through TCAM key search. FFLP provides ZCP packet's Receive DMA channel Table number and the table offset number of that selected table.

ZCP gathers packet's characteristics, the RDC Table number, and the table offset number provided by FFLP's header parsing;

ZCP then provides RDMC the RDMA channel number, and the system memory address of a packet to be transferred to, via the zcp_ctrl_fifo.

NIU_RXC, and the interfacing portions of the MAC, and RDMC are running under the same core_clock.

Based upon system's configuration,

the core_clock is running up to 375MHZ(OpenSPARC T2 mode).

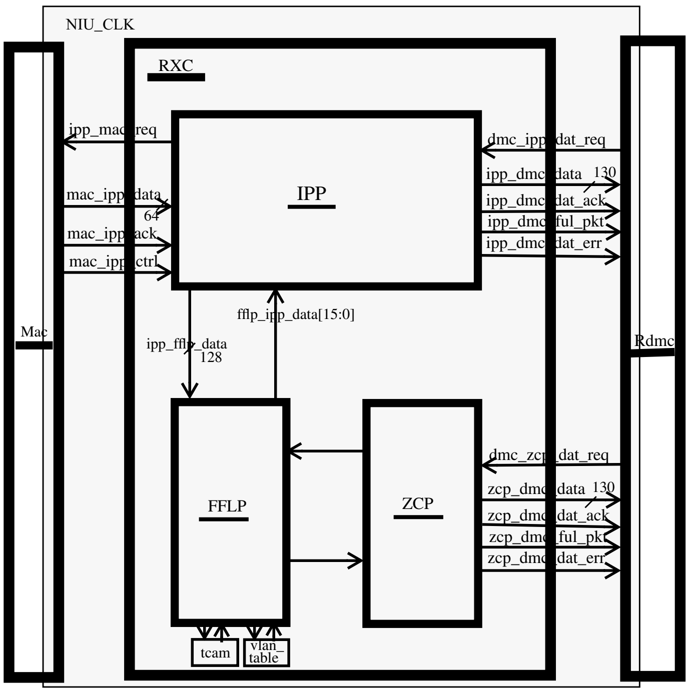
Depending upon system's configuration,

NIU_RXC supports up to 20Gbits/sec receiving data rate.

An on-chip ternary TCAM is required.

A VLAN_table is used. It has 4096 entries, covering all 12 bits of the vlan_tags.

FIGURE 6-11 NIIU-RXC Block Diagram



Abreviations:
RXC: Receive Transfer Control unit
IPP: Input Packet Process unit
ZCP: Zero Copy Process unit
FFLP: Flow Forwarding Learning Process unit

6.15.1 NIU_RXC_TOP Interface Signals

TABLE 6-6 NIU_RXC_TOP Top Level Interface Signals

Signal Name	Width	Direction	Description
MAC Interface			
mac_ipp_ack0	1	I	Xmac0 sends the ack to ipp0
mac_ipp_tag0	1	I	Xmac0 identifies the end of a packet
mac_ipp_data0	64	I	Xmac0 writing the packet's data to ipp0
mac_ipp_ctrl0	1	I	Active high for packet's control information
mac_ipp_stat0	23	I	Xmac0 writing the packet's status to ipp0
ipp_mac_req0	1	O	Request (as rdy) from ipp0 to xmac0
mac_ipp_ack1	1	I	Xmac1 sends the ack to ipp1
mac_ipp_tag1	1	I	Xmac1 identifies the end of a packet
mac_ipp_data1	64	I	Xmac1 writing the packet's data to ipp1
mac_ipp_ctrl1	1	I	Active high for packet's control information
mac_ipp_stat1	23	I	Xmac1 writing the packet's status to ipp1
ipp_mac_req1	1	O	Request (as rdy) from ipp1 to xmac1
ipp_rdmc interface			
dmc_ipp_dat_req0	1	I	Rdmc requests data from ipp_dat_fifo_0
ipp_dmc_dat_ack0	1	O	If 1, validates ipp_dat_fifo_0 data to rdmc
ipp_dmc_data0	130	O	ipp_dat_fifo_0's data to rdmc
ipp_dmc_ful_pkt0	1	O	If 1, ipp_dat_fifo_0 has complete packet/s
ipp_dmc_dat_err0	1	O	If 1, ipp_dat_fifo_0 data has error
dmc_ipp_dat_req1	1	I	Rdmc requests data from ipp_dat_fifo_1
ipp_dmc_dat_ack1	1	O	If 1, validates ipp_dat_fifo_1 data to rdmc
ipp_dmc_data1	130	O	ipp_dat_fifo_1's data to rdmc
ipp_dmc_ful_pkt1	1	O	If 1, ipp_dat_fifo_1 has complete packet/s
ipp_dmc_dat_err1	1	O	If 1, ipp_dat_fifo_1 data has error
ipp_rdmc interface			
dmc_ipp_dat_req0	1	I	Rdmc requests data from ipp_dat_fifo_0

TABLE 6-6 NIU_RXC_TOP Top Level Interface Signals *(Continued)*

Signal Name	Width	Direction	Description
ipp_dmc_dat_ack0	1	O	If 1, validates ipp_dat_fifo_0 data to rdmc
ipp_dmc_data0	130	O	ipp_dat_fifo_0's data to rdmc
ipp_dmc_ful_pkt0	1	O	If 1, ipp_dat_fifo_0 has complete packet/s
ipp_dmc_dat_err0	1	O	If 1, ipp_dat_fifo_0 data has error
dmc_ipp_dat_req1	1	I	Rdmc requests data from ipp_dat_fifo_1
ipp_dmc_dat_ack1	1	O	If 1, validates ipp_dat_fifo_1 data to rdmc
ipp_dmc_data1	130	O	ipp_dat_fifo_1's data to rdmc
ipp_dmc_ful_pkt1	1	O	If 1, ipp_dat_fifo_1 has complete packet/s
ipp_dmc_dat_err1	1	O	If 1, ipp_dat_fifo_1 data has error
zcp_rdmc interface			
dmc_zcp_req0	1	I	Rdmc requests data from zcp_ctrl_fifo_0
zcp_dmc_ack0	1	O	if 1, zcp_ctrl_fifo_0 sending valid data to rdmc
zcp_dmc_dat0	130	O	zcp_ctrl_fifo_0's data sent to rdmc
zcp_dmc_ful_pkt0	1	O	f 1, zcp_ctrlfifo_0 has complete packet/s
zcp_dmc_dat_err0	1	O	f 1, zcp_ctrl_fifo_0 data has error
dmc_zcp_req1	1	I	Rdmc requests data from zcp_ctrl_fifo_1
zcp_dmc_ack1	1	O	if 1, zcp_ctrl_fifo_1 sending valid data to rdmc
zcp_dmc_dat1	130	O	zcp_ctrl_fifo_1's data sent to rdmc
zcp_dmc_ful_pkt1	1	O	f 1, zcp_ctrlfifo_1 has complete packet/s
zcp_dmc_dat_err1	1	O	f 1, zcp_ctrl_fifo_1 data has error
global signals			
niu_reset_l	1	I	core reset, active low
niu_clk	1	I	core_clock
pio_ipp_sel	1	I	if 1, pio access ipp
pio_zcp_sel	1	I	if 1, pio access zcp
pio_fflp_sel	1	I	if 1, pio access fflp
pio_clients_addr	20	I	pio access address
pio_clients_rd	1	I	if 1, pio read, else pio write

TABLE 6-6 NIU_RXC_TOP Top Level Interface Signals *(Continued)*

Signal Name	Width	Direction	Description
pio_clients_wdata	64	I	pio write data
pio_client_32b	1	I	N/A
ipp_pio_ack	1	O	if 1, ipp ack back the pio access request
ipp_pio_rdata	64	O	pio read data from ipp
ipp_pio_err	1	O	if 1, ipp reports an error pio access
ipp_pio_intr	1	O	if 1, ipp reports an interrupt condition
ipp_debug_port	32	O	ipp's debug bus
zcp_pio_ack	1	O	if 1, zcp ack back the pio access request
zcp_pio_rdata	64	O	pio read data from zcp
zcp_pio_err	1	O	if 1, zcp reports an error pio access
zcp_pio_intr	1	O	if 1, zcp reports an interrupt condition
zcp_debug_port	32	O	zcp's debug bus
fflp_pio_ack	1	O	if 1, fflp ack back the pio access request
fflp_pio_rdata	64	O	pio read data from fflp
fflp_pio_err	1	O	if 1, fflp reports an error pio access
fflp_pio_intr	1	O	if 1, fflp reports an interrupt condition
fflp_debug_port	32	O	fflp's debug bus

6.16 NIU_RXC_TOP Sub-Modules

6.16.1 niu_ipp

Refer to NIU_IPP Microarchitecture Specification

6.16.2 fflp

Refer to FFLP Microarchitecture

6.16.3 niu_zcp

Refer to ZCP Microarchitecture Block

6.16.4 tcam

The NIU_cam is a 200-bit wide, 128-entry on-chip ternary cam.

Each data bit of every entry has a mask bit. When a mask bit is set to 0, that associated data bit becomes a matched bit among the 200 bits of a comparing entry.

The tcam operates at 375Mhz.

A comparison takes three cycles to produce a matching result. Then one more niu_clk registers out the comparison results.

A pio_write or pio_read to the tcam takes two cycles.

The tcam is a synchronous and static one.

Venders provide testability circuitries.

6.16.5 vlan_table

The vlan_table is a single port 4096-entry x 9-bit, or 4096-entry x 18-bit sram.

6.17 NIU_IPP Microarchitecture Specification

6.17.1 NIU_IPP Overview

Input Port Processor, the ipp, is one of the three RXC submodules.

ipp loads packets from MAC, passes packet's header to fflp, gets checksum information back from fflp, then, does checksum, and writes packets to the ipp_data_fifo which is the data source of the RDMC.

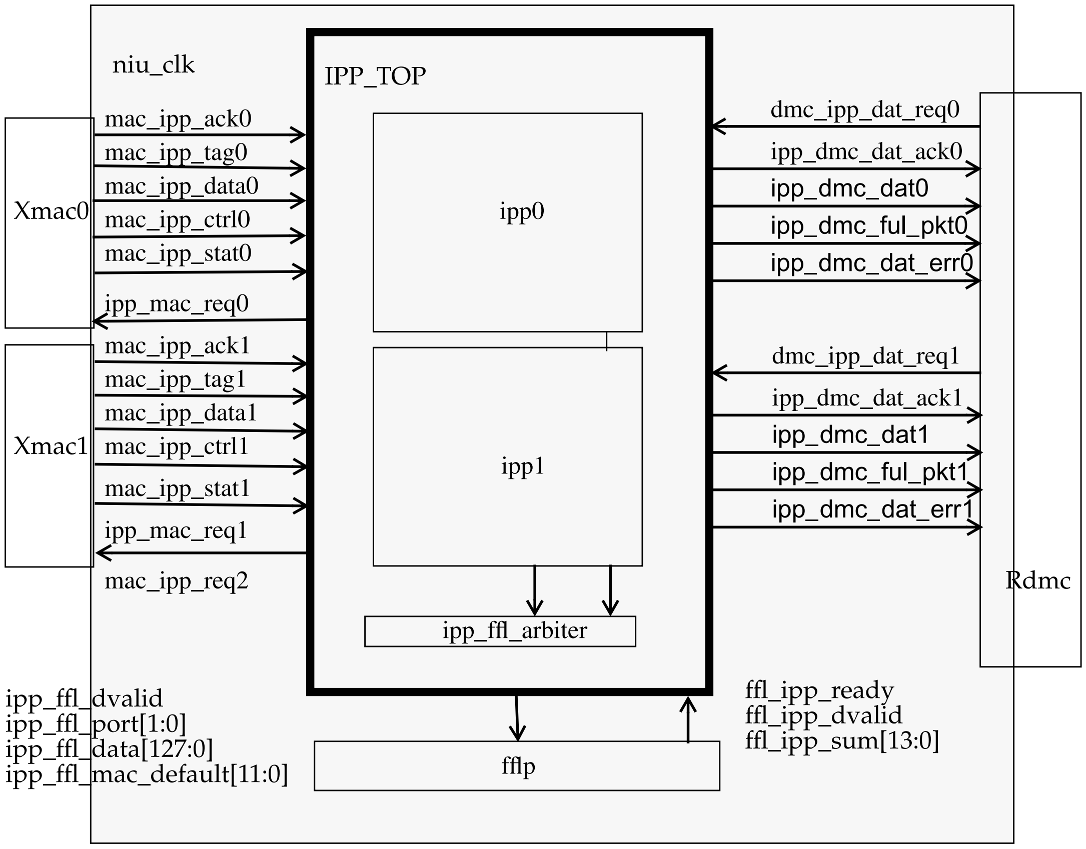
At the ipp_top level, there is an ffl_arbiter and, multiple ipps can be instantiated. The number of ipps instantiate corresponds to the number of the MAC ports that the chip supports.

The ipp_ffl_arbiter is a pure round-robin arbiter.

The ipp supports full checksum for Tcp and UDP packets.

To provide RDMC the early information of a packet's byte size and other conditions, at the beginning of an original Ethernet packet, ipp injects an ipp_data_status_word(128-bits). The data_status_word indicates the packet_length(calculated by the MAC), whether it has bad_mac_crc, mac_abort, or if this packet has a bad checksum.

FIGURE 6-12 NIU_IPP Interfacing Blocks Diagram



6.17.2 NIU_IPP Interface signals

TABLE 6-7 NIU_IPP Top Level Interface Signals

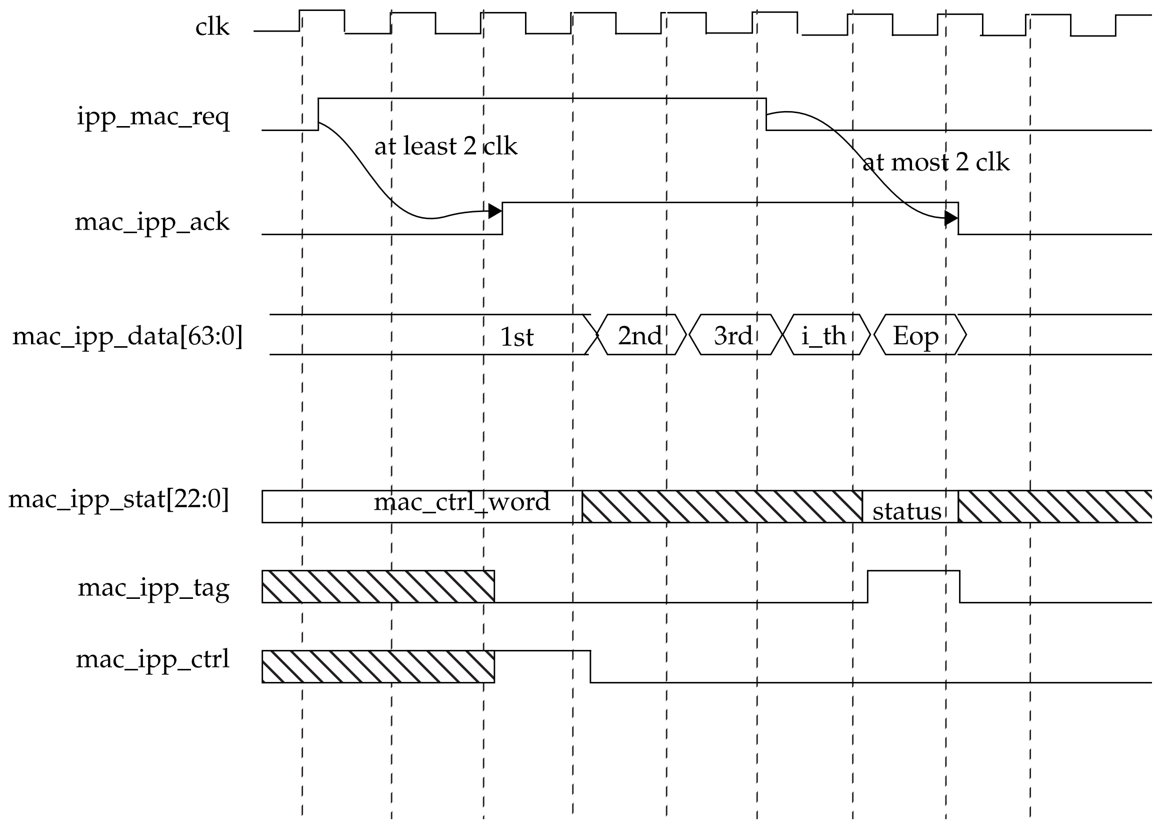
Signal name	Width	Direction	Description
MAC Interface			
mac_ipp_ack0	1	I	Xmac0 sends the ack to ipp0
mac_ipp_tag0	1	I	Xmac0 identifies the end of a packet
mac_ipp_data0	64	I	Xmac0 writing the packet's data to ipp0
mac_ipp_ctrl0	1	I	Active high for packet's control information
mac_ipp_stat0	23	I	Xmac0 writing the packet's status to ipp0
ipp_mac_req0	1	O	Request (as rdy) from ipp0 to xmac0
mac_ipp_ack1	1	I	Xmac1 sends the ack to ipp1
mac_ipp_tag1	1	I	Xmac1 identifies the end of a packet
mac_ipp_data1	64	I	Xmac1 writing the packet's data to ipp1
mac_ipp_ctrl1	1	I	Active high for packet's control information
mac_ipp_stat1	23	I	Xmac1 writing the packet's status to ipp1
ipp_mac_req1	1	O	Request (as rdy) from ipp1 to xmac1
RDMC interface			
dmc_ipp_dat_req0	1	I	Rdmc requests data from ipp_dat_fifo_0
ipp_dmc_dat_ack0	1	O	If 1, validates ipp_dat_fifo_0 data to rdmc
ipp_dmc_data0	130	O	ipp_dat_fifo_0's data to rdmc
ipp_dmc_ful_pkt0	1	O	If 1, ipp_dat_fifo_0 has complete packet/s
ipp_dmc_dat_err0	1	O	If 1, ipp_dat_fifo_0 data has error
dmc_ipp_dat_req1	1	I	Rdmc requests data from ipp_dat_fifo_1
ipp_dmc_dat_ack1	1	O	If 1, validates ipp_dat_fifo_1 data to rdmc
ipp_dmc_data1	130	O	ipp_dat_fifo_1's data to rdmc
ipp_dmc_ful_pkt1	1	O	If 1, ipp_dat_fifo_1 has complete packet/s
ipp_dmc_dat_err1	1	O	If 1, ipp_dat_fifo_1 data has error
fflp interface			
fflp_ipp_ready	1	I	fflp is ready to accept data from ipp
fflp_ipp_dvalid	4	I	Each line validates the fflp output of a port
fflp_ipp_sum	14	I	Parsed header information for a checksum

TABLE 6-7 NIU_IPP Top Level Interface Signals *(Continued)*

Signal name	Width	Direction	Description
ipp_fflp_dvalid	1	O	Packet header data is valid to fflp
ipp_fflp_port	2	O	Encoded 4 mac port number
ipp_fflp_data	128	O	ipp sends fflp a packets's header data
ipp_fflp_mac_default	12	O	Default values from MAC's address table

6.17.3 NIU_IPP Interface Timing

FIGURE 6-13 IPP_XMAC Interface Timing Diagram



1. As long as IPP can accept packets from Xmac, IPP asserts ipp_mac_req=1,
2. Once Xmac has packet data ready, it sends data to IPP, validated by setting mac_ipp_ack=1,
3. When mac_ipp_tag=1, it indicates this is the end of the packet and it also validates the packet's mac_status_word via the mac_ipp_stat[22:0] bus,
4. When mac_ipp_ctrl=1, it indicates the beginning of the packet and it also validates the packet's mac_ctrl_word via the mac_ipp_stat[22:0] bus.

FIGURE 6-14 IPP_FFLP Interface Timing Diagram

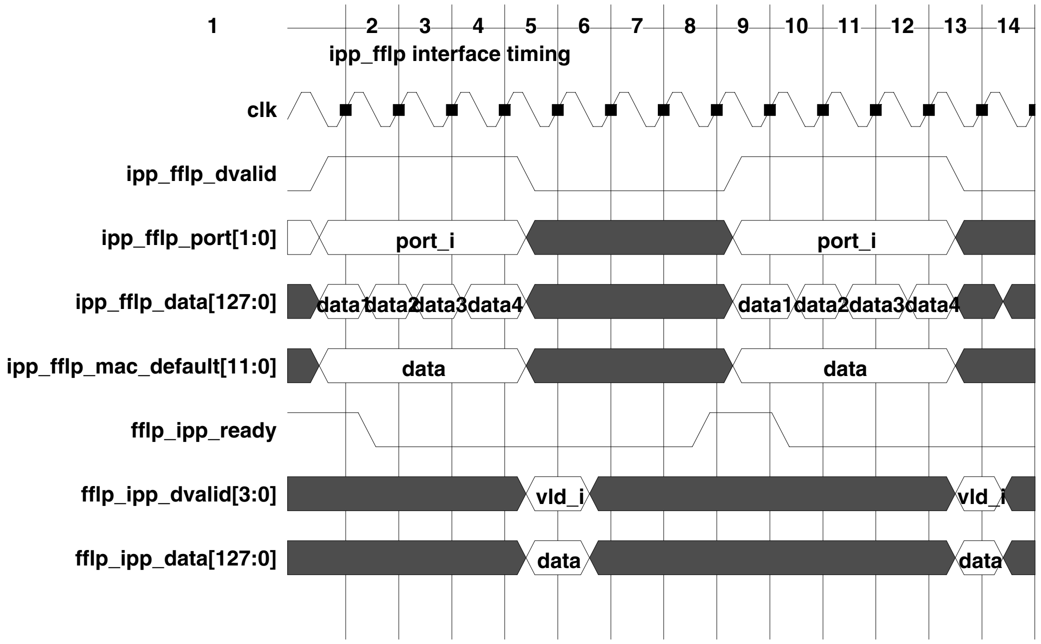
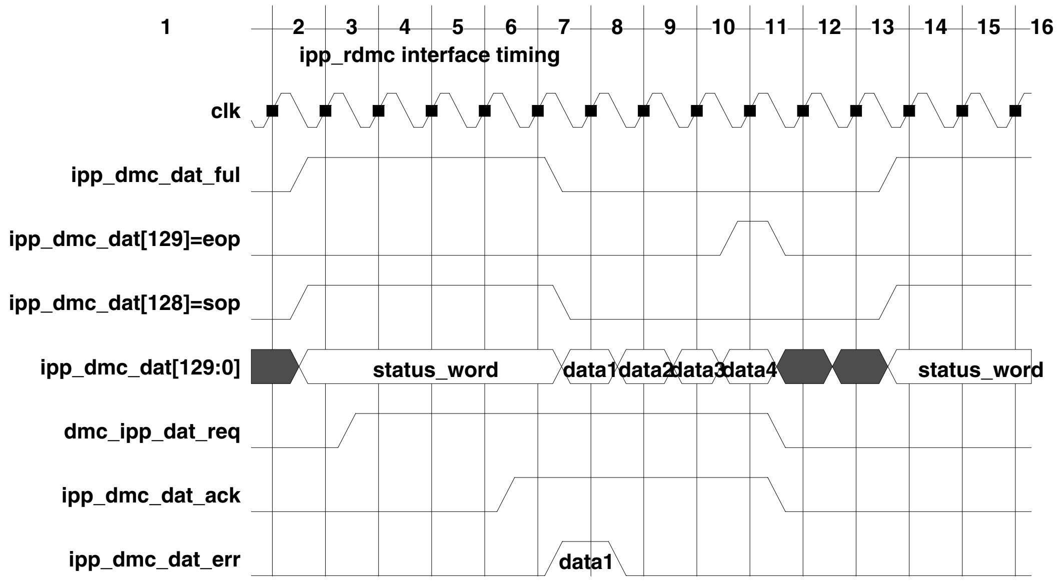


FIGURE 6-15 IPP_RDMC Interface Timing Diagram



6.17.4 IPP Operation

ipp is loosely a two pipeline-stage operation.

The first pipeline-stage, the ipp_Load, receives MAC's packet data, writes them into ipp_pre_fifo, sends a packet's header to fflp.

The second pipeline-stage, the ipp_Unload, receives fflp's checksum information, then reads the ipp_pre_fifo, copies them into the ipp_data_fifo which is the packet source of the RDMC.

On the way the packet data is being read from the ipp_pre_fifo and copied into the ipp_data_fifo, the checksum unit sniffs the ipp_pre_fifo data bus and does 1's complement full checksum calculations.

The ipp_Unload collects MAC's status, fflp's checksum information, and the packet's checksum results to form an "ipp_data_status_word" which becomes the first entry of a packet, stored in the ipp_data_fifo.

The IPP's data_FIFO requires "random access".
At the end of an Ethernet packet being copied from ipp_pre_fifo into ipp_dat_fifo, the hardware jumps back to the beginning of this packet and inserts the "ipp_data_status_word" into the data_fifo as the first entry of the packet.

Multiple ipp units can be instantiated and working parallel. For packet's headers from multiple ipps to fflp, an arbiter is used to combine multiple port's header data into one data stream, sending to fflp for parsing; for packet data stream from ipp to RDMC, it is a per port independent operation to the RDMC.

FIGURE 6-16 IPP Block Diagram

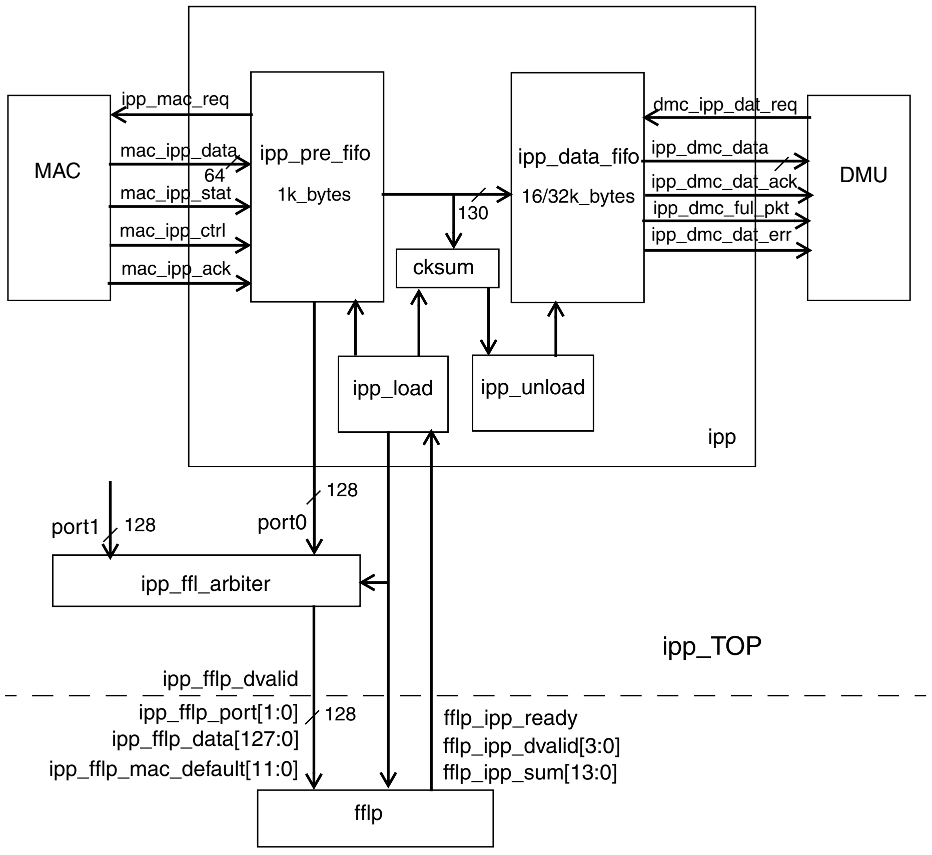
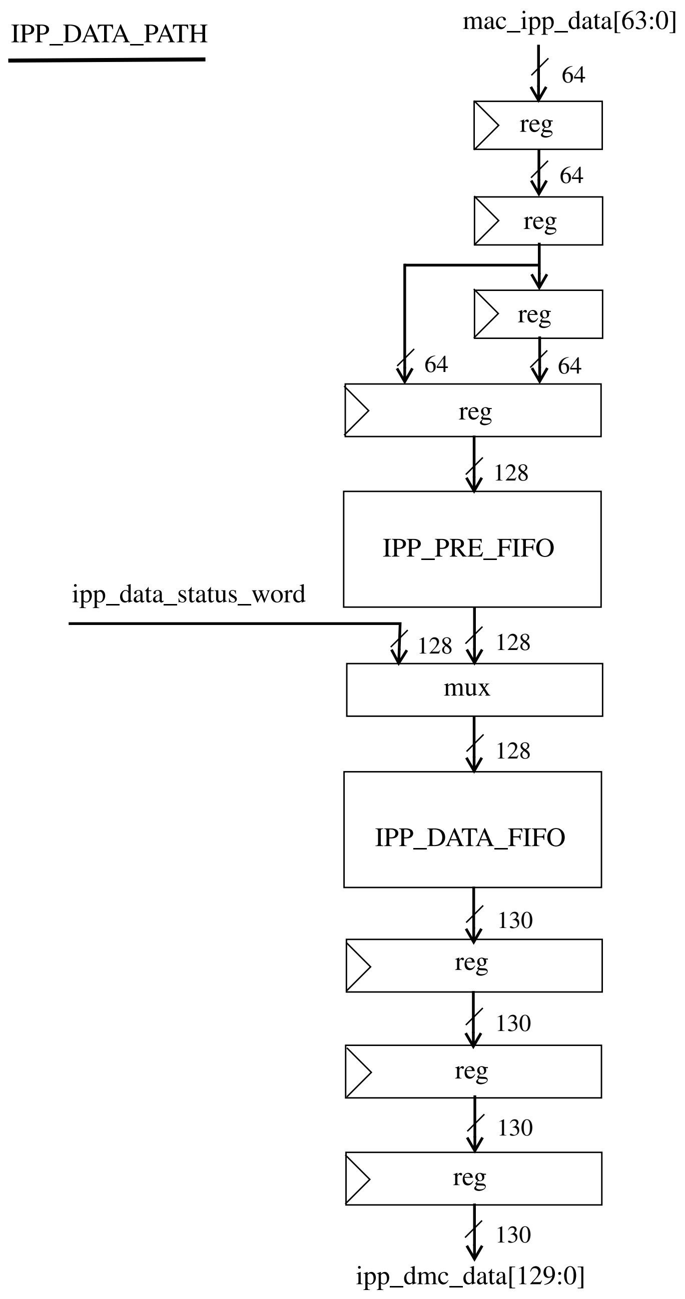


FIGURE 6-17 IPP Datapath Diagram



6.17.5 IPP_LOAD

ipp_Load unit, mainly, interfaces with MAC, fflp, and prepares for packet's unload operation.

6.17.5.1 Interface with MAC

1. Data width: ipp stages MAC's 64-bits/cycle to a 128-bits/cycle data bus, padding last bus cycle:
   a. if a MAC packet data is an odd set of 64-bits, the last entry in ipp_pre_fifo will be. Xmac: {41'h0,mac_status[22:0],mac_data[63:0]}
   b. if a MAC packet data is an even set of 64-bits, the last entry in ipp_pre_fifo will be. Xmac: {41'h0,mac_status[22:0],41'h0,mac_status[22:0]}
2. This extra word will be part of the packet written into the ipp_pre_fifo, but it will be filtered out when the packet is finally written into the ipp_data_fifo.
3. Smaller than 57 bytes/pkt: ipp quietly drop the packet and increase the pkt_drop_counter.

6.17.5.2 Interface with FFLP

1. Size: 64 bytes to 128 bytes per packet, i.e., the first 4,5,6,7, or 8 bus cycles of a packet.

2. Consecutive: ipp accumulates sufficient header bytes and sends them to fflp in one shot.

3. Fixed cycle response from fflp

If at the ith cycle, ipp sends fflp the first valid header data

. at the i+1th cycle, fflp indicates fflp is busy,

. at the i+4th cycle, fflp provides ipp the checksum information,

. at the i+7th cycle, fflp indicates fflp is ready to accept the next header.

4. Arbitration: a round-robin arbiter controls which port's header gets to fflp

TABLE 6-8 <fflp_ipp_sum> Checksum Information from fflp to ipp

#	bit	field	function
1	[1:0]	L2_option[1:0]	{LLC,VLAN}
2	[3:2]	L3_version[1:0]	00:unknown, 01:IPv4, 10:IPv6, 11:reserved

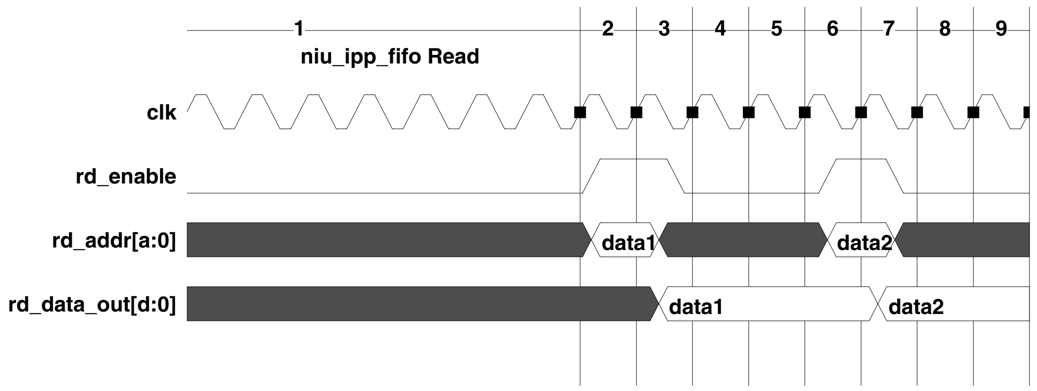
TABLE 6-8 <fflp_ipp_sum> Checksum Information from fflp to ipp *(Continued)*

#	bit	field	function
3	[7:4]	IPv4_hd_length[3:0]	Ipv4's header_length_field
4	[9:8]	L4_protocol[1:0]	00:unknown, 01:Tcp, 10:UDP, 11:reserved
5	[13:10]	fflp_pkt_id[3:0]	fflp issued packet ID

6.17.5.3 SRAM

There is a 64-entry x 146-bit two port SRAM used as the ipp_pre_fifo per port.

FIGURE 6-18 IPP_FIFO Read Timing Diagram



6.17.6 IPP_UNLOAD

Ipp_Unload, mainly, interfaces with the RDMC, forwarding the stored packets to RDMC.

6.17.6.1 Interface with RDMC

1. A simple rdmc_request and ipp_ack protocol.
2. "ipp_ful_pkt" signal: to start requesting a new packet, RDMC needs to check if the "ipp_ful_pkt"=1, i.e., there is, at least, one complete packet existed to be read out by RDMC.
3. Ipp_data_status_word

The 128-bit word is inserted and is the first entry of a packet to RDMC.

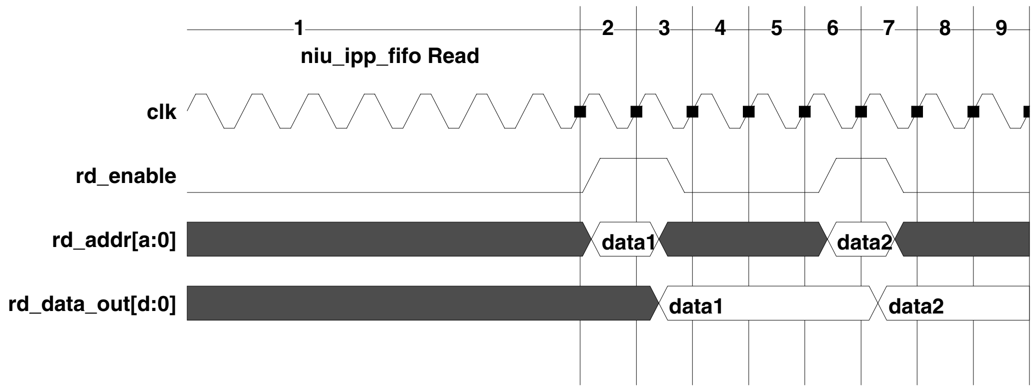
TABLE 6-9 <ipp_data_status_word> a Packet Information from ipp to rdmc

#	Bit	Field	Function
1	[15:0]	IP_cksum[15:0]	Hardware calculated checksum value
2	[16]	Full_checksum	If 1, a full cksum can be calculated
3	[17]	Full_cksum_err	If 1, hardware reports a bad full checksum
4	[18]	Full_cksum_ena	If 1, full checksum calculation is enabled
5	[19]	reserved	0
6	[35:20]	IP_length[15:0]	IPv4/6 header's IP_length field
7	[37:36]	L2_option[1:0]	{LLC,VLAN}
8	[39:38]	L3_version[1:0]	00:unknown, 01:IPv4, 10:IPv6, 11:reserved
9	[43:40]	IPv4_hd_length[3:0]	IPv4's header length
10	[45:44]	L4_protocol[1:0]	00:unknown, 01:Tcp, 10:UDP, 11:reserved
11	[49:46]	fflp_pkt_id[3:0]	fflp issued packet ID
12	[51:50]	reserved	0's
13	[52]	inc_ipp_dsc_pkt_cnt	If 1, this packet increases ipp_discard_packet_counter. This bit becomes 1 if an ipp_pre_fifo_parity_error is detected and the config_reg[3] is enabled.
14	[53]	reserved	0
15	[54]	MAC_abort	If 1, MAC determines to drop this packet
16	[71:55]	reserved	0's
17	[84:72]	dfifo_sop_addr[12:0]	ipp_data_fifo sop address
18	[87:85]	reserved	0's
		Below are the direct copy of the mac_rxc_status	
19	[101:88]	pkt_length[13:0]	MAC reported packet length in bytes
20	[102]	bad_crc	If 1, MAC detects a bad CRC, for diagnostic
21	[103]	MAC_abort	If 1, MAC determines to drop this packet
22	[x:104]	hash_value	MAC's hash value, Xmac:x=109,
23	[y]	hash_hit_match	MAC has hash hit, Xmac:y=110,
24	[127:z]	reserved	0's, Xmac:z=111,

6.17.6.2 SRAM

For OpenSPARC T2 mode, port_0/port_1, there is a 1024-entry x 146-bit two port sram used as ipp_data_fifo per port.

FIGURE 6-19 IPP_FIFO Read Timing Diagram



The ipp DATA_FIFO uses “single bit error correction, double bit error detection” ECC algorithm.

6.17.7 Checksum

The Checksum unit supports 1′s complement full checksum per port.

The overhead of the checksum is 4 cycles per packet.

6.18 NIU_PIO Microarchitecture Specification

6.18.1 NIU_PIO Overview

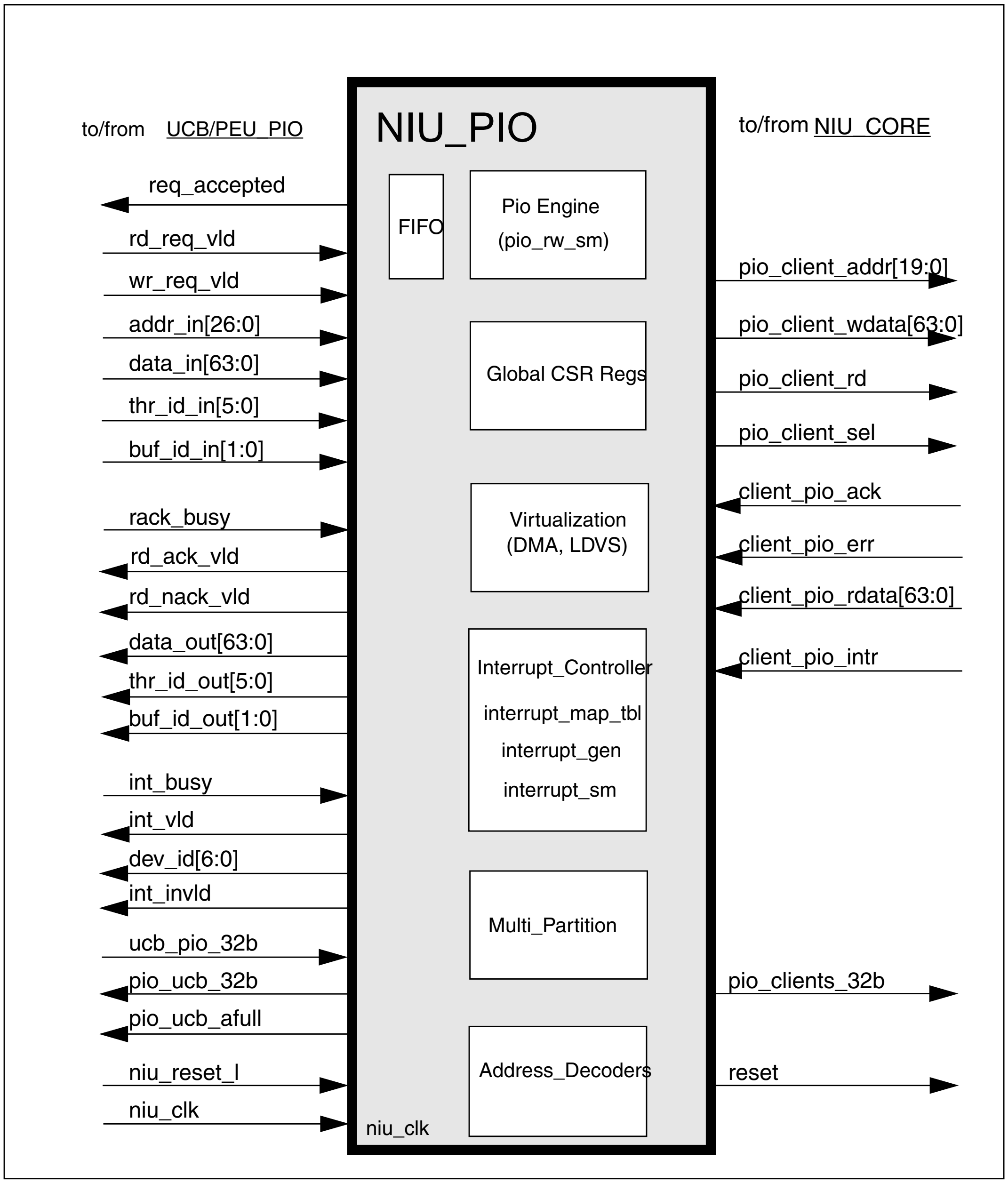
The NIU_PIO block is a generic programmable IO interface for NIU. It converts NIU internal PIO protocol to a generic protocol that is suitable for OpenSPARC T2

To communicate with the host, for OpenSPARC T2 mode, the niu_pio talks to NCU (Non Cacheable Unit) block via UCB.

The NIU_PIO block provides the following functions:

- Decode generic host PIO requests and turns them into internal PIO cycles.
- Provide pio address space global configuration, interrupt status, interrupt masks and diagnostic registers.
- Supports non-posted PIO read cycle and posted PIO write cycle.
- Supports split transactions.
- The niu_PIO block also hosts registers for the blocks that do not have PIO interface signals. In the OpenSPARC T2 mode, niu_PIO hosts registers for SMX and Meta Arb. Accept interrupt signals from internal modules, group them and generate generic interrupt vectors sent to UCB.

FIGURE 6-20 NIU_PIO and Interfacing Blocks Block Diagram



6.18.2 NIU_PIO Interface Signals

TABLE 6-10 NIU_PIO Top Level Interface Signals

Signal name	Width	Direction	Description
UCB Interface			
rd_req_vld	1	I	read request.
wr_req_vld	1	I	write request
addr_in[26:0]	27	I	address in
data_in[63:0]	64	I	write data in
thr_id_in[5:0]	6	I	thread ID in
buf_id_in[1:0]	2	I	buffer ID in
rack_busy	1	I	read busy
rd_ack_vld	1	O	read acknowledgment This signal is mutually exclusive with rd_nack_vld
rd_nack_vld	1	O	read not acknowledgment; bad address detected.
req_accepted	1	O	read or write request acceptance
data_out[63:0]	64	O	read return data
thr_id_out[5:0]	6	O	thread ID outgoing
buf_id_out[1:0]	2	O	buffer ID outgoing
MSI interface			
int_busy	1	I	Interrupt busy
int_vld	1	O	Interrupt valid
dev_id[6:0]	7	O	dev_id is valid only when int_vld/int_invld is ’1’. dev_id[6:5] == msi_fun_num[1:0] dev_id[4:0] == msi_vector[4:0]

6.18.3 NIU_PIO Interface Timing

6.18.3.1 PIO - Client Interface Protocol

FIGURE 6-21 Client Side PIO Read Timing Diagram

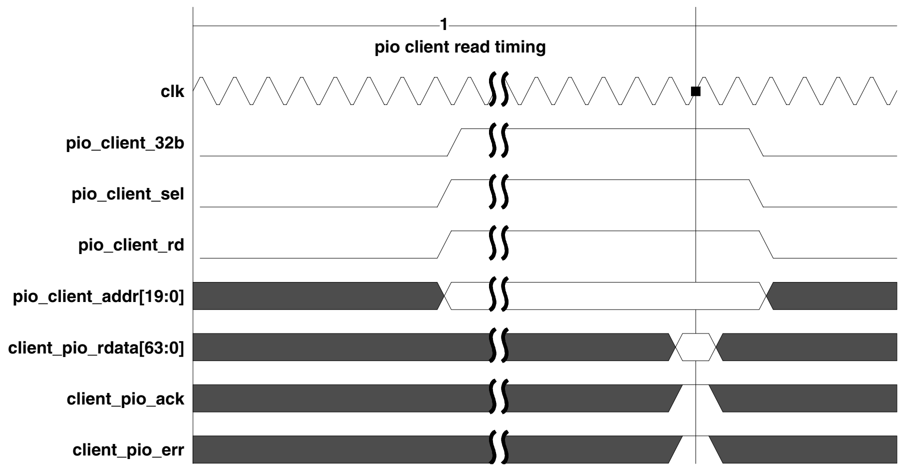
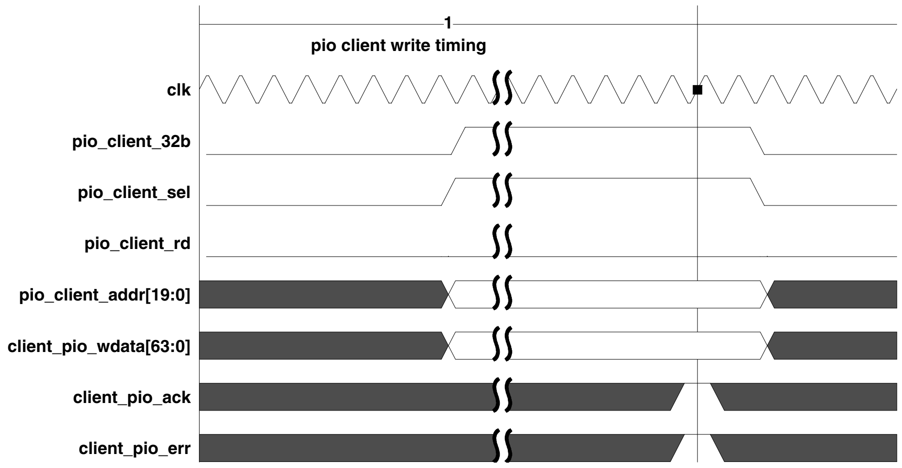


FIGURE 6-22 Client Side PIO Write Timing Diagram



6.18.3.2 PIO - UCB Interface Protocol

FIGURE 6-23 Back_to_Back Read Timing Diagram

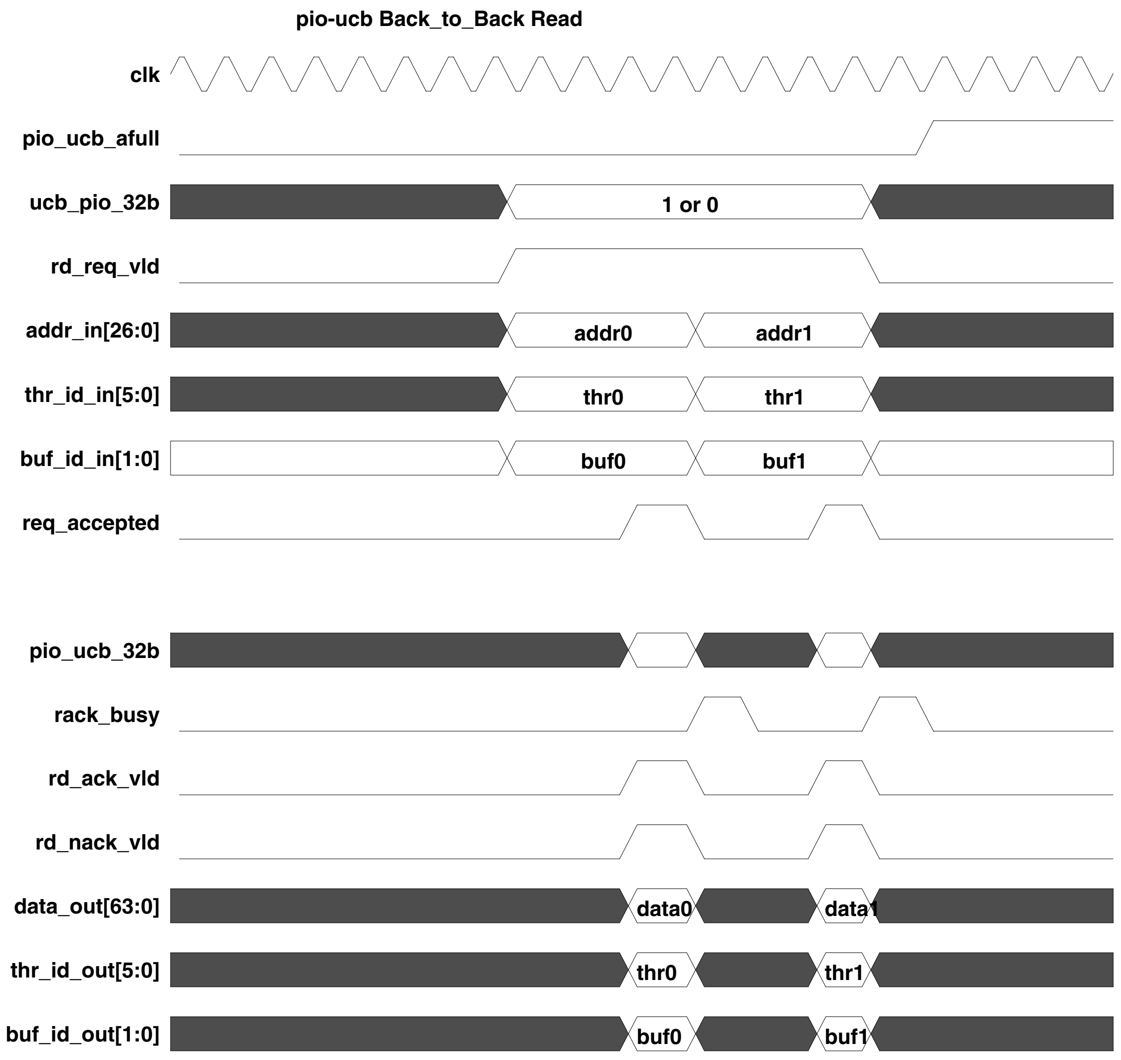

pio-ucb Back_to_Back Read
clk
pio_ucb_afull
ucb_pio_32b
1 or 0
rd_req_vld
addr_in[26:0]
addr0
addr1
thr_id_in[5:0]
thr0
thr1
buf_id_in[1:0]
buf0
buf1
req_accepted
pio_ucb_32b
rack_busy
rd_ack_vld
rd_nack_vld
data_out[63:0]
data0
data1
thr_id_out[5:0]
thr0
thr1
buf_id_out[1:0]
buf0
buf1

FIGURE 6-24 Back_to_Back Write Timing Diagram

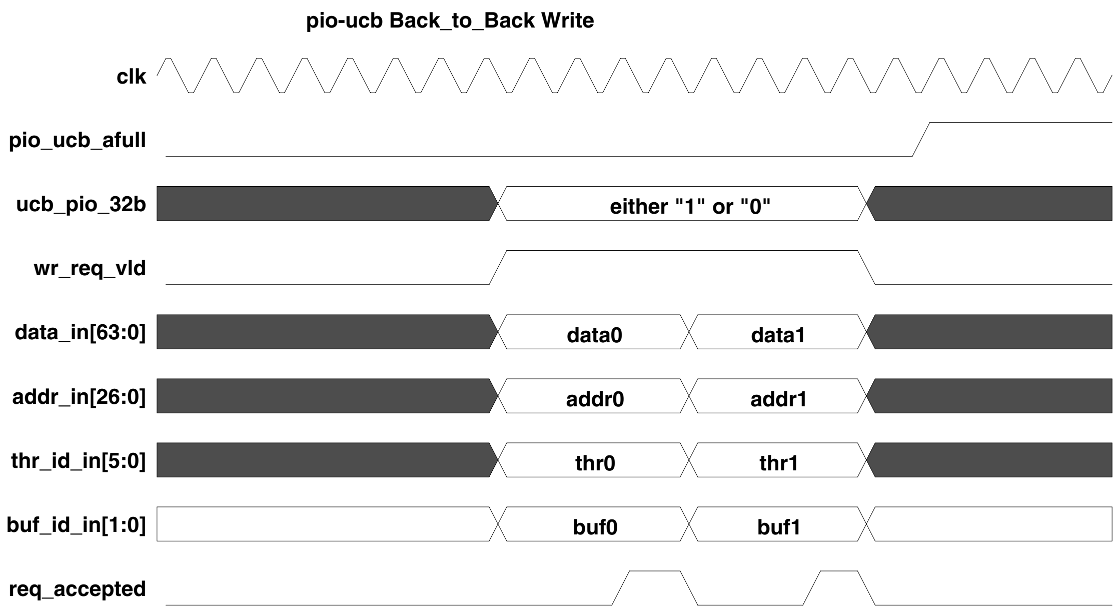
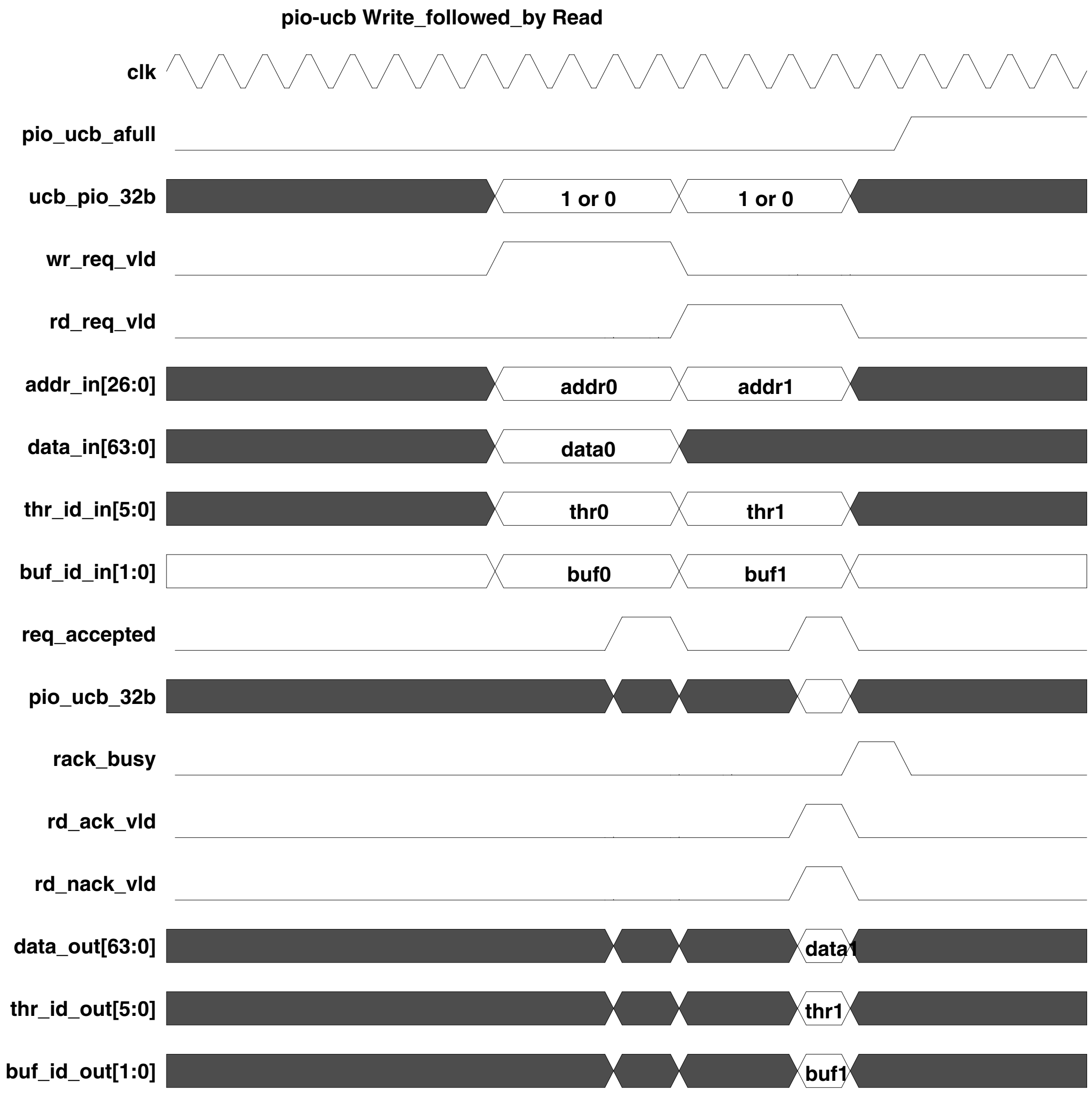


FIGURE 6-25 Write followed by Read Timing Diagram



6.18.3.3 PIO Write Cycle

When a PIO write command is executed to the niu_PIO block, the signal 'wr_req_vld' is asserted to indicate the followings are valid, the addr_in[26:0], data_in[63:0], thr_id_in[5:0] and buf_id_in[1:0].

niu_PIO block asserts a one clock pulse wide signal ’req_accepted’ to indicate the acceptance of the PIO write command. Based on the ’addr_in[26:0]’, niu_PIO block will update the CSR information accordingly. The signals ’thr_id_in[5:0],’ and ’buf_id_in[1:0]’ are not used in the PIO write case. Write command to invalid address should be discarded silently. This is a posted write operation. There is no ’ack’ or ’nack’ signal going back to UCB interface block.

6.18.3.4 PIO Read Cycle

In the read case, ’rd_req_vld’ signal is asserted to indicate a read command is ready and the following signals are valid, the addr_in[26:0], thr_id_in[5:0], and buf_id_in[1:0].

rd_ack_vld and rd_nack_vld are mutually exclusive and are used to qualify the following signals, the data_out, thr_id_out and buf_id_out.

niu_PIO block asserts a one clock pulse wide signal ’req_accepted’ to indicate the acceptance of the PIO read command. Based on the addr_in[26:0], niu_PIO block will read the corresponding register and put the return data on ’data_out[63:0]’. Along with the signals ’thr_id_out[5:0],’ and ’buf_id_out[1:0]’ that came with the read command, niu_PIO block asserts the ’rd_ack_vld’ to send the return data. In the case of an unsuccessful read, e.g., caused by invalid address, niu_PIO block asserts the signal ’rd_nack_vld’ along with the ’thr_id_out[5:0]’ and ’buf_id_out[1:0]’ signals which came from the read message. If rack_busy is asserted, niu_PIO block waits until it is cleared.

niu_PIO block supports split transactions. The read response can occur at a later time.

In the case of ’rack_busy’ signal is asserted, niu_PIO block will not assert ’rd_ack_vld’ nor ’rd_nack_vld’ until ’rack_busy’ signal is de-asserted.

6.18.3.5 PIO Error Condition

In NIU mode, there are two conditions that can result in nack.

(1) Address is out of range.
(2) Time out from client.

The time out value has a default of 1024 and is a programmable value. I

6.18.4 Interrupt Controller Microarchitecture

Internal logical device sends a level interrupt signal to niu_PIO block. niu_PIO block will generate a pulse at group level to the interfacing host module. After generating one interrupt, software needs to re-enable the device by clearing the status bits which generate the interrupt. If there is another event pending on the logical device, as soon as the device is enabled, an interrupt will be issued.

1. **Interrupt Mapping Table**

There is a 64 deep and 7 bit wide programmable MSI data table which is used to generate MSI function number and MSI Vector/data.

The output bus name is call dev_id.

dev_id is valid only when int_vld/int_invld is '1'.

dev_id[6:5] == msi_fun_num[1:0]
dev_id[4:0] == msi_vector[4:0].

2. **OpenSPARC T2 Interrupts**

On OpenSPARC T2, interrupt signal is sent to the NCU. Inside NCU there is an ID table, where the thread ID and interrupt vector number are stored. NCU uses the table information to issue a trap to the CPU.

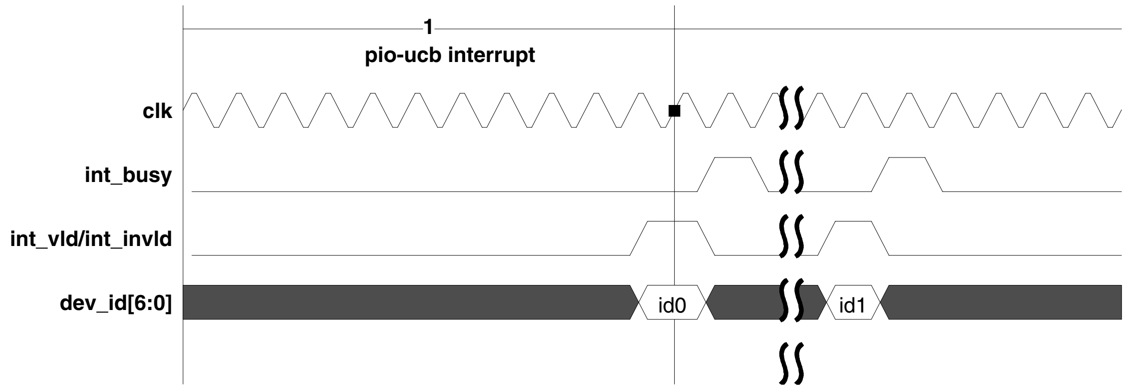
3. **Interrupt Timing Diagram**

   a. **UCB Interrupt Timing Diagram**

   The niu_PIO block sends interrupt along with a Device I.D. (dev_id[6:0]) to the host block - UCB(NCU). The Device I.D. is used to associate with the source of interrupts.

   Signal 'int_vld' indicates a valid interrupt request to meta block UCB(NCU), and it asserts only when "int_busy" is '0'. Signal 'int_busy' means that the device is currently busy and cannot accept interrupt. The signal 'int_invld' is generated when the source of the interrupt of a group goes away, the group is to be released. The interrupt request/release signals are generated on a per group basis.

FIGURE 6-26 UCB Side Interrupt Timing Diagram



b. **Client side Interrupt Timing Diagram**

All the interrupts other than DMU are a level signal coming to niu_PIO. These client interrupt signals stay active till they are cleared by software.

4. **Interrupt Device ID Allocation**

a. **RX: 16**

b. **TX: OpenSPARC T2 mode 15**

c. **MAC: OpenSPARC T2 mode 2**

d. **MIF: 1**

e. **System: 1**

5. **Interrupt Generation**

An interrupt generation is a function of the following factors:

a. **Logic device interrupt event, of the possible 69 Interrupt Devices,**

b. **Interrupt event mask_pair, per logic device,**

c. **Interrupt group membership,**

d. **Group Time-out counter,**

e. **Group ARM bit.**

The "ARM" bit is to gate the generation of a pio interrupt request to the host, jointly by software & hardware, at per group level.

If the ARM=1, a pio interrupt request can be generated if an interrupt condition exists in the group.

The ARM bit can only be set by the software; can be cleared by software or hardware.

6. Interrupt State Machine

Multiple groups' "interrupt requests" and "interrupt releases" can be active at the same time. The Interrupt State Machine is to choose and process one interrupt requesting group or one interrupt releasing group at a time.

"interrupt request": hardware is notifying host that there is an interrupt coming out of logic device/s of a group.

"interrupt release": hardware is notifying host that the software has cleared all the interrupting conditions within this group.

6.18.5 Virtualization

Niu-pio supports virtualization in two places:

1. DMA channel addresses,
2. LDSV registers (Logic Device State Vector).

The physical address of each DMA channel can be virtualized by programming the DMA_BIND register. See niu_pio PRM Chapter 18.2 Virtualization Region and the DMA Channel Binding register(FZC_PIO+0x10000).

6.18.6 Multi-Partition

Niu_pio handles 27 bits of pio_address[26:0].

The 27-bit pio_address space is evenly divided into four chunks, one chunk for each function.

Within each chuck, there is an FZC zone, the Function Zero Control zone.
The FZC zone registers normally can be read or written by any of the 4 functions.
The FZC zone registers can then only be written via function_0 accesses if the MPC bit is set.

6.18.7 PIO Transaction FIFO Microarchitecture

There is a 16 entry deep, 101 bit wide niu_PIO Transaction FIFO to queue up all the niu_PIO read/write operation requests from the host. It enforces strong ordering.

6.18.8 Registers for Other Modules Microarchitecture

The niu_PIO block hosts registers for the blocks that do not have niu_PIO interface signals. In the OpenSPARC T2 mode niu_PIO hosts registers for SMX and Meta Arb. FIGURE 6-11 shows interface signals from the niu_PIO block to SMX, and Meta Arb.

TABLE 6-11 <Registers for Other Modules> Interface Signals

Signal name	Width	Direction	Description
SMX Interface			
pio_smx_cfg_data	32	O	Configuration Data Register for SMX, hosted by niu_PIO
smx_pio_intr	1	I	Interrupt Signal from Smx
smx_pio_status	32	I	Status register for SMX, hosted by niu_PIO
pio_smx_clear_intr	1	O	Interrupt clear signal (pulse) for SMX
pio_smx_ctrl	32	O	Control register for SMX, hosted by niu_PIO
pio_smx_debug_vector	32	O	Debug register for SMX, hosted by niu_PIO
Meta Arb Interface			
pio_arb_ctrl	32	O	Control register for Meta Arb, hosted by niu_PIO
pio_arb_debug_vector	32	O	Debug register for Meta Arb, hosted by niu_PIO
arb_pio_all_npwdirty	1	I	Registered in niu_PIO, read accessible by software
arb_pio_all_rddirty	1	I	Registered in niu_PIO, read accessible by software
pio_arb_dirtid_enable	1	O	Register for Meta Arb, hosted by niu_PIO
pio_arb_dirtid_clr	1	O	Register for Meta Arb, hosted by niu_PIO
pio_arb_np_threshold	6	O	Register for Meta Arb, hosted by niu_PIO
pio_arb_rd_threshold	6	O	Register for Meta Arb, hosted by niu_PIO
arb_pio_dirtid_rdstatus	6	I	Registered in niu_PIO, read accessible by software
arb_pio_dirtid_npwstatus	6	I	Registered in niu_PIO, read accessible by software

6.19 FFLP Microarchitecture

6.19.1 Overview

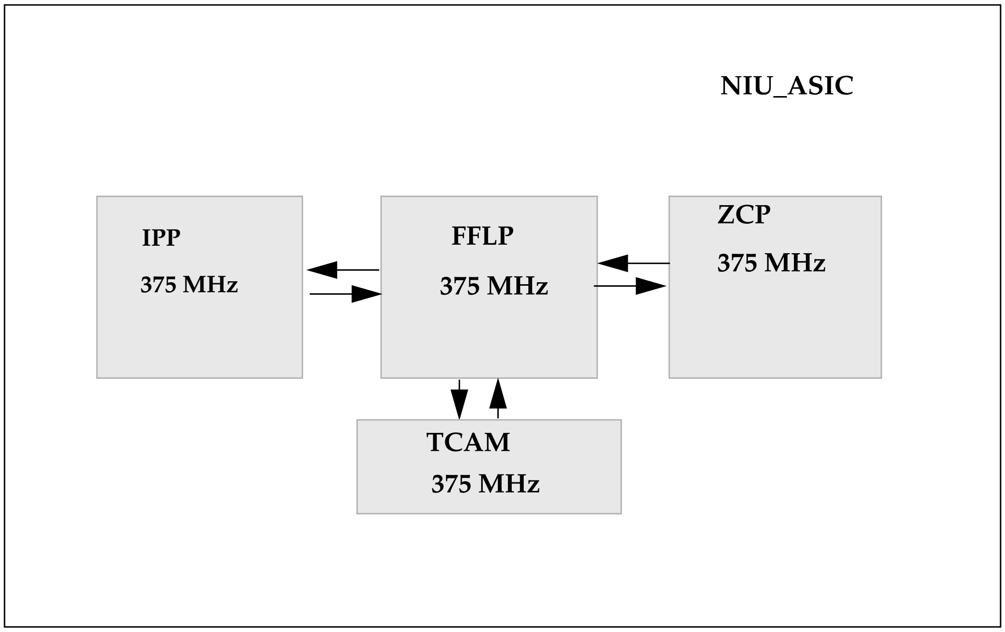
In general, the FFLP block provides link layer, network (IP) and transport layer (TCP) packet header parser, match and search mechanisms, which in conjunction with an on-chip TCAM-RAM, External hash table as well as CPU management, performs Layer 2/3/4 packet/flow classification. Based upon the result of classification, FFLP selects the receive DMAchannel for the incoming packet, provides the packet header information to IPP block to perform TCP/IP checksum calculation and in certain cases determines packet filtering status. The interfaces to FFLP are IPP, ZCP, PIO, and CAM.

The main functions of the FFLP are:

- Perform fast, real time L2/L3/L4 header parsing. The parser parses through TCP header to assist packet reassembling.
- Perform register based L3/L4 packet classification (class filter).
- Based on 1, 2, generate TCAM key to search and match the TCAM database to perform CAM based classification.
- Generate flow key and hash key to external hash table to do further flow classification, if needed.
- provide a command interface for software to manage both CAM and hash table entries, such as adding new entries, updating entries, invalidating unwanted entries, etc.
- FCRAM controller to manage the access to external FCRAM.

Note – Flow table lookups to the external FCRAM are not supported by the current version of T2 NIU. References to the flow table lookups and external FCRAM in this chapter are meant for reference only.

FIGURE 6-27 FFLP and Interfacing Blocks Block Diagram



6.19.2 Interface Signals

TABLE 6-12 FFLP Top Level Interface Signals

Signal name	width	direction	Description
fflp_ipp_ready	1	O	FFLP is ready to receive header
ipp_fflp_dvalid	1	I	IPP header data is valid.
ipp_fflp_data	128	I	Header data sent by IPP. The header data sent to fflp is between 64 bytes to 128bytes depending on packet size.
ipp_fflp_port	2	I	The MAC port which the header data comes from.
ipp_fflp_mac_index	8	I	
ipp_fflp_mac_default	11	I	
fflp_ipp_dvalid	1	O	FFLP checksum info to IPP is ready.
fflp_ipp_data	16	O	The checksum info from packet header.
fflp_zcp_wr	1	O	Control data is ready to write to control Fifo
fflp_zcp_data	216	O	Control data from classification
fflp_zcp_port	4	O	Which MAC port the packet comes from.
zcp_fflp_rdy	1	I	The fifo between fflp and zcp is full.
cam_compare	1	O	Start cam search
cam_data_inp	200	O	Cam search key
cam_pio_wr	1	O	PIO write cam key or cam mask
cam_pio_rd	1	O	PIO read cam key or cam mask
cam_pio_sel	1	O	Select key or mask when pio read/write cam. 0: cam key, 1: mask.
cam_index	10	O	Address for pio read/write cam key or mask. When not all bits are used, the higher bits are not used.
cam_hit	1	I	cam search results in a match.
cam_valid	1	I	Qualify cam_hit signal, not used by fflp.
cam_haddr	10	I	Match index when cam hit. If not all bits are used, higher bits are tied to 0.
pio_rd_valid	1	I	Indicate cam_msk_dat_out is valid for cam_pio_rd.

TABLE 6-12 FFLP Top Level Interface Signals *(Continued)*

Signal name	width	direction	Description
cam_msk_dat_out	200	I	cam pio read data (key/mask)
fflp_pio_ack	1	O	fflp ask to pio access.
fflp_pio_rdata	64	O	pio read data.
fflp_pio_err	1	O	pio access results in error.
fflp_pio_intr	1	O	fflp interrupt to pio.
pio_fflp_sel	1	I	pio select signal.
pio_fflp_rd	1	I	pio read/write. 1: read, 0: write.
pio_fflp_addr	20	I	pio read/write address.

6.19.3 Interface Timing

FIGURE 6-28 IPP <-> FFLP Interface Timing

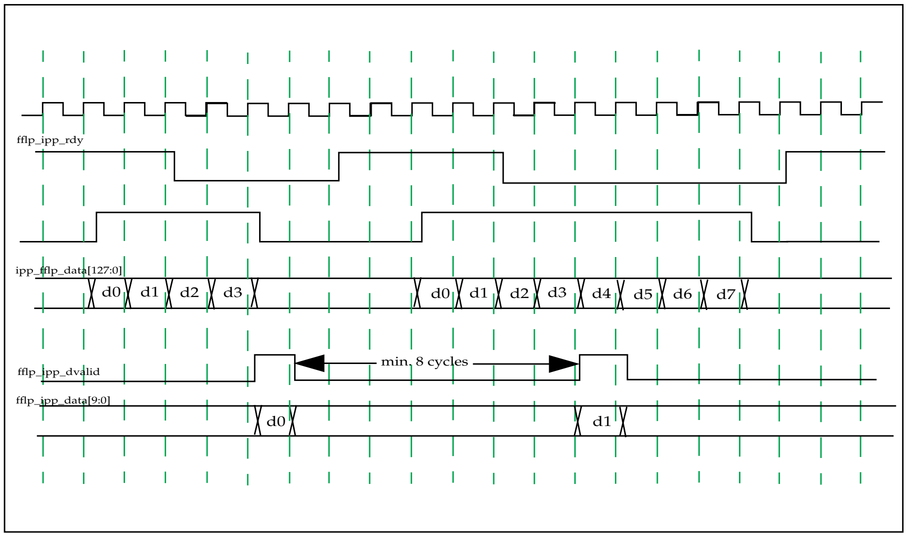


FIGURE 6-29 FFLP <-> ZCP Interface Timing

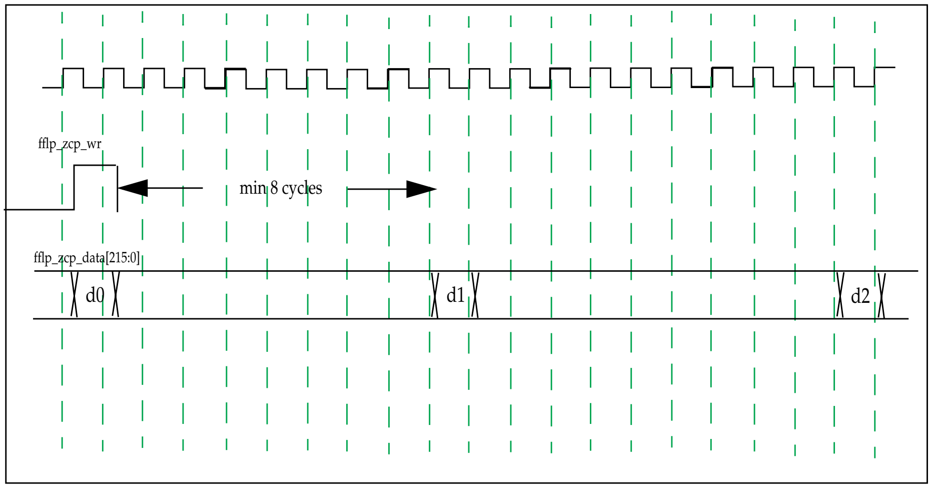
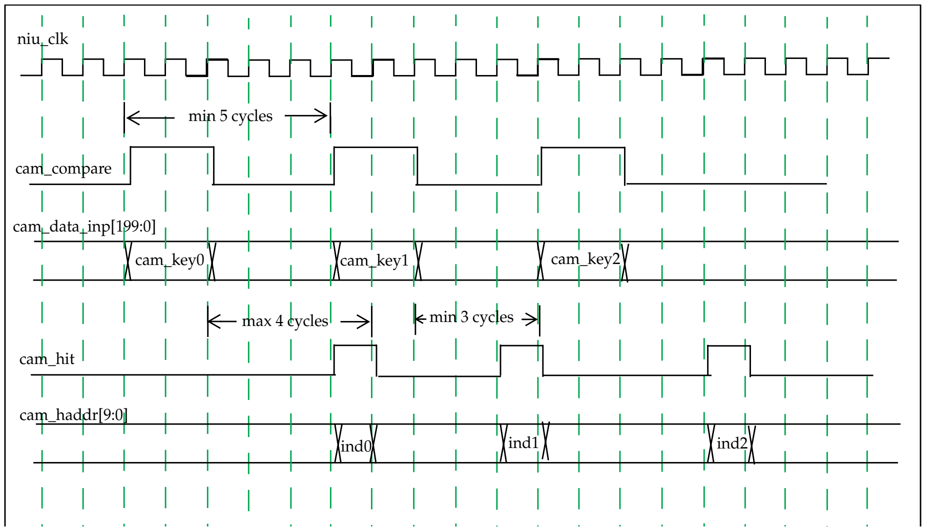


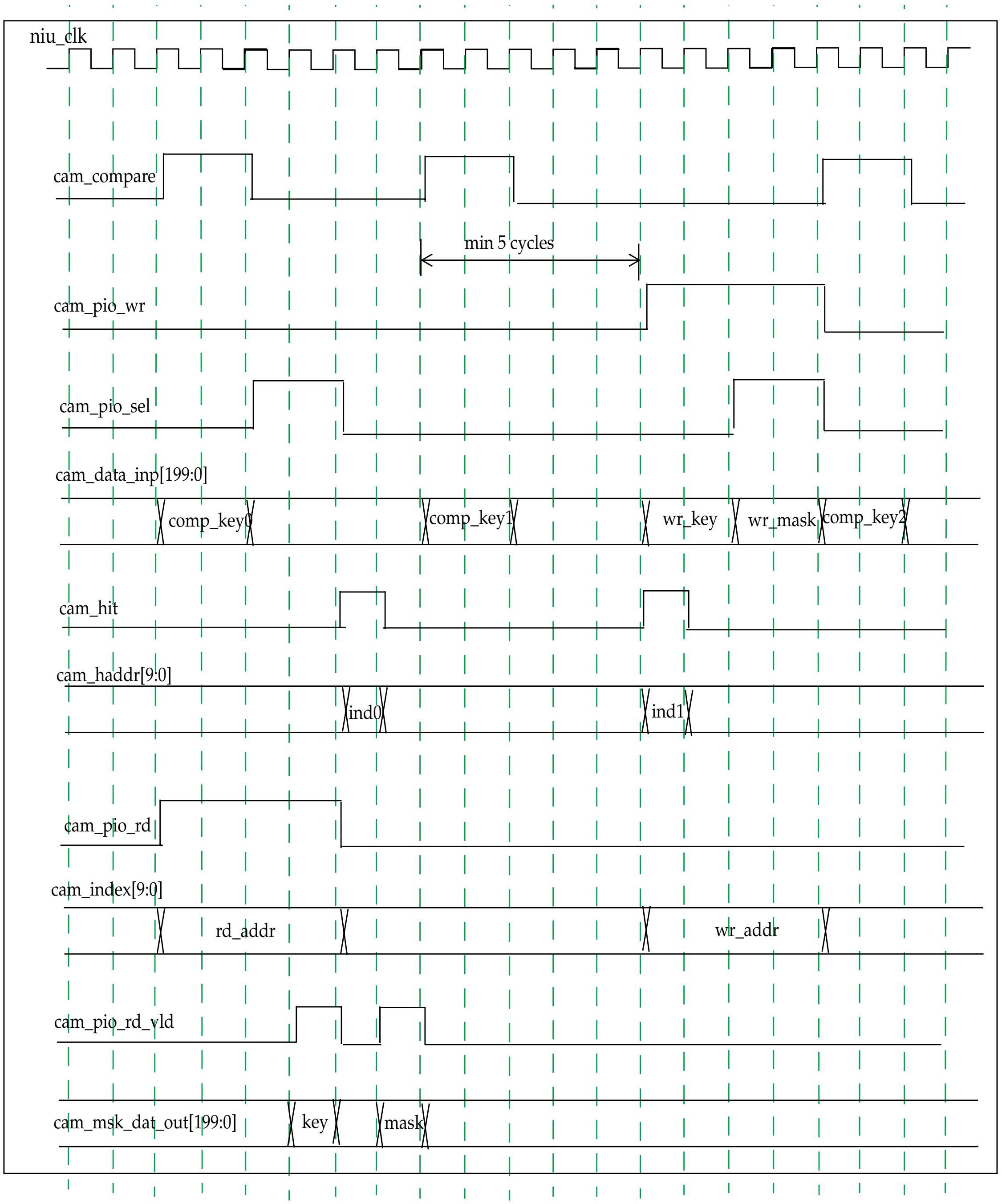
FIGURE 6-30 Timing Reference for Back to Back CAM Search



6.19.3.1 Principle of CAM Access:

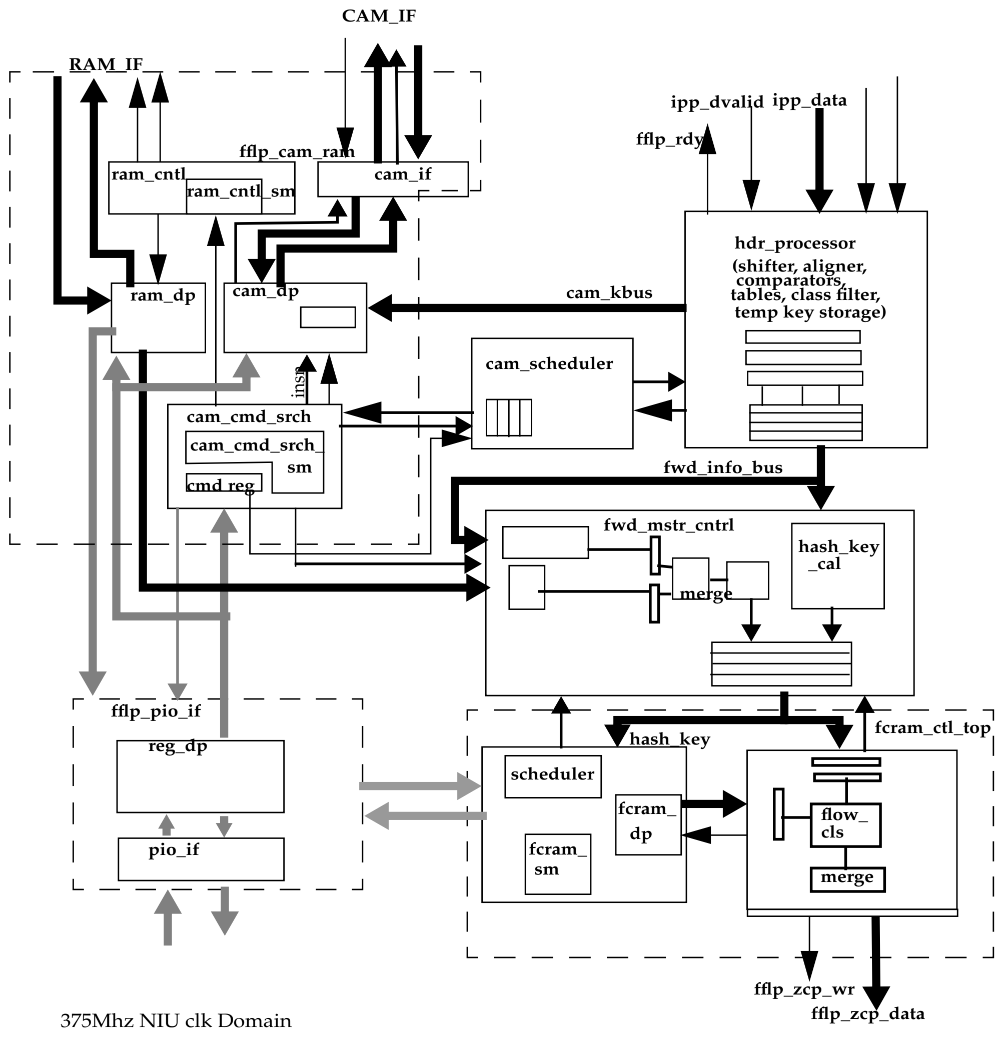
1. Minimum number of cycles between two comparison commands is 5.
2. A TCAM comparison takes three niu_clk cycles.
3. A TCAM read or write takes two niu_clk cycles.
4. TCAM write and comparison can not be active at the same time.
5. TCAM read and comparison can be active at the same time.
6. TCAM read and write can not be active at the same time.

FIGURE 6-31 Example of Interleaved CAM Search and PIO Access



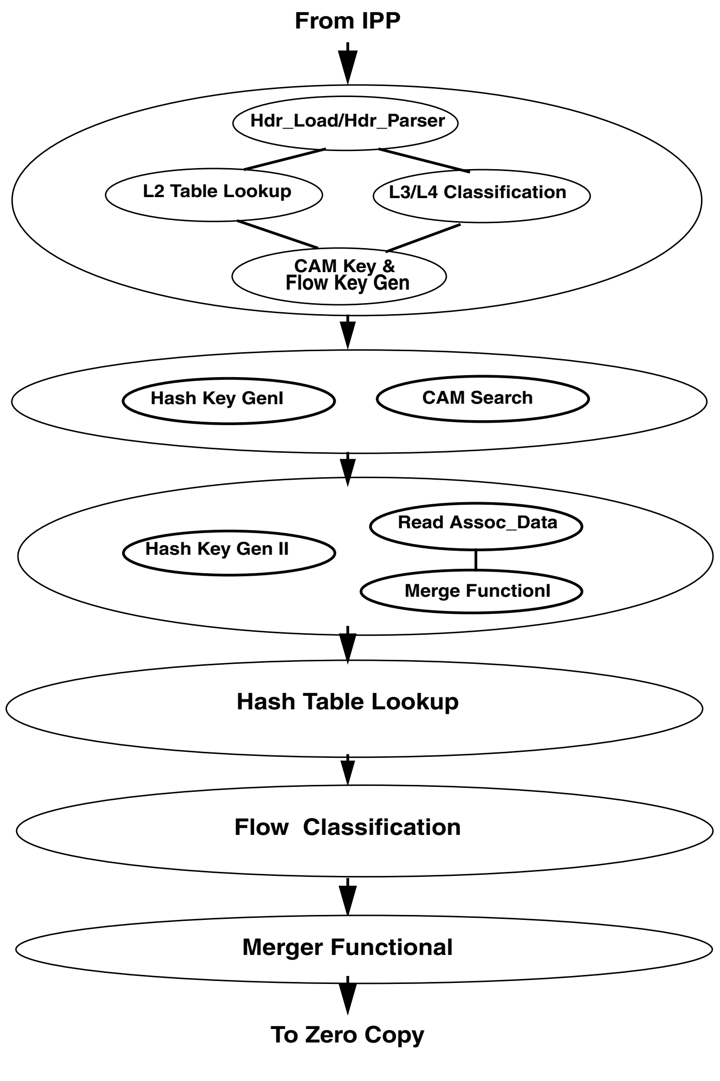
6.19.4 FFLP Microarchitecture Block

FIGURE 6-32 Sub-block Block Diagram with Interfacing Sub-block



6.19.5 Major Pipeline Stages

FIGURE 6-33 FFLP Logic Pipeline Stages



The FFLP design is fully pipelined to help throughput. Each pipeline takes less than eight fixed cycles regardless of the packet size. An fifo is put in place to handle the handshaking Between different clock domain.

This fifo is also used as a data buffer in case there is temperate cycle mismatch between Pipeline stage4 and previous stages.

Pipeline Stage1: Header Processor (8 cycles)

The first stage of pipeline loads the packet header from IPP via four to eight cycles of data transfer between IPP and FFLP. The Header Processor includes three major functions: packet header parser, L2/L3/L4 packet classification and search key generations. Those functions as well as the load function are also pipelined.

Pipeline Stage2: TCAM Search (6 cycles)

AT this stage, TCAM search is performed based on the key generated by header processor. It also calculates hash key from flow key created by header Processor. In order to accommodate various latency from different vendors, a small fifo is placed in between pipeline stage1 and pipeline stage2.

Pipeline Stage3: TCAM Merge Function (6 cycles)

This stage includes read associated data based on the TCAM match index, merge the results from L2-L4 classification and TCAM search. The hash key calculation is finished at this stage. The merge function result goes into Fifo which handles the synchronization between different clock domain.

Pipeline Stage4: Hash Table Lookup (8 cycles @ 375Mhz)

This stage performs hash table lookup based on the hash key generated by previous stages. The hash table access is via a FCRAM controller which handles FCRAM access required protocol.

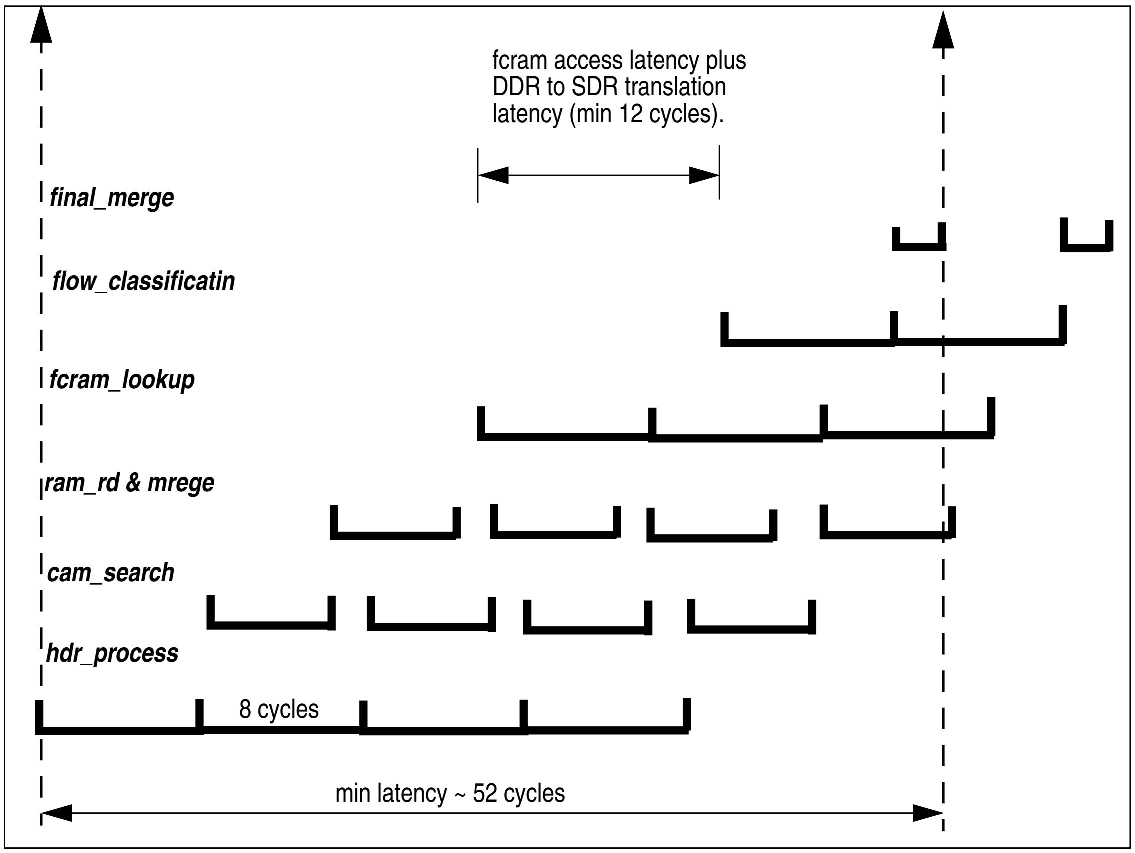
Pipeline Stage5: Flow Classification (8 cycles @ 375Mhz)

This stage processes the results from hash table lookups and determines if the hash results in exact match, or optimistic match, or no hash hit and generates the forward decision based on flow classification.

Pipeline Stage6: Merge Function II (2 cycles @ 375Mhz)

This stage generates final forward decision from flow classification and previous CAM search results. The result is written into ZCP.

FIGURE 6-34 FFLP Pipeline Stages



6.19.6 L2/L3/L4 Header Classification

6.19.6.1 Header Parsing

The header parsing function applies to the receive packets from IPPs. It is responsible for providing various header offsets to IPPs to do checksum calculation and other operations. it is also participating in creating the CAM lookup key as well as flow key. The header parsing block contains logic to shift, coalesce and accumulate packet header sixteen bytes as they are read out of each IPP FIFO according to programmable parameters, options and packet header types.

The parser detects various encapsulations and protocols including:

- - 802.1Q (VLAN-tagged) and LLC SNAP encapsulations.
- - IPv4, IPv6 and IPv4 header options.
- - IPsec on IPv4, ESP or AH.
- - IPsec on IPv6, ESP or AH.
- - TCP, UDP, STCP header.

IPv4 headers are 20 bytes each without IP header options and IPv6 headers are 40 bytes each. The header parser does not attempt to parse IP header options other than IPsec. It skips the option field and move to the next header. It does not process IP fragments beyond the IP header.

The TCP header is detected based on the protocol or next header field of the IP header identified by the parser function. It provides TCP header information, such as TCP sequence number, TCP SYN, FIN flags, to ZCP block to assist zero copy function. The parser does not process any TCP header options. It only uses the header length value to locate the starting point of TCP segment payload for the purpose of checksum calculations.

The CAM search key as well as flow key are formed from accumulated packet header fields combining with the results from various classifications.

6.19.6.2 L2 Header Classification

The L2 header classification happens when FFLP reads in the first cycle of 16 bytes of data from IPP.

The control information sent out by MAC tells L2 classifier if the MAC has an address match. In case if the packet does not have MAC address match, the per port default RDC table number will be used.

In the meantime, if Header parser determines the received packet is a VLAN tagged packet after processing the Ethernet header, the VLAN ID is used to lookup into a VLAN table to select L2 VLAN based RDC table number.

Based on the figure2-11, the L2 merge function decides the preference between MAC RDC table number and VLAN RDC table if the packet is VLAN tagged packet. If the packet is not VLAN tagged packet, the MAC RDC table number will be used as L2 RDC table number and passed to the Merge Functional to participate further RDC table number selection.

6.19.6.3 L3/L4 Header Classification

The L3/L4 header classification starts when the header parser identifies the incoming packet L2/L3 packet type. It could start as early as the third set of 16 bytes data been read into FFLP. In fact, the header classification process and header parsing are going in parallel to shorten the cycles.

The CAM lookup key generation as well as the flow lookup key generation use the concept of classes of packets to assemble a reasonable size of key. With this CAM key, a packet goes through a single CAM lookup for an associative search. The L2 class is determined based on the EtherType value. The L3 class is designed exclusively for IP packets. It is obtained by matching a number of packet header fields against 12 possible L3 class filter entries. The header fields which are selected for L3 class match include Protocol ID/Next Head and TOS byte. A set of bits mask are also used to allow certain bits in header bit pattern not participating in header matching. The class matched determines if further CAM classification or flow classification is needed and how the CAM key and flow key are assembled. The five bits class filter index which indicates matched class becomes part of CAM lookup key, and the lowest matched index has the highest priority if multiple classes are matched.

There are ten classes which are hardwired, the rest are software programmable. The hardwired classes start at index 8 in the class filter table.

The FFLP assigns class 00000 to the packet which does not match any class in class filter. Class 00001, 10010-11111 are reserved for dummy class used by software or testing.

TABLE 6-13 Class Code

Class Code	Description	Detected by
00000	No class match	
00001	Dummy Class.	
00010	class_reg2[16:0]. Programmable L2 Class.	class_reg2[16] == 1, pkt_EtherType == l2_class_reg2[15:0]
00011	class_reg3[16:0]. Programmable L2 Class	class_reg3[16] == 1, pkt_EtherType == l2_class_reg3[15:0]
00100	class_reg4[25:0] Programmable L3 Class	class_reg4[25] == 1, (class valid bit) v4: class_reg4[24] == 0, v6: class_reg4[24] == 1, protocol/next_hdr == class_reg4[23:16], class_reg4[15:8]: mask bits for TOS field. pkt_TOS == class_reg4[7:0], if per mask bit is set.

TABLE 6-13 Class Code *(Continued)*

Class Code	Description	Detected by
00101	class_reg5[25:0] Programmable L3 Class	class_reg5[25] == 1, (class valid bit) v4: class_reg5[24] == 0, v6: class_reg5[24] == 1, protocol/next_hdr == class_reg5[23:16], class_reg5[15:8]: mask bits for TOS field. pkt_TOS == class_reg5[7:0], if per mask bit is set.
00110	class_reg6[25:0] Programmable L3 Class	class_reg6[25] == 1, (class valid bit) v4: class_reg6[24] == 0, v6: class_reg6[24] == 1, protocol/next_hdr == class_reg6[23:16], class_reg6[15:8]: mask bits for TOS field. pkt_TOS == class_reg6[7:0], if per mask bit is set.
00111	class_reg7[25:0] Programmable L3 Class	class_reg7[25] == 1, (class valid bit) v4: class_reg7[24] == 0, v6: class_reg7[24] == 1, protocol/next_hdr == class_reg7[23:16], class_reg7[15:8]: mask bits for TOS field. pkt_TOS == class_reg7[7:0], if per mask bit is set.
01000	TCP over IPv4 Hardwired L3 class	Ethertype == 0x0800, Version == 4, Protocol == 0x06, No fragment.
01001	UDP over IPv4 Hardwired L3 class	Ethertype == 0x0800, Version == 4, Protocol= 0x11, No fragment|.
01010	AH or ESP over IPv4 Hardwired L3 class	Ethertype == 0x0800, Version == 4, Protocol == 0x33 or 0x32, no fragment|.
01011	STCP over IPv4 Hardwired L3 class	Ethertype == 0x0800, Version == 4, Protocol == 0x06, No fragment.

TABLE 6-13 Class Code *(Continued)*

Class Code	Description	Detected by
01100	TCP over IPv6 Hardwired L3 class	Ethertype == 0x86dd, Version == 6, Next_Hdr == 0x06.
01101	UDP over IPv6 Hardwired L3 class	Ethertype == 0x86dd, Version == 6, Next_Hdr == 0x11.
01110	AH or ESP over IPv6 Hardwired L3 class	Ethertype == 0x86dd, Version == 6, Next_hdr == 0x33 or 0x32
01111	STCP or IPv6 Hardwired L3 class	Ethertype == 0x86dd, Version == 6, Next_Hdr == 0x11.
10000	ARP	Ethertype == 0x0806
10001	RARP	Ethertype == 0x8035
10010 - 11111	Dummy Class	

Note that the IP header TOS field is not needed for L3 hardwired class comparison.

The hardwired classes are always enabled. Whereas the programmable classes can be disabled by setting the valid bit to zero. Furthermore, each L3 class filter is associated with two class action register which set the rule to form the CAM key and flow key. The packet which belongs to particular L3 class can also be filtered out by programming class action register.

6.19.7 TCAM Classification

6.19.7.1 Associative Memory Organization

The FFLP classification logic, in collaboration with management software, maintains a combined L3/L4 packet header fields based classification database. This database is stored in an on-chip associative memory (CAM-RAM) and contains information for making real-time packet matching, filtering and processing decisions. Theoretically, the entire address space could be supported by the database. However, in practice, only a small given subset of this address space is assumed to be active at a time. In addition, when the search criteria is based on L3 (IP) header information, it would be based on active IP flows. The number of active flows is limited at any given time, especially with flow aggregation.

FFLP CAM-RAM is designed to support total of 128 (or 256) simultaneously active flows at any given time. However, the supported flow space is not a hard limit, since the physical size of the CAM is transparent to FFLP. If deeper CAMs are available, the flow spaces increase. Newly arrived entries can replace inactive existing entries in the CAM-RAM. When the associative memory is full, software invalidates old entry and replace it by new entry. In any cases, the associative memory full is transparent to FFLP.

To be able to support both IPv4 and IPv6 packets, the CAM is configured to 200 bits key.

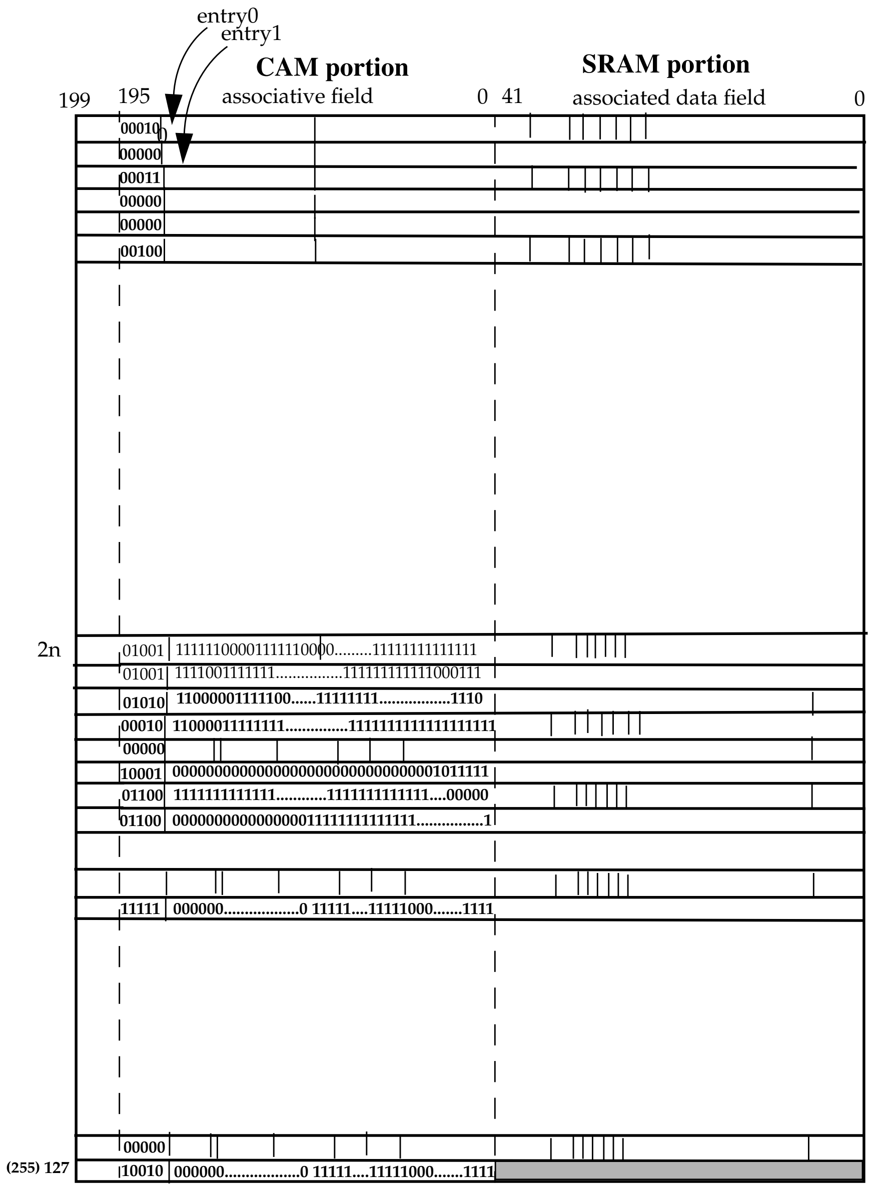
The CAM-RAM is made up of two portions: a fully associative CAM portion made up of L3/L4 header and an associated data SRAM. Figure 2-10 shows the logical organization of the CAM-RAM.

Each CAM line is 200 bits. A active entry contains any of the following fields. #1 is already present:

1. a five bit class denoting the type of class.
2. a five bit RDC number from L2 classification.
3. 1 bit no_ports to avoid erroneous matching of non-standard TCP/IP headers.
4. 8 bit protocol/next header field.
5. 8 bit TOS field.
6. IP destination address, IP source address, 16 bit destination port number and 16 bit source port number.
7. 7). 32 bit SPI from IPSec packet.

Each SRAM line contains information like which RDC the packet should be sent to, if the packet needs flow classification, control bits, and information to Zero copy.

FIGURE 6-35 Associative Memory (CAM-RAM) Logical Organization



6.19.7.2 Search Key & Search Execution

The class filter match determines how the search key is assembled. The search key for each connection is generated from extracting certain fields from IP header and TCP header combined with the header class that is from the result of packet classification. The fields which are extracted from IP header and TCP header (key elements) include:

- IPv4 protocol ID/IPv6 next header,
- IP source or/and destination address (4-tuple or 5-tuple key),
- TOS field,
- TCP source and destination port number/SPI number for IPSec.

With each class identified, there is a programmable register to tell how the key should be built. For IPv4 packets, the keys are always 5-tuple key. For IPv6 packets, only 4-tuple key is required with choice of IP source address or IP destination address.

TABLE 6-14 Class - Key Relationship

Class Code	TCAM Key	Flow Key
00000	No class match, TCAM search is not needed.	Same as TCAM key
00001	Dummy class, Software use only. No key generated by fflp.	No flow key generated
00010	First 11 bytes after EtherType	No flow key generated
00011	First 11 bytes after EtherType	No flow key generated

00100 - 01111	Based on key elements. User programmable register determines: IP src/dest address.
	IPADDR= 1 -> source addr,
	IPADDR= 0 -> destination addr
		Based on MAC DA, VLAN_ID, IP address, TCP port number, Protocol_ID, port_ID and others. User programmable register tells which elements are selected as part of key and which are not.

10000	First 11 bytes after EtherType	No flow key generated
10001	First 11 bytes after EtherType	No flow key generated
10010 - 11111	Dummy class, Software use only. No key generated by fflp.	No flow key generate

The header parser detects if the incoming packet is IPv4, IPv6 or other packet type. Those information combined with packet classification as well as programming resource are used to determine whether IP source address or IP destination address is required for this particular packet.

Please note, in order to differentiate the IPv4 IP fragment packets, a bit “NOPORT” is used in the key format to indicate the TCP port field is invalid and should be masked out. The NOPORT definition is stated as:

NOPORT = (Fragment_bit == 1) | (Fragment_offset[12:0]!= 0);

The following is the list of various TCAM search keys:

5-Tuple IPv4/TCP (UDP)

1.	key[199:195]:	header class index (5 bits)
2.	key[194:190]:	reserved, 5’b0
3.	key[189:187]:	L2 DRC table number (3 bits)
4.	key[186]:	noport (1 bit)
5.	key[185:112]:	reserved, 74’b0
6.	key[111:104]:	TOS (8 bits)
7.	key[103:96]:	protocol (8 bits)
8.	key[95:80]:	TCP source port number (16 bits)
9.	key[79:64]:	TCP destination port number (16 bits)
10.	key[63:32]:	IPv4 source address (32 bits)
11.	key[31:0]:	IPv4 destination address (32 bits)

5-Tuple IPv4/IPSec

1.	key[199:195]:	header class index (5 bits)
2.	key[194:190]:	reserved, 5’b0
3.	key[189:187]:	L2 DRC table number (3 bits)
4.	key[186]:	noport (1 bit)
5.	key[185:112]:	reserved, 74’b0
6.	key[111:104]:	TOS (8 bits)
7.	key[103:96]:	protocol (8 bits)
8.	key[95:64]:	SPI (32 bits)

9.	key[63:32]:	IPv4 source address (32 bits)
10.	key[31:0]:	IPv4 destination address (32 bits)

4-Tuple IPv6/TCP (UDP)

1.	key[199:195]:	header class index (5 bits)
2.	key[194:190]:	reserved, 5’b0
3.	key[189:187]:	L2 DRC table number (3 bits)
4.	key[186]:	reserved, 1’b0
5.	key[185:176]:	reserved, 10’b0
6.	key[175:168]:	TOS (8 bits)
7.	key[167:160]:	next header (8 bits)
8.	key[159:144]:	TCP source port number (16 bits)
9.	key[143:128]:	TCP destination port number (16 bits)
10.	key[127:0]:	IPv6 source or destination address (128 bits)

4-Tuple IPv6/IPSec

1.	key[199:195]:	header class index (5 bits)
2.	key[194:190]:	reserved, 5’b0
3.	key[189:187]:	L2 DRC table number (3 bits)
4.	key[186]:	reserved, 1’b0
5.	key[185:176]:	reserved, 10’b0
6.	key[175:168]:	TOS (8 bits)
7.	key[167:160]:	next header (8 bits)
8.	key[159:28]:	SPI (32 bits)
9.	key[127:0]:	IPv6 source or destination address (128 bits)

EtherType and others:

1.	key[199:195]:	header class index (5 bits)

2. key[194:192]: reserved, 3'b0
3. key[191:104]: first 11 bytes after EtherType {11th,... second, first}
4. key[103:0]: reserved, 104'b0

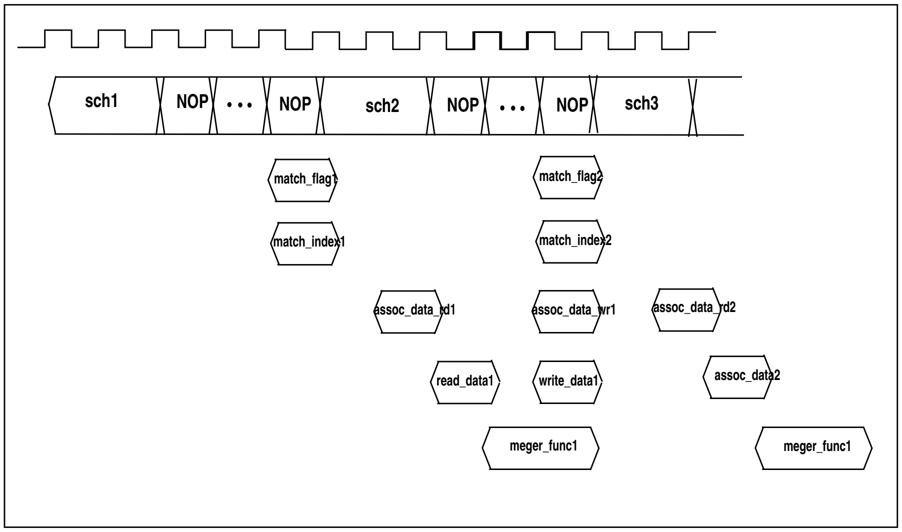
If there is a CAM match, the CAM match index will be used as address to read associated RAM. The associated data contains TCAM classification result that includes:

- If flow (hash table) lookup is required,
- TCAM RDC table index,
- TCAM RDC table offset number,
- Which RDC table number to use, L2 or TCAM,
- Zero copy information
- If the packet should be filtered.

The TCAM classification result as well as L2/L3/L4 registered based classification result are sent to a Merge Function for further data processing.

In addition, the FFLP performs searches/updates into the CAM on behalf of the software user. All requests will go through an arbitration mechanism.

FIGURE 6-36 Search-update CAM Cycle with associated SRAM



CAM Access Scheduler

The scheduler arbitrates the requests from both CPU management access and packet cam lookup. For current TCAM, the cam comparison is very fast - the search/comparison command can be issued every five cycles. Because of that, the CPU request will be granted almost right away once the request reaches fflp. On the other hand, FFLP is designed to support bigger CAM (if it is needed later on), and the CAM search rate could be lower. So the flexibility is desirable for the cam access schedule. In general, if needed, more bandwidth can be reserved for packet cam lookup than CPU access. The ration between them is programmable through register (fflp config register). Note: if the number is programmed to 0 (which is the default value), FFLP will always give high priority to packet cam lookup.

Please note, the CPU request for CAM read does not consume any cam bandwidth.

CAM Bandwidth Considerations

The dead cycles (access time) between the CAM key presented to the CAM and the match flag becoming valid varies from different vendors. Especially the CAM depth can be increased to up to 1K entries to meet the requirement of certain applications. The design is implemented in such way that it is not sensitive to the number of dead cycles as long as it is within reasonable range and it is at least be able to support two 10G and two 1G ports.

The CAM runs at 375MHz, same as the core clock. To support back to back minimum size (64 bytes) packets at aggregate packet rate of 20Gb/s, the CAM lookup rate has to be (at least) 10 cycles per packet. In order to leave enough bandwidth to CUP management access, the number of cycles should be much less than 10 cycles.

Software Access to CAM **Algorithm**

Major Software commands:

FFLP provides a command interface via bus PIOs to software for accessing the CAM. The command interface supports a minimum set of basic commands and a set of key and data registers to support the following functions:

Opcode: 3’b000: write CAM key and write CAM mask.

Opcode: 3’b001: read CAM key and read CAM mask.

Opcode: 3’b010: CAM key compare.

Opcode: 3’b011: reserved.

Opcode: 3’b100: write associated data.

Opcode: 3’b101: read associated data.

Opcode: 3’b110, 111: reserved.

All software commands are atomic operations. All read/write commands are address counter based.

For write or compare command, software has to write data or key to proper registers before issuing write/compare command. Software command execution is triggered by software writing to CAM control/status register.

Since the internal CAM lookup is fairly fast and total CAM bandwidth is sufficient enough, the software command can be executed almost right way once the request has reached FFLP block.

Software always checks for command done status bit before issuing another command to make sure current command has been carried out by FFLP.

Aging and CAM Entry Invalidation

“Aging” is the process of time stamping entries and removing expired entries from the database. Aging helps reclaim inactive flow space for new flows and allows for more efficient database management. Setting the Aged bits is the responsibility of software. It can be achieved by writing the associated data with the appropriate Aged bit set using write command. Checking the Aged bits to see if they are still set when the corresponding Age timer expires is also the responsibility of software. Hardware clears the aged bit every time its packet lookup for the entry in the database results a CAM match.

There are two occasions when a CAM entry becomes invalid: 1) during power up, where all entries are marked invalid by the CAM initialization. 2) when software sends a command (in the form of a PIO write to a command register) to the hardware to perform invalidation of a specific entry.

Software is responsible for invalidating entries to make room for possible new entries. It does so via write CAM key with invalid class code (0x0) or dummy classes. If software wants to replace an old entry with new one, it can first invalidate that entry, following by writing new key via write command.

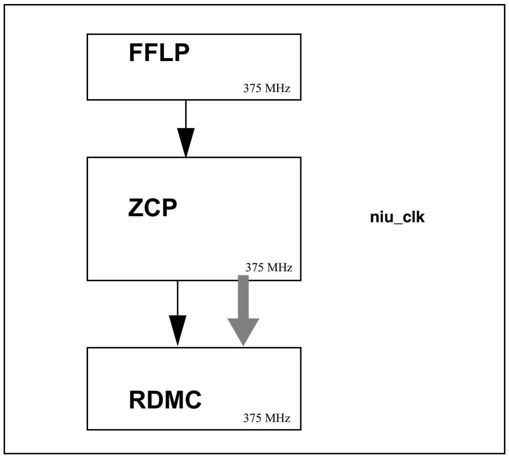
6.20 ZCP Microarchitecture

6.20.1 ZCP Overview

ZCP contains two control FIFOs. One for each port. The control FIFO (CFIFO) is used to deposit forwarding decision which is generated by FFLP. A shared RDC table is used to lookup the Receive DMA channel number. The main consumer of CFIFO and RDC table is RDMC.

ZCP operates in niu clock domain. There is one clock domain.

FIGURE 6-37 ZCP and Interfacing Blocks Block Diagram



6.20.2 ZCP Interface Signals

TABLE 6-15 ZCP Top Level Interface Signals

Signal Name	Width	Direction	Description
FFLP-ZCP			
fflp_zcp_wr	5	I	For timing issue, the write enable is driven out as a five bit bus. All five bits have the same function and waveform.
fflp_zcp_data	216	I	FFLP post forwarding decision to ZCP control FIFO.
PORT0 ZCP-RDMC			
dmc_zcp_req0	1	I	Rx dmc request data from ctrl_fifo. Rx dmc uses this signal to advance control fifo read pointer.
zcp_dmc_ack0	1	O	Valid data. Rx dmc should use this signal to qualify valid data. This signal is asserted three clocks after dmc_zcp_req is asserted.
zcp_dmc_dat0	130	O	ctrl_fifo's data to rxdma
zcp_dmc_dat_err0	1	O	Uncorrectable ECC error. It has timing as the corresponding data.
zcp_dmc_ful_pkt0	1	O	ctrl_fifo has at least one full packet. When it is asserted, the first control information of the packet is ready.
PORT1 ZCP-RDMC			
dmc_zcp_req1	1	I	Rx dmc request data from ctrl_fifo. Rx dmc uses this signal to advance control fifo read pointer.
zcp_dmc_ack1	1	O	Valid data. Rx dmc should use this signal to qualify valid data. This signal is asserted three clocks after dmc_zcp_req is asserted.
zcp_dmc_dat1	130	O	ctrl_fifo's data to rxdma
zcp_dmc_dat_err1	1	O	Uncorrectable ECC error. It has timing as the corresponding data.
zcp_dmc_ful_pkt1	1	O	ctrl_fifo has at least one full packet. When it is asserted, the first control information of the packet is ready.

6.20.3 ZCP Microarchitecture Block

FIGURE 6-38 ZCP Block Diagram

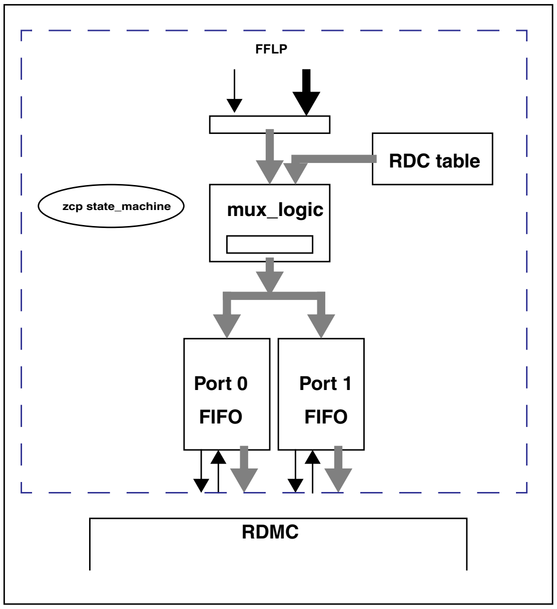
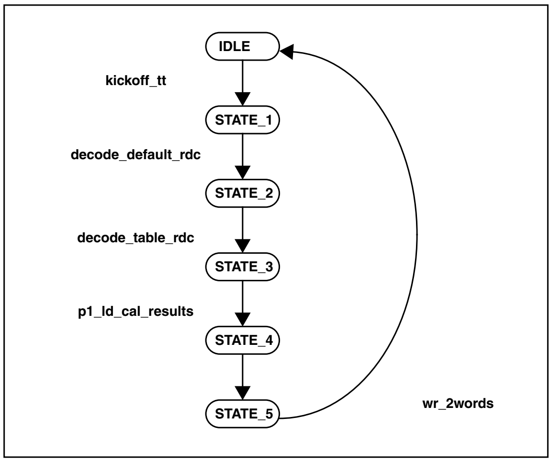


FIGURE 6-39 ZCP State Machine.



6.20.4 RDC Table Microarchitecture

The RDC table is 4 bit wide and 128 deep. Each table entry is used to store the Receive DMA channel number. It is logically organized as eight tables. Each table is 16 deep.

6.20.5 ZCP State Machine Microarchitecture

The ZCP state machine is used to pace the RDC table look up and control FIFO loading.

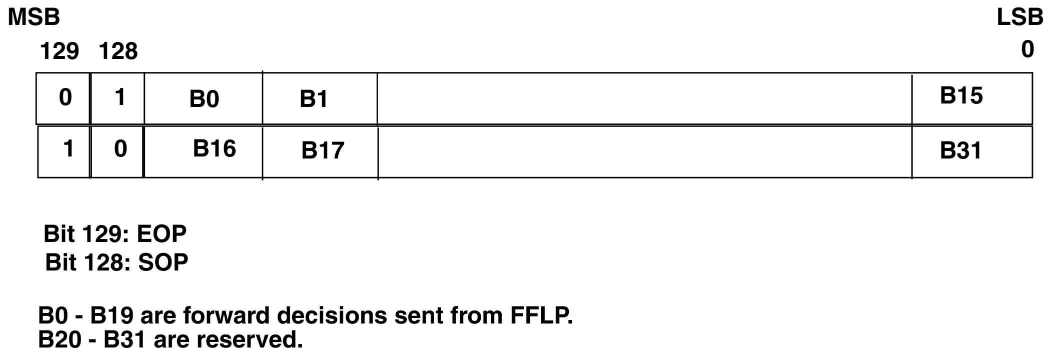
6.20.6 ZCP Control FIFO Microarchitecture

The ZCP control FIFO module is housing a SRAM based FIFO with zcp-rdmc interface protocol logic. The SRAM is protected by single error correction and double error detection ECC.

6.20.7 ZCP - RDMC Interface Data Format

Each packet occupies two entries in CFIFO. The data format is shown in FIGURE 6-40.

FIGURE 6-40 Data Format



6.20.8 ZCP FIFO Memory Configuration

ZCP only has two per port control FIFO memory. All previous translation table SRAM are removed.

TABLE 6-16 ZCP FIFO Memory Configuration

Memory NAME	Logical Dimension D x W	RAS	D x W with RAS	Size	Physical Configuration (D x W)
Port 0 Control FIFO	2K x 16B	ECC	1664 x 18B (1664 x 146bit)	30KB	One 1664x72b + one 1664x74b
Port 1 Control FIFO	2K x 16B	ECC	1664 x 18B (1664 x 146bit)	30KB	One 1664x72b + one 1664x74b

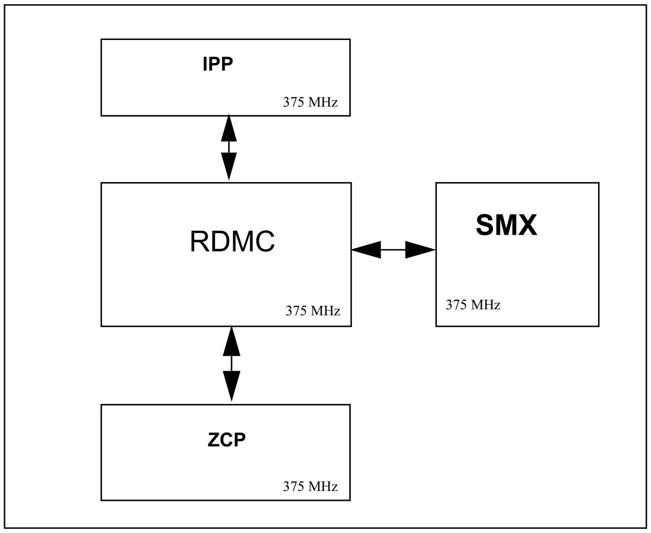
6.21 RDMC Microarchitecture Specification

6.21.1 RDMC Overview

The main functions of the RDMC are:

1. Support 16 DMA channels.
2. Descriptor prefetch and buffer allocation and buffer management to support different packet sizes as well as jumbo frame.
3. Completion write back logic maintains a RCR shadow buffer to reduce overhead of updating the RCR to the system RAM
4. Port scheduler with Deficit Round Robin algorithm to guarantee no starvation between 1Gbps and 10Gbps ports.
5. Full data alignment logic to support zero copy mode.
6. Support Weighted RED buffer management scheme per channel based on RDC state.

FIGURE 6-41 RDMC and Interfacing Blocks Block Diagram



6.21.2 RDMC Interface Signals

TABLE 6-17 Top Level Interface Signals

Class Code	Description	Detected by	
IPP-RDMC:			
dmc_ipp_dat_req0	1	I	Rdmc requests data from ipp_dat_fifo_0
ipp_dmc_dat_ack0	1	O	If 1, validates ipp_dat_fifo_0 data to rdmc
ipp_dmc_data0	130	O	ipp_dat_fifo_0's data to rdmc
ipp_dmc_ful_pkt0	1	O	If 1, ipp_dat_fifo_0 has complete packet/s
ipp_dmc_dat_err0	1	O	If 1, ipp_dat_fifo_0 data has error
dmc_ipp_dat_req1	1	I	Rdmc requests data from ipp_dat_fifo_1
ipp_dmc_dat_ack1	1	O	If 1, validates ipp_dat_fifo_1 data to rdmc
ipp_dmc_data1	130	O	ipp_dat_fifo_1's data to rdmc
ipp_dmc_ful_pkt1	1	O	If 1, ipp_dat_fifo_1 has complete packet/s
ipp_dmc_dat_err1	1	O	If 1, ipp_dat_fifo_1 data has error
ZCP-RDMC			
dmc_zcp_req0	1	I	Rx dmc request data from ctrl_fifo. Rx dmc uses this signal to advance control fifo read pointer.
zcp_dmc_ack0	1	O	Valid data. Rx dmc should use this signal to qualify valid data. This signal is asserted three clocks after dmc_zcp_req is asserted.
zcp_dmc_dat0	130	O	ctrl_fifo's data to rxdma
zcp_dmc_dat_err0	1	O	Uncorrectable ECC error. It has timing as the corresponding data.
zcp_dmc_ful_pkt0	1	O	ctrl_fifo has at least one full packet. When it is asserted, the first control information of the packet is ready.
dmc_zcp_req1	1	I	Rx dmc request data from ctrl_fifo. Rx dmc uses this signal to advance control fifo read pointer.
zcp_dmc_ack1	1	O	Valid data. Rx dmc should use this signal to qualify valid data. This signal is asserted three clocks after dmc_zcp_req is asserted.
zcp_dmc_dat1	130	O	ctrl_fifo's data to rxdma
zcp_dmc_dat_err1	1	O	Uncorrectable ECC error. It has timing as the corresponding data.

TABLE 6-17 Top Level Interface Signals *(Continued)*

Class Code	Description	Detected by	
zcp_dmc_ful_pkt1	1	O	ctrl_fifo has at least one full packet. When it is asserted, the first control information of the packet is ready.
PIM-RDMC			
meta0_rdmc_wr_req_accept	1	I	rdmc write request accepted.
meta0_rdmc_wr_data_req	1	I	meta write data request.
meta0_rdmc_rcr_req_accept	1	I	rdmc rcr write back request accepted.
meta0_rdmc_rcr_data_req	1	I	meta rcr write data request.
meta0_rdmc_rcr_ack_ready	1	I	non-posted write ack for rcr write back
meta0_rdmc_rcr_ack_cmd	8	I	rcr write back ack command
meta0_rdmc_rcr_ack_cmd_status	4	I	rcr write back ack command status.
meta0_rdmc_rcr_ack_client	1	I	rcr write back ack client ID
meta0_rdmc_rcr_ack_dma_num	5	I	rcr write back ack dma number
meta1_rdmc_rbr_req_accept	1	I	rdmc read request accepted
meta1_rdmc_rbr_req_error	1	I	rdmc read request error
meta1_rdmc_rbr_resp_ready	1	I	rdmc read request response ready
meta1_rdmc_rbr_resp_cmd	8	I	rdmc read request response command
meta1_rdmc_rbr_resp_cmd_status	4	I	rdmc read request response status
meta1_rdmc_rbr_resp_dma_num	5	I	rdmc read request response dma number
meta1_rdmc_rbr_resp_client	1	I	rdmc read request response client
meta1_rdmc_rbr_resp_comp	1	I	rdmc read request response completed

TABLE 6-17 Top Level Interface Signals *(Continued)*

Class Code	Description	Detected by	
meta1_rdmc_rbr_resp_trans_comp	1	I	rdmc read request response transfer completed
meta1_rdmc_rbr_resp_data_valid	1	I	rdmc read request response data valid
meta1_rdmc_rbr_resp_data	128	I	rdmc read request response data
meta1_rdmc_rbr_resp_byteenable	16	I	rdmc read request response byte enable
meta1_rdmc_rbr_resp_data_status	4	I	rdmc read request response data status
rdmc_meta0_wr_req	1	O	rdmc write request
rdmc_meta0_wr_req_cmd	8	O	rdmc write request command
rdmc_meta0_wr_req_address	64	O	rdmc write request address
rdmc_meta0_wr_req_length	14	O	rdmc write request length
rdmc_meta0_wr_req_port_num	2	O	rdmc write request port number
rdmc_meta0_wr_req_dma_num	5	O	rdmc write request dma number
rdmc_meta0_wr_req_func_num	2	O	rdmc write request function number
rdmc_meta0_wr_data_valid	1	O	rdmc write data valid
rdmc_meta0_wr_data	128	O	rdmc write data
rdmc_meta0_wr_req_byteenable	16	O	rdmc write data byte enable
rdmc_meta0_wr_transfer_comp	1	O	rdmc write data transfer completed
rdmc_meta0_wr_status	4	O	rdmc write data status
rdmc_meta0_rcr_req	1	O	rdmc rcr write back request
rdmc_meta0_rcr_req_cmd	8	O	rdmc rcr write back request command
rdmc_meta0_rcr_req_address	64	O	rdmc rcr write back request address

TABLE 6-17 Top Level Interface Signals *(Continued)*

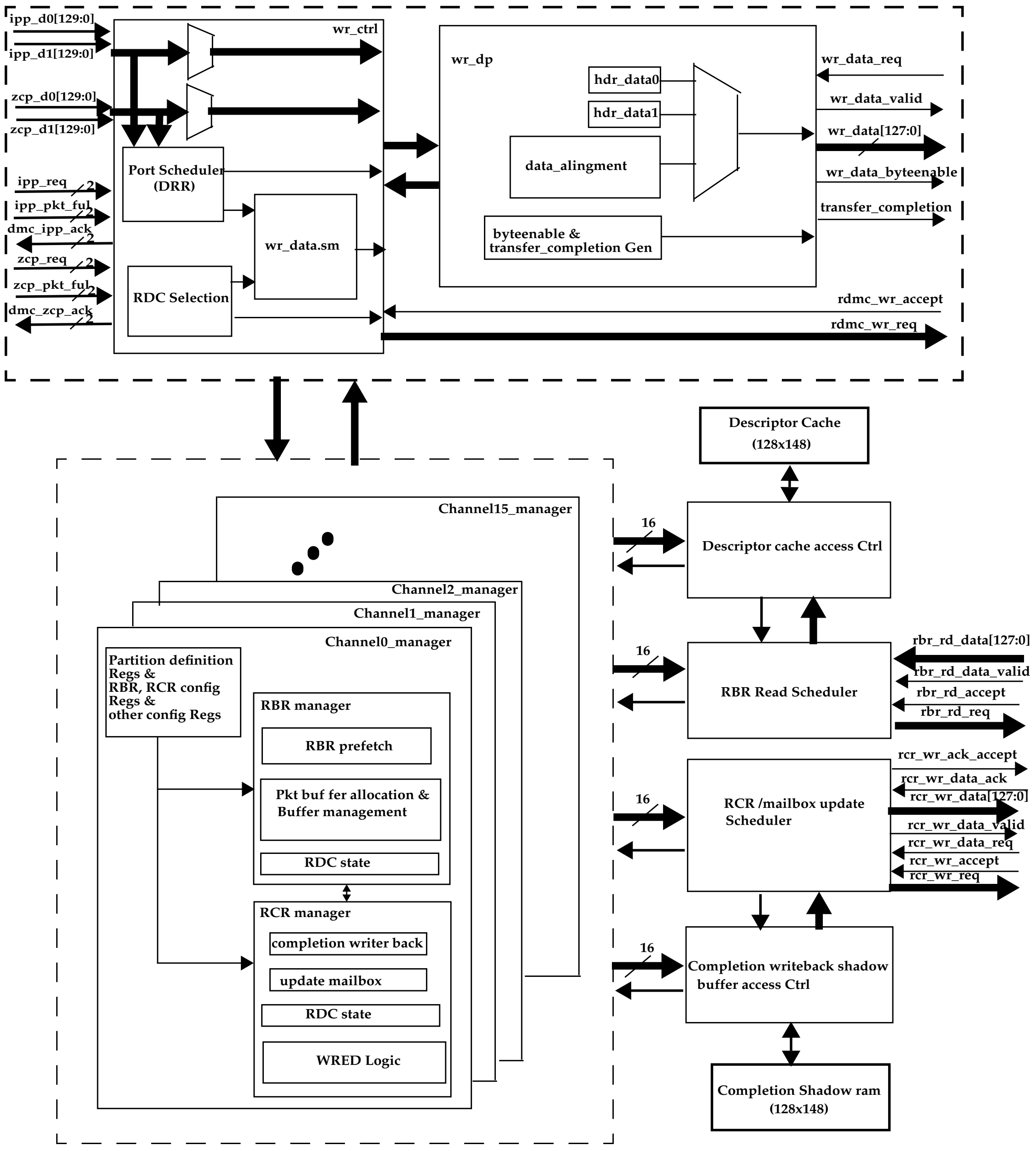
Class Code	Description	Detected by	
rdmc_meta0_rcr_req_length	14	O	rdmc rcr write back request length
rdmc_meta0_rcr_req_port_num	2	O	rdmc rcr write back request port number
rdmc_meta0_rcr_req_dma_num	5	O	rdmc rcr write back request dma number
rdmc_meta0_rcr_req_func_num	2	O	rdmc rcr write back request function number
rdmc_meta0_rcr_data_valid	1	O	rdmc rcr write back data valid
rdmc_meta0_rcr_data	128	O	rdmc rcr write back data
rdmc_meta0_rcr_req_byteenable	16	O	rdmc rcr write back data byte enable
rdmc_meta0_rcr_transfer_comp	1	O	rdmc rcr write back data transfer completed
rdmc_meta0_rcr_status	4	O	rdmc rcr write back data status
rdmc_meta0_rcr_ack_accept	1	O	rdmc rcr write back ack accepted
rdmc_meta1_rbr_req	1	O	rdmc rbr request
rdmc_meta1_rbr_req_cmd	8	O	rdmc rbr request command
rdmc_meta1_rbr_req_address	64	O	rdmc rbr request address
rdmc_meta1_rbr_req_length	14	O	rdmc rbr request length
rdmc_meta1_rbr_req_dma_num	5	O	rdmc rbr request dma number
rdmc_meta1_rbr_req_port_num	2	O	rdmc rbr request port number
rdmc_meta1_rbr_req_func_num	2	O	rdmc rbr request function number
rdmc_meta1_rbr_resp_accept	1	O	rdmc rbr read response accepted.

6.21.3 RDMC Interface Timing

Please see NIU_IPP Interface Timing, ZCP Interface Signals, NIU_PIO Interface Timing.

6.21.4 RDMC Microarchitecture Block

FIGURE 6-42 Sub-block Block Diagram with Interfacing Sub-block



6.21.5 Descriptor Cache & Descriptor Fetch

1. RDMC has on chip 256X148 cache for descriptor prefetch.
   - It can store maximum 32 block buffers per channel.
   - Each entry can store 4 block buffers at most. Each channel occupies eight entries.
   - Each buffer is represented by 32 bits, which is the higher order bits of 44 bits block address aligned to 4KB. One valid bit is associated to each block buffer.
   - The block size can be programmed to 4KB, 8KB, 16KB and 32KB.
2. The access to the descriptor cache is a flat round robin all for channels
3. Two conditions to trigger RDMC to assert descriptor fetch request.
   - There is at least one buffer available in RBR memory, and
   - The descriptor cache has five entry space available.
4. There is only one outstanding request per channel at any given time. Each DMA will not assert next descriptor read request before the current descriptor read response come back.
5. Each request is a 64 bytes read at most, and 4 bytes as a minimum which is one block buffer size. The read response can be split into two transactions since the read can be cross cacheline boundary. The RDMC does not do re-ordering if the second transaction comes back first.
6. When store data into descriptor cache, it writes 128 bits data along with the 4 valid bits and 16 parity bits into one entry. The writes always start from an new entry regardless of the previous entry having empty space or not.
7. The DMA state will be updated once the read response data are received. The DMA state includes RBR current buffer length and RBR head pointer.
8. The read request obeys Meta interface protocols.

Note – After channel reduction from 32 to 16. the rdmc only uses up half of the descriptor cache memory, which is 128 entries instead of 256 entries.

6.21.6 Packet Buffer Selection

1. The RDMC supports three packet buffer size programmed by software. Each buffer size has a valid bit associated with it.
   - size0: 256 bytes, 512 bytes, 1K, and 2K.

- size1: 1K, 2K, 4K, and 8K.
- size2: 2K, 4K, 8K, and 16K.

2. The channel manager reads one entry from descriptor cache per request and fetches in at most 4 block buffers. Then it partitions these block buffers into one of three packet buffer sizes, one at a time and it sets the size buffer available bit to 1 when partition is done.
3. There are six registers, two for each size buffer, to hold the current address and current available buffer number for that size. They will be updated once one buffer from that size is assigned to current incoming packet. When all buffers are used up from that size buffer, a fresh block buffer will be partitioned to that size buffer.
4. When all block buffers from one request are used up, the channel manage will assert an new request to access the descriptor cache if the descriptor cache is not empty.
5. The incoming packet size is compared against the three buffer sizes and block buffer size.
   - Packet size used to do comparison = packet_L2_len + control_header.
   - The control header is either 2 or 18 bytes programmable by software per dam channel.
   - The data offset has to be taken into consideration also.
   - The first buffer size that is larger than the packet size will be assigned to that packet.
   - For the packet size larger than the largest buffer size, up to three fresh block buffers can be used for that packet.
   - drop packet conditions related to select packet buffer:
   - Packet size is larger than three fresh block buffers;
   - There is no buffer available at time when packet arrives regardless of cache empty or not.
   - If the size buffers which are large enough to store the packet are not available;

The RDMC performs address translation for each buffer selected from the buffer pool for partition control.

6.21.7 Port Scheduler

1. Both IPP_dmc_data_full and zcp_dmc_data_full have to be high in order for port to be participated in port scheduling.
2. DRR is used to perform arbitration among two ports.

- Init. weight is programmable for all ports.
- The scheduler schedules one packet from one port at time and moves to next port as long as the current weight for the port is positive.
- The current weight is subtracted by one once a 16 bytes data is read out from data FIFO.
- Once the current weight for all ports becomes negative or the port which has positive weight does not have packet to schedule, the init. weight will be re-loaded to the current weight register.

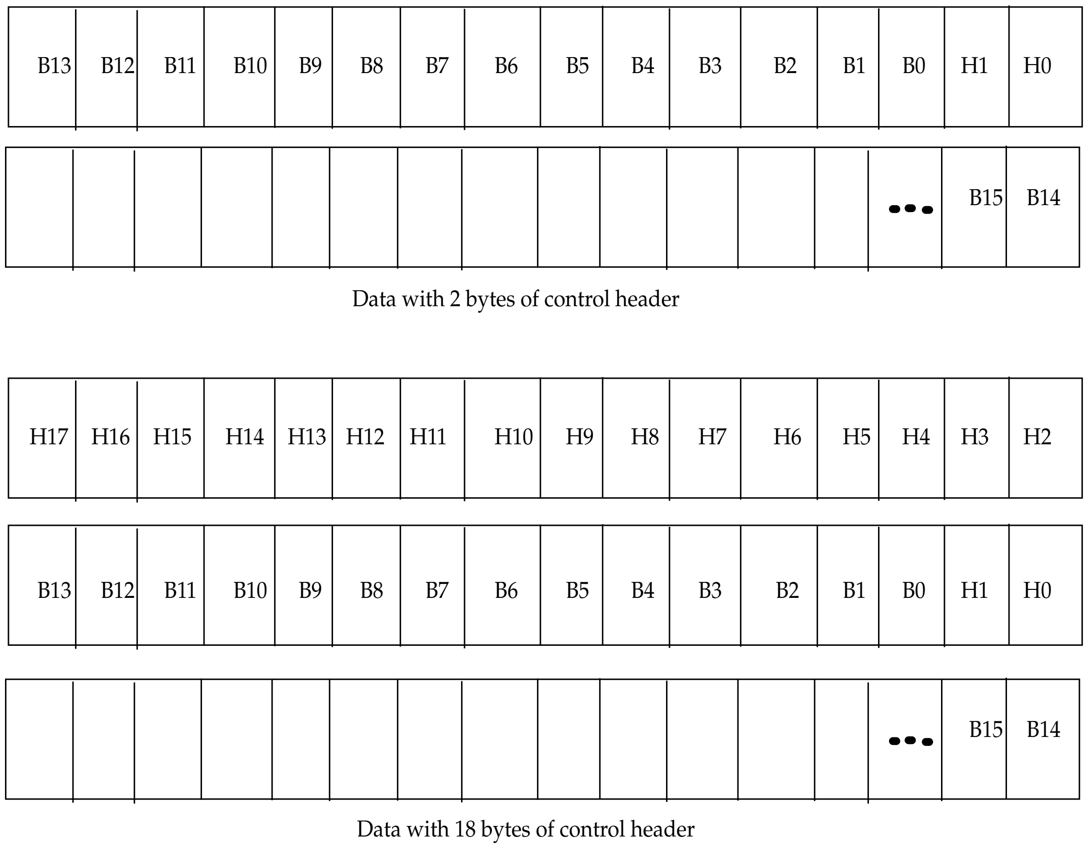
6.21.8 Packet Processing

1. After port scheduler, the RDMC reads control FIFO and the first word of data FIFO to get the control information for the packet by issuing two consecutive reads to ZCP.
2. Each packet will go through three major pipelines before being put on the meta interface bus.
   - Packet buffer selection.
   - Bus request (posted write request) and grant phase.
   - Data request phase and packet alignment.
3. Depends on when RDMC gets the bus_accept and data_request from Meta interface, RDMC can process back to back 64 bytes packet in about eight cycles if more than one port are active. If only one port is active, it takes about 18 cycles for RDMC to process one 64 bytes packet.
4. After RDMC receives data_req from meta interface, it asserts data_req to IPP to read data from data fifo. The IPP sends back data_ack after three cycles. The RDMC takes one cycle to do data alignment before asserting data_valid along with data to meta interface.
5. For jumbo frame, it takes up to three transactions to finish one packet transfer.

After alignment, the data_byteenable signal is always 16’hffff unless it is for the last chunk of data.

FIGURE 6-43 shows two alignment cases for bypass mode.

FIGURE 6-43 Packet Data Format


B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0 H1 H0
••• B15 B14
Data with 2 bytes of control header
H17 H16 H15 H14 H13 H12 H11 H10 H9 H8 H7 H6 H5 H4 H3 H2
B13 B12 B11 B10 B9 B8 B7 B6 B5 B4 B3 B2 B1 B0 H1 H0
••• B15 B14
Data with 18 bytes of control header

6.21.9 Completion Shadow RAM and Completion Write Back

1. The RDMC has on chip 256X148 RCR shadow RAM to reduce the overhead of per packet updating the RCR memory. Each shadow entry can store two sets of eight bytes completion write back data. The upper 16 bits are used as parity bits.

2. Each channel occupies 16 entries which contains 32 sets of completion write back data.

3. For normal packet which uses up only one packet buffer, the eight bytes completion write back data will be written into shadow RAM once the last set of data is put on the Meta bus.
4. For jumbo frame which uses more than one buffers, each transaction will have eight bytes completion write back data and it will be written into shadow RAM once the last set of data is put on the Meta bus. The mult-bit is set if it is not the last transaction for the packet.
5. Since the shadow ram is not per byte writable ram, to prevent the first set of completion data been overwritten, a temp register per channel is used to store the first set of completion data.
6. The shadow write pointer will be updated by one once two sets of completion data are written into one entry.
7. The shadow read pointer will be updated by eight once the full cacheline is read out from shadow memory.
8. A counter is used to count how many eight bytes space are available in shadow memory. The shadow fullness is determined by the counter. Packets will be dropped if shadow is full.
9. The completion write back to RCR is always full cacheline write regardless if there is full cacheline data in shadow ram or not. If it is not, the shadow read pointer will not be updated.
10. Completion write back to RCR memory triggers:
    - There are at least eight sets of completion write back data in shadow RAM, or
    - The completion write back timeout is expired and shadow is not empty, or
    - The RCR_QLEN is larger than the PKT_THRESH at the time when MEX is set and shadow is not empty.
    - The RCR flush bit in RCR flush register is set and shadow is not empty.
11. The WRED should be always enabled in order for RDMC not to overflow RCR memory. The WRED uses RCR QLEN as well as the THRESH and WINDOW, both are software programmable, as parameters to do the calculation to determine if the incoming packet should be dropped or not.
12. The completion write back to RCR is a non-posted write. The RDMC waits for the ack coming back before it updates all RCR related DMA state, such as QLEN and RCR tail pointer.
13. At any given time, there is only one outstanding non-posted write request per channel. The non-posted write request comes from RCR write back and mailbox update.
14. A flat round robin is used to grant write requests among all channels.

6.21.10 Mailbox Update

1. After each completion write back to RCR, the RDMC checks if the mailbox update is needed or not.
2. Mailbox update is triggered by
   - When there is completion write back timeout and MEX bit is set and the RCR_QLEN is not zero, or
   - When there is packet threshold crossing and MEX bit is set.
3. If hardware and software DMA states are in sync, RDMA does not do completion write back to RCR before it does mailbox update.
4. The rdmc will assert interrupt "ldf_a" to indicate that the mailbox update has happened and completed.

6.21.11 Drop Packet

1. Events to trigger dropping packets:
   - Packet buffer not available,
   - RCR Shadow RAM full,
   - fflp/ipp tells to drop packet,
   - WRED drop,
   - DMA not enabled when packet arrives,
   - Encounter fatal errors
2. Two counters per channel to count dropped packets.
   - Counter for WRED drop,
   - Shared counter to count dropped packet from the first three events.
   - Does not count the packet dropped by fatal errors or DMA not enabled.

6.22 TDMC Microarchitecture Specification

6.22.1 TDMC Overview

The Transmit DMA Controller (TDMC) block is the interface between the Transmit Control Engine (TXC) and the Meta Interface to the system memory. This block is responsible for managing the Transmit Descriptors. It also has a mechanism for caching descriptors and providing these to the TXC upon request.

The Transmit DMA Controller supports a total of 16 Transmit DMA Channels. Each Transmit DMA Channel is comprised of a Transmit Ring and a set of status registers. Software creates a list of packet descriptors for the Transmit Ring in system memory and informs the TDMC block about the corresponding pointers by updating the appropriate Control and Status Registers (CSRs). The TDMC block fetches the descriptors from memory and presents them to the TXC which then transmits the packets out.

The TDMC block also relocates the packet address to the appropriate page based upon the CSR values.

The packets are transmitted in the same order as they are posted in the ring. Each packet descriptor may also be structured as a gather list.

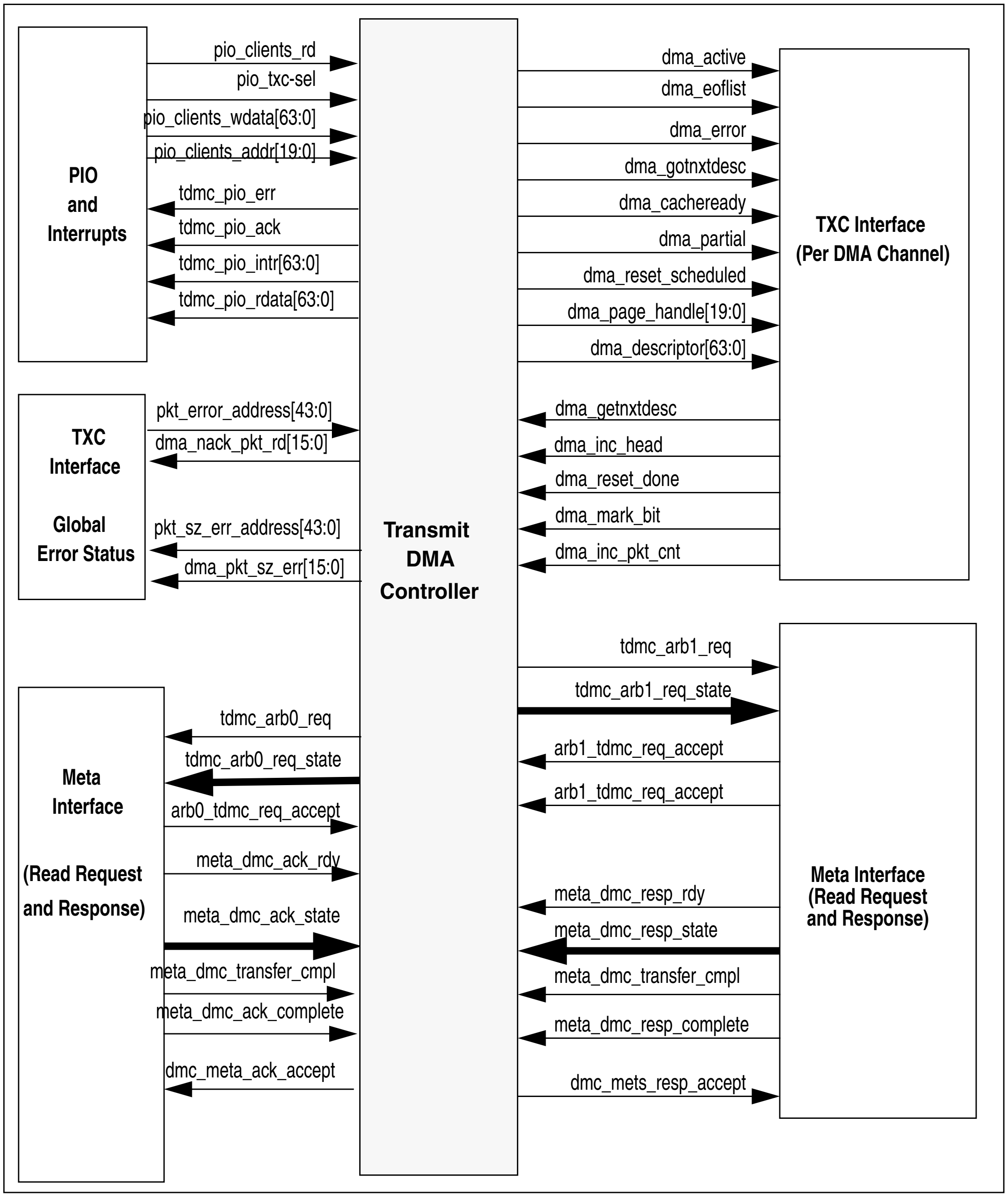
The TDMC Block updates the Mailbox in the system memory with the current state of the hardware when instructed by the software. The update occurs once the descriptor with the mailbox update instruction gets scheduled by the TXC. If necessary, an interrupt event would also be generated.

There are three main sub-blocks within this block:

- PIO Interface Block.
- Cache Fetch and Management Block.
- Mailbox Processing

Details about the functioning of these sub-blocks are given in subsequent sections.

FIGURE 6-44 TDMC Interface Diagram



6.22.2 TDMC Interface Signals

6.22.2.1 Transmit Controller Engine and Transmit DMA Interface

TABLE 6-18 TXC - TDMC DMA Cache Interface Signals

Signal Name	I/O	Size	From/To	Description
# is DMA Channel number				OpenSPARC T2: #= 16 channels, 0 through 15
txc_dmc_DMA#_getnextdesc	I	1	TXC->TDMC	Get next descriptor list
txc_dmc_DMA#_reset_done	I	1	TXC->TDMC	Indicates TXC is done with resetting/stopping the DMA#
txc_dmc_DMA#_inc_head_ptr	I	1	TXC->TDMC	Increment DMA#'s Head Pointer
txc_dmc_DMA#_update_mbox	I	1	TXC->TDMC	Update Mailbox for DMA#
txc_dmc_DMA#_mark_bit	I	1	TXC->TDMC	Indicates mark bit set for DMA#
dmc_txc_DMA#_active	O	1	TDMC->TXC	DMA # is active
dmc_txc_DMA#_descriptor	O	64	TDMC->TXC	DMA# descriptor
dmc_txc_DMA#_eoflist	O	1	TDMC->TXC	DMA# end of descriptor list
dmc_txc_DMA#_error	O	1	TDMC->TXC	DMA# error
dmc_txc_DMA#_gotnxtdesc	O	1	TDMC->TXC	DMA# got next descriptor list
dmc_txc_DMA#_page_handle	O	20	TDMC->TXC	Relocation handle for packet address
dmc_txc_DMA#_func_num	O	2	TDMC->TXC	Function number to be sent along with requests to host memory.
dmc_txc_DMA#_cache_ready	O	1	TDMC->TXC	DMA# Cache Ready
dmc_txc_DMA#_partial	O	1	TDMC->TXC	DMA# Indicating Partial descriptors available
dmc_txc_DMA#_reset_scheduled	O	1	TDMC->TXC	DMA# Indicating software reset/stop is scheduled

6.22.2.2 Transmit Controller Engine and Transmit DMA Error Interface

TABLE 6-19 TXC- TDMC Error Interface Signals

Signal Name	I/O	Size	From/To	Description
txc_dmc_dma_nack_pkt_rd	I	16	TXC->TDMC	One- Hot Encoded signal indicating the DMA number for which the error is being reported
txc_dmc_nack_pkt_rd_addr	I	44	TXC->TDMC	Error Address for packets which timeout
txc_dmc_nack_pkt_rd	I	1	TXC->TDMC	Pulse indicating Timeout Error
txc_dmc_p0_dma_pkt_size_err	I	16	TXC->TDMC	One- Hot Encoded signal indicating the DMA number for which the error is being reported. This is reported from Port0's Packet Engine block in TXC.
txc_dmc_p0_pkt_size_err_addr	I	44	TXC->TDMC	Error Address for packets which exceeded required size
txc_dmc_p0_pkt_size_err	I	1	TXC->TDMC	Pulse indicating size error.
txc_dmc_p1_dma_pkt_size_err	I	16	TXC->TDMC	One- Hot Encoded signal indicating the DMA number for which the error is being reported. This is reported from Port0's Packet Engine block in TXC.
txc_dmc_p1_pkt_size_err_addr	I	44	TXC->TDMC	Error Address for packets which exceeded required size.
txc_dmc_p1_pkt_size_err	I	1	TXC->TDMC	Pulse indicating size error.

6.22.2.3 Transmit DMA-Meta Interface Signals

TABLE 6-20 TDMC- Meta Interface Write Request Interface Signals

Signal Name	I/O	Size	From/To	Description
Write REQUEST Transaction Type and Transaction Control				
tdmc_arb0_req_cmd	O	8	TDMC->Meta	Request Command
tdmc_arb0_req_address	O	64	TDMC->Meta	Memory Address: *64-bit address, 32-bit address.*
tdmc_arb0_req_length	O	14	TDMC->Meta	Data Length (bytes)
tdmc_arb0_req_dma_num	O	5	TDMC->Meta	Channel Number
tdmc_arb0_req_func_num	O	2	TDMC->Meta	Function Number

TABLE 6-20 TDMC- Meta Interface Write Request Interface Signals *(Continued)*

Signal Name	I/O	Size	From/To	Description
tdmc_arb0_req	O	1	TDMC->Meta	REQUEST Queue Request
arb0_tdmc_req_accept	I	1	Meta->TDMC	Grant REQUEST Queue request
tdmc_arb0_transfer_complete	O	1	TDMC->Meta	Transfer complete.
arb0_tdmc_data_req	I	1	Meta->TDMC	Meta Request for memory line transfer.
tdmc_arb0_data	O	128	TDMC->Meta	Packet Data.
tdmc_arb0_req_byteenable	O	16	TDMC->Meta	Always driven as 0xffff
tdmc_arb0_data_valid	O	1	TDMC->Meta	TDMC Acknowledges memory-line transfer.

TABLE 6-21 Write Acknowledge Interface Signals

Signal Name	I/O	Size	From/To	Description
Write REQUEST Transaction Type and Transaction Control				
meta_dmc_ack_ready	I	1	Meta->TDMC	Validate ACK Queue request
meta_dmc_ack_client	I	1	Meta->TDMC	Signal Indicating transaction is for TDMC
meta_dmc_ack_complete	I	1	Meta->TDMC	Completion of acknowledgement
meta_dmc_ack_dma_num	I	5	Meta->TDMC	acknowledgement dma_num
meta_dmc_ack_cmd	I	8	Meta->TDMC	acknowledgement command
meta_dmc_ack_cmd_status	I	4	Meta->TDMC	acknowledgement status
dmc_meta_ack_accept	O	1	TDMC->Meta	Grant ACK Queue Request

TABLE 6-22 Read Request Interface Signals

Signal Name	I/O	Size	From/To	Description
tdmc_arb1_req_cmd	O	8	TDMC->Meta	Commands: *Memory Read.*
tdmc_arb1_req_address	O	64	TDMC->Meta	Memory Address: *64-bit address, 32-bit address.*
tdmc_arb1_req_length	O	14	TDMC->Meta	Data Length (bytes)
tdmc_arb1_req_dma_num	O	5	TDMC->Meta	Channel Number

TABLE 6-22 Read Request Interface Signals *(Continued)*

Signal Name	I/O	Size	From/To	Description
tdmc_arb1_req_func_num	O	2	TDMC->Meta	Function Number
tdmc_arb1_req	O	1	TDMC->Meta	REQUEST Queue Request
arb1_tdmc_req_accept	I	1	Meta->TDMC	Grant REQUEST Queue Request

TABLE 6-23 Read Response Interface Signals

Signal Name	I/O	Size	From/To	Description
meta_dmc_resp_cmd	I	8	Meta->TDMC	Command Requests:
meta_dmc_resp_address	I	64	Meta->TDMC	Memory Address: *64-bit address, 32-bit address.*
meta_dmc_resp_length	I	14	Meta->TDMC	Data Length (bytes)
meta_dmc_resp_cmd_statu	I	4	Meta->TDMC	Command Status
meta_dmc_resp_dma_num	I	5	Meta->TDMC	Channel Number
meta_dmc_resp_client	I	1	Meta->TDMC	Signal Indicating transaction is for TDMC
meta_dmc_resp_ready	I	1	Meta->TDMC	Validate RESPONSE Queue Request
dmc_meta_resp_accept	O	1	TDMC->Meta	Grant RESPONSE Queue Request.
meta_dmc_resp_complete	I	1	Meta->TDMC	Fragment complete.
meta_dmc_transfer_complete	I	1	Meta->TDMC	Transfer Complete.
meta_dmc_data	I	128	Meta->TDMC	Packet Data.
meta_dmc_resp_byteenable	I	16	Meta->TDMC	First/Last Byte Enable
meta_dmc_data_status	I	4	Meta->TDMC	Packet Transfer Status:
meta_dmc_data_valid	I	1	Meta->TDMC	Meta Acknowledges Burst Transfer

6.22.2.4 Transmit DMA Interface and PIO Interface

TABLE 6-24 TDMC - PIO Interface

Signal Name	I/O	Size	From/To	Description
tdmc_pio_ack	O	1	TDMC->PIO	PIO Acknowledge.
tdmc_pio_rdata	O	64	TDMC->PIO	PIO Read Data.
tdmc_pio_err	O	1	TDMC->PIO	PIO Error
tdmc_pio_intr	O	64	TDMC->PIO	Interrupt signals from TDMC. Bits 31:0 indicate Interrupts for LDF0 and bits 63:32 indicate interrupts for LDF1.
pio_clients_addr;	I	20	PIO->TDMC	PIO Address.
pio_clients_wdata;	I	64	PIO->TDMC	PIO Write Data.
pio_clients_rd	I	1	PIO->TDMC	PIO Read/Write
pio_tdmc_sel	I	1	PIO->TDMC	PIO Client Select

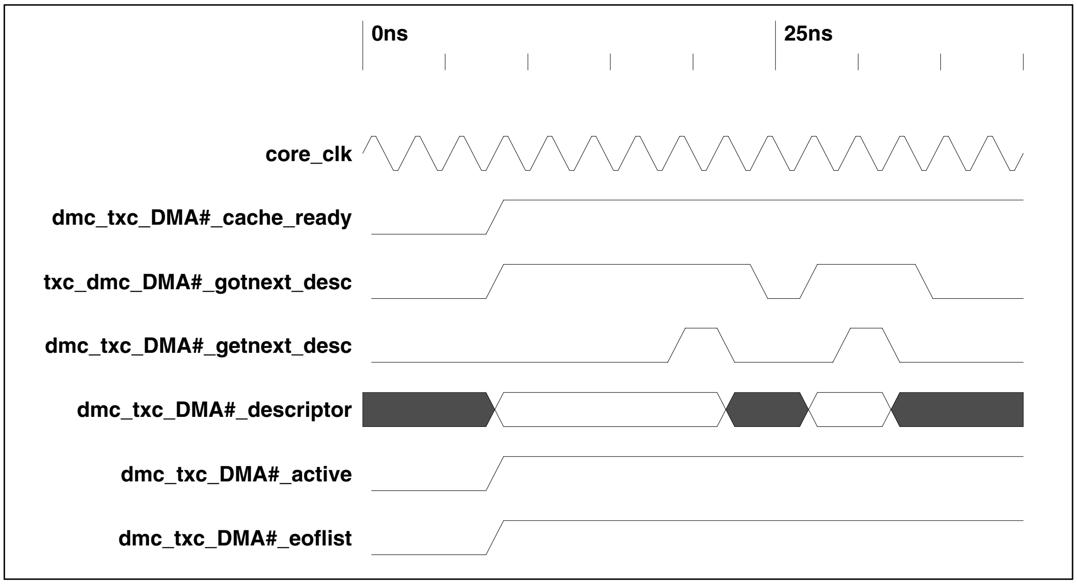
6.22.3 TDMC Interface Timing Diagrams

6.22.3.1 TXC-TDMC Interface Timing Diagrams

- Conditions: Initial Kick Received, DMA restarted after a software reset

Under the described condition, dma_active and cache_read signals are asserted for TXC. Along with these signals TDMC also asserts gotnext_desc signal which indicates to TXC to pop descriptors from TDMC. For every getnext_desc signal descriptors are popped for further processing.

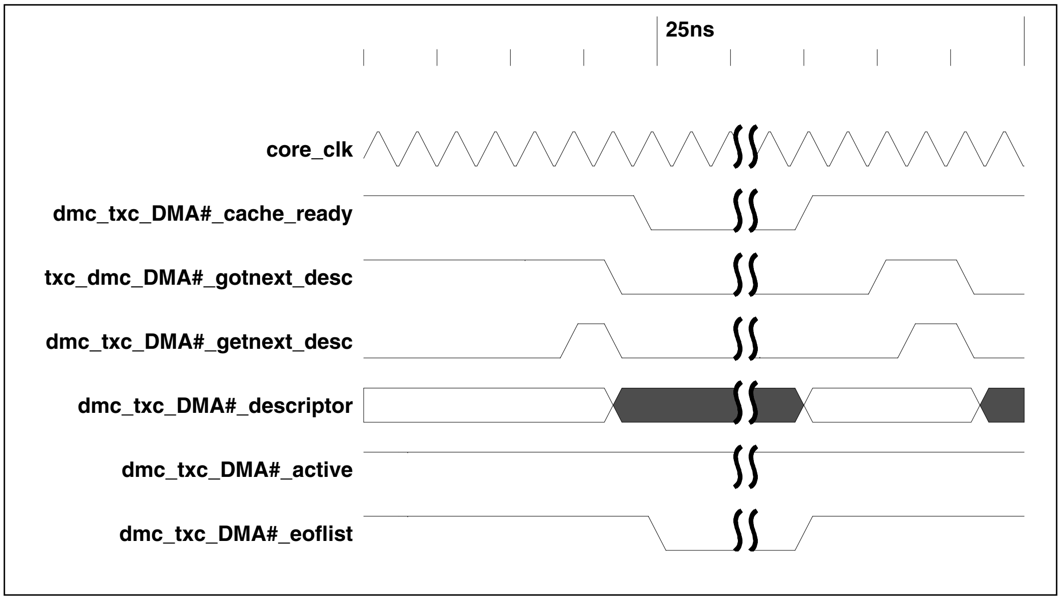
FIGURE 6-45 TDMC-TXC Interface Timing Diagram



- Conditions: Appending descriptors to an active DMA when the descriptor ring is empty.

Under this condition, dma_active remains asserted. Whenever the emptiness of the cache reaches a certain threshold as described in detail in Section read requests are sent to the host memory. Once the response is received and re-ordered if need be, cache is populated again and cache_ready signal is asserted for TXC for further processing of the data.

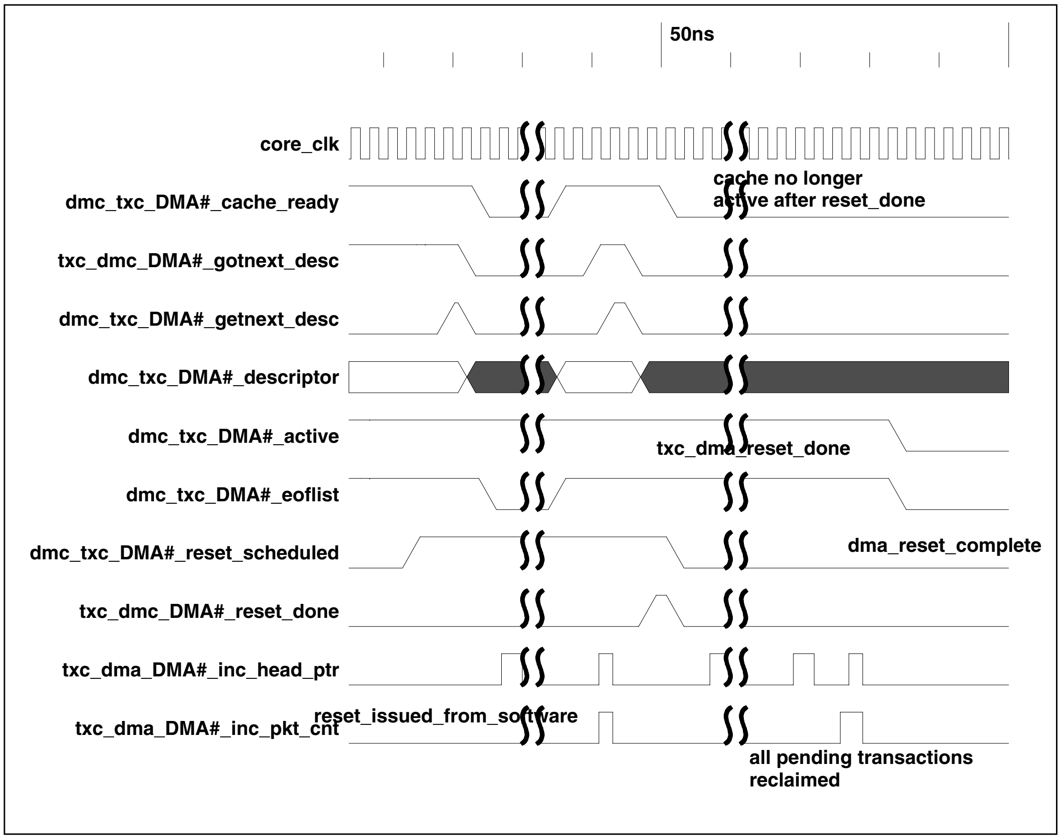
FIGURE 6-46 TDMC-TXC Interface Timing Diagram



- Conditions: Software reset issued to an active DMA.

A timing diagram of the reset process is shown in FIGURE 6-47.

FIGURE 6-47 TDMC-TXC Interface Timing Diagram



6.22.3.2 TDMC-Meta Interface Timing Diagrams

Please refer to Meta Interface Overview.

6.22.3.3 TDMC-PIO Interface Timing Diagrams

Please refer to PIO Interface.

6.22.4 Functional Block Diagram

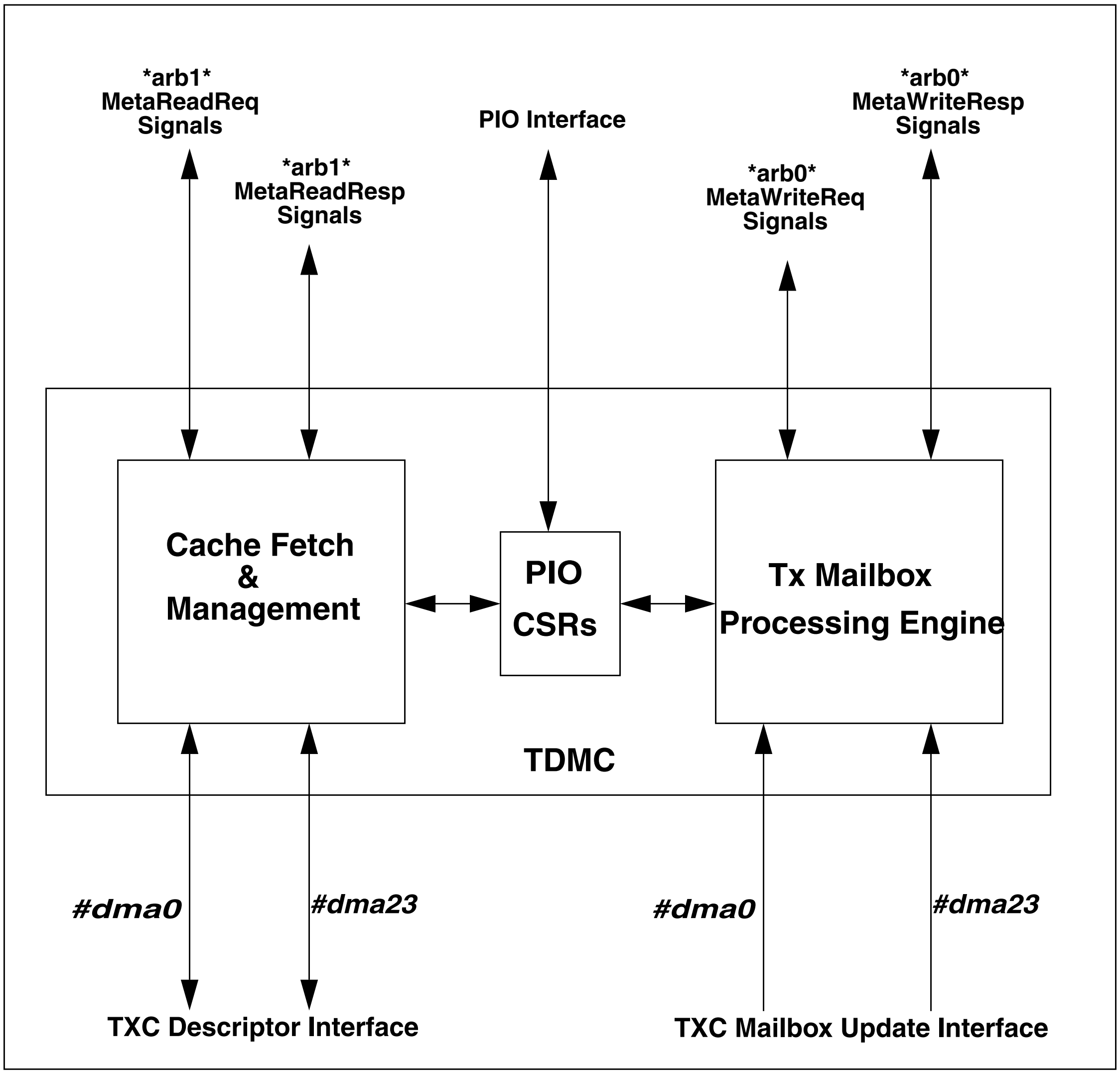
Functionally there are two main engines within the TDMC, namely

- Cache Fetch and Management Engine.
- Mailbox Processing Engine.

Both of these block are controlled by CSRs programmed by Software. A Detailed description of each of these subblocks is given in subsequent sections.

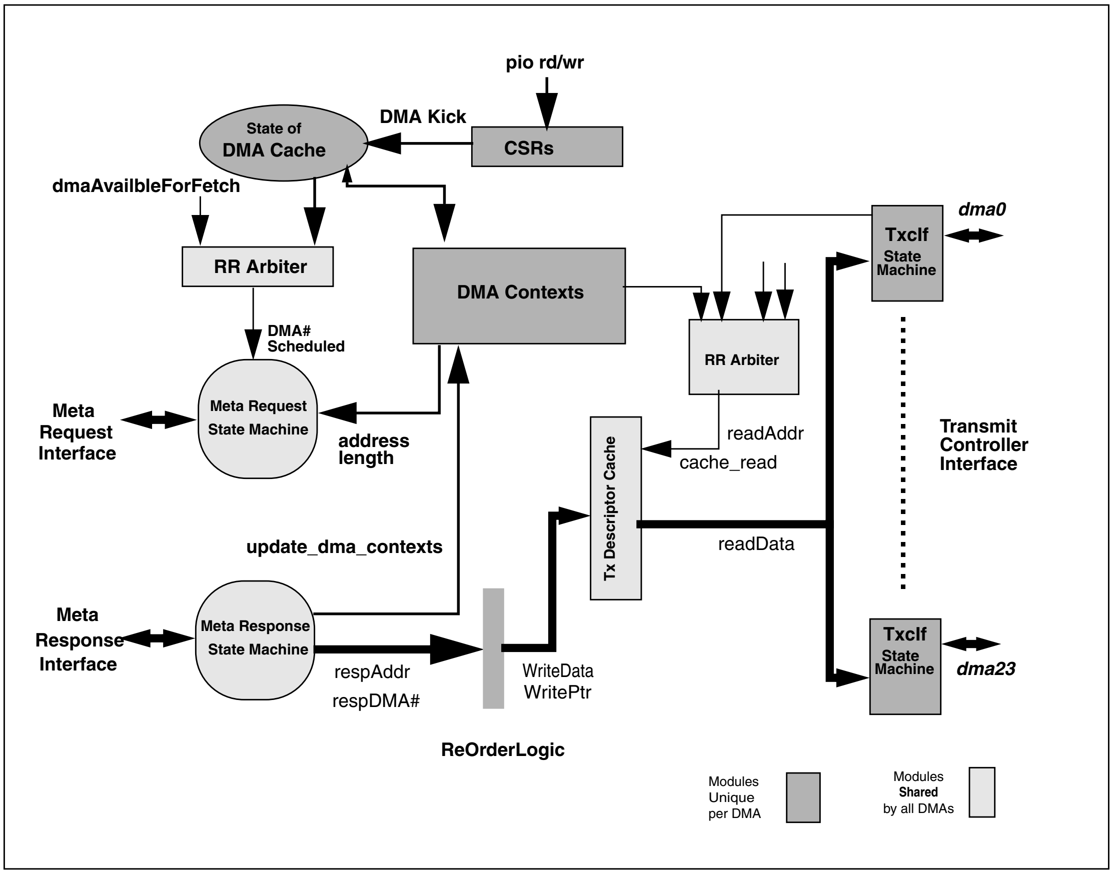
A block diagram of how external interfaces interact with each of the main engines within TDMC is shown in FIGURE 6-48.

FIGURE 6-48 TDMC Block Diagram


arb1 MetaReadReq Signals
arb1 MetaReadResp Signals
PIO Interface
arb0 MetaWriteReq Signals
arb0 MetaWriteResp Signals
Cache Fetch & Management
PIO CSRs
Tx Mailbox Processing Engine
TDMC
#dma0
#dma23
#dma0
#dma23
TXC Descriptor Interface
TXC Mailbox Update Interface

The data flow within the Cache Fetch and Management Engine is shown in FIGURE 6-49. Detail functionality of each of these subblocks is given in subsequent sections.

FIGURE 6-49 TDMC Descriptor Cache Management Data Flow



6.23 TXC Microarchitecture Specification

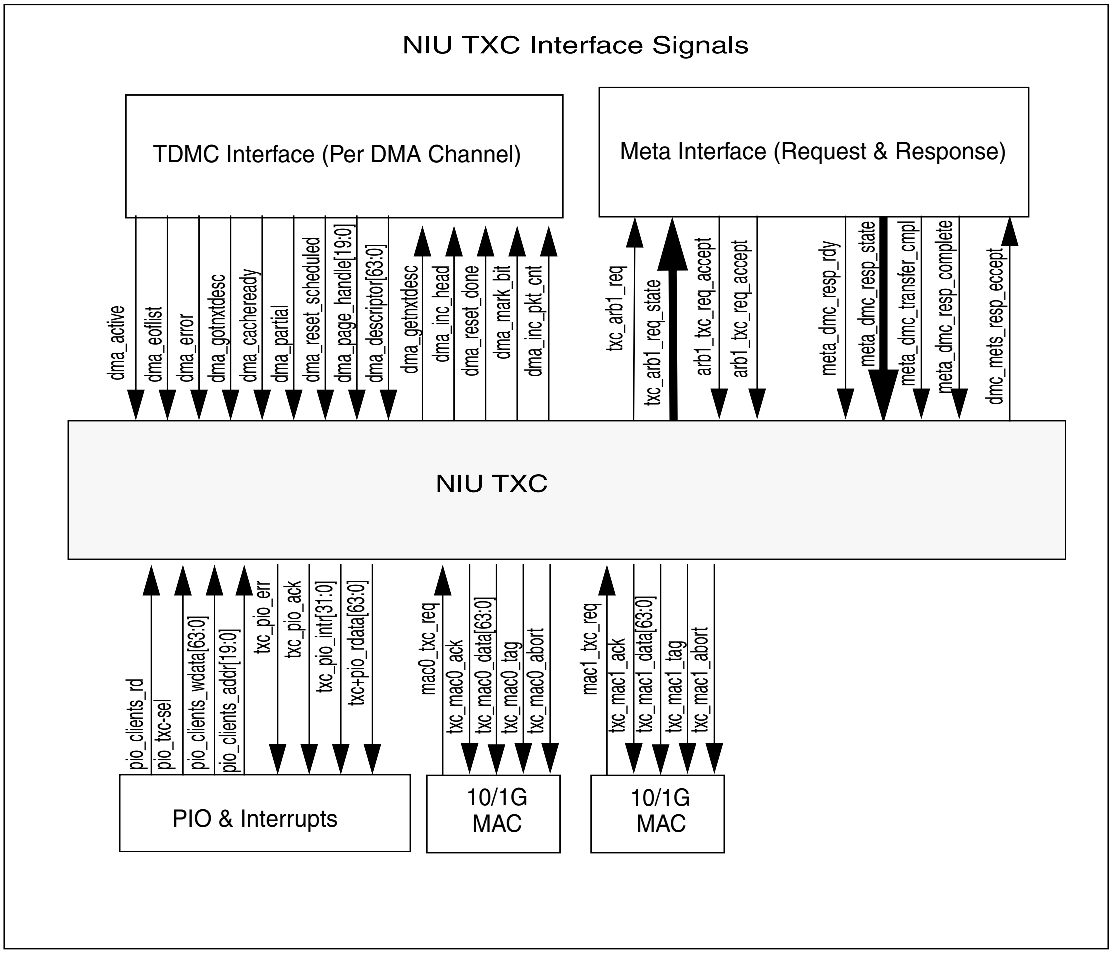
6.23.1 TXC Overview

The Transmit Controller (TXC) consists of the following functional blocks: the DRR Scheduler, Data Fetch State Machine and per port re-order/realigning buffer. The engine is designed as true store and forward, such that latencies on the host bus are abstracted from any temporal issues during transmission of packet data. The Scheduler and Data State machines are shared by all the ports in the Tx path.

Software posts transmit packets into a transmit DMA channel where each packet may be made up of a gather list. The DMA controller fetches the descriptor and informs the DRR scheduler of active DMA channels and their corresponding descriptor information. The Scheduler, coupled with the DRR information and re-order state, dispatches data request to the Data fetch state machine. This engine is now responsible for fetch and re-ordering data from the host system and storing the response data with the appropriate alignment information in the re-order buffer. Upon completion of the request, the state machine causes the re-aligner to process the stored data in the buffer. The Realigner reads data from the buffer, aligns the data into16 byte-aligned chunks and pushes the data into the Store and Forward FIFO, while at the same time primes the checksum engine, if enabled. On the completion of the packet, the re-aligner updates the checksum information and forwards the packet for transmission. The data state machine interfaces with the DMA controller for updates to the completion ring and next descriptor fetches.

The Transmit Controller (TXC) interfaces to the host system through the DMA Controller & Cache via the Meta interface. The architecture of the Meta interface is better described in the TDMC Microarchitecture specification. However, the use of the Meta interface abstracts out the physical MTU size of the host bus.

FIGURE 6-50 NIU_TXC Interface Diagram



6.23.2 Meta Interface Signals

TABLE 6-25 Meta Request Queue

Signal Name	I/O	Size	From/To	Description
dmc_txc_req_ack	I	1	DMU->TXC	Request Acknowledge from Meta
txc_dmc_transID	I	6	TXC->DMU	Transaction Identification
txc_dmc_req	O	8	TXC->DMU	Transaction Requests
txc_dmc_cmd	O	8	TXC->DMU	Command Requests: *Memory Read, Memory Read Bypass, Message Interrupt.*
txc_dmc_dma_#	O	5	TXC->DMU	DMA initiating request
txc_dmc_address	O	64	TXC->DMU	Memory Address: *64-bit address, 32-bit address.*
txc_dmc_length	O	16	TXC->DMU	Data Length (DW units)

TABLE 6-26 Meta Response Queue

Signal Name	I/O	Size	From/To	Description
dmc_txc_resp_rdy	I	1	DMU->TXC	Read Response ready
dmc_txc_port_num	I	2	DMU->TXC	Port Number
dmc_txc_transID	I	5	DMU->TXC	Transaction Identification
dmc_txc_cmd_status	I	4	DMU->TXC	Command Phase Status
dmc_txc_dma_#	I	5	DMU->TXC	DMA channel number
dmc_txc_resp_complete	I	1	DMU->TXC	Response complete
dmc_txc_trans_complete	I	1	DMU->TXC	Transaction Complete
dmc_txc_data_valid	I	1	DMU->TXC	Data valid strobe
dmc_txc_length	I	12	DMU->TXC	Current Data Length in bytes
dmc_txc_bytecount	I	12	DMU->TXC	Remaining Byte Count for Request.
dmc_txc_byteenable	I	16	DMU->TXC	Byte Enable
dmc_txc_address	I	64	DMU->TXC	Current Data Address:
dmc_txc_data	I	128	DMU->TXC	Response Data
dmc_txc_data_status	I	4	DMU->TXC	Data Phase Status
dmc_txc_resp_accept	O	1	TXC->DMU	Response accept

6.23.3 TXC to TDMC Interface

# is the DMA channel number. 0-15

TABLE 6-27 TXC to TDMC Interface

Signal Name	I/O	Size	From/To	Description
txc_dmc_DMA#_getnextdesc	O	1	TXC->DMU	Get next descriptor list
txc_dmc_DMA#_unrecov	O	1	TXC->DMU	Unrecoverable error
txc_dmc_DMA#_inc_head_ptr	O	1	TXC->DMU	Increment Head Pointer
txc_dmc_DMA#_ update_mbox	O	1	TXC->DMU	Update Mailbox
txc_dmc_DMA#_mark_bit	O	1	TXC->DMU	Mark bit seen
dmc_txc_DMA#_active	I	1	DMU->TXC	DMA # is active
dmc_txc_DMA#_eofList	I	1	DMU->TXC	DMA# end of descriptor list
dmc_txc_DMA#_error	I	1	DMU->TXC	DMA# error
dmc_txc_DMA#_gotnxtdesc	I	1	DMU->TXC	DMA# got next descriptor list
dmc_txc_DMA#_cache_ready	I	1	DMU->TXC	DMA# Cache Ready
dmc_txc_DMA#_partial	I	1	DMU->TXC	DMA# Indicating Partial gather descriptors available
dmc_txc_DMA#_page_handle	I	20	DMU->TXC	Relocation handle for packet address
dmc_txc_DMA#_descriptor	I	64	DMU->TXC	DMA# descriptor

6.23.4 TXC to 10 G MAC Interface

TABLE 6-28 XMAC Interface

Signal Name	I/O	Size	From/To	Description
mac_txc_req	I	1	MAC->TXC	Request for Data. This signal is driven by the MAC and is used to indicate to the TXC block that MAC is ready for the next burst of data transfer.
txc_mac_ack	O	1	TXC->MAC	This signal is used to indicate that the TXC block is executing a 8 byte data transfer.
txc_mac_tag	O	1	TXC->MAC	This signal is used to indicate the completion of packet transfer from the TXC to MAC. For each packet transmitted, the last data double-word and the status word will have this bit set. The remaining double-words of the packet shall have this bit cleared.
txc_mac_abort	O	1	TXC->MAC	This signal is used to indicate that the current packet being transferred over to the MAC should be aborted.
txc_mac_status	O	4	TXC->MAC	Packet Status information
txc_mac_data	O	64	TXC->MAC	This data bus is used to transfer 8 bytes of transmit packet data. The data will be presented to MAC in a `little endian' format.

6.23.5 PIO Interface

TABLE 6-29 PIO Interface

Signal Name	I/O	Size	From/To	Description
txc_pio_ack_l	O	1	TXC->PIO	PIO acknowledge.
txc_pio_rdata	O	64	TXC->PIO	PIO read data.
pio_txc_req_l	I	1	PIO->TXC	PIO request.
pio_txc_wr_l	I	1	PIO->TXC	PIO write
pio_txc_address	I	18	PIO->TXC	PIO address.
pio_txc_wdata	I	64	PIO->TXC	PIO write data.

6.23.6 NIU_TXC Block Diagrams

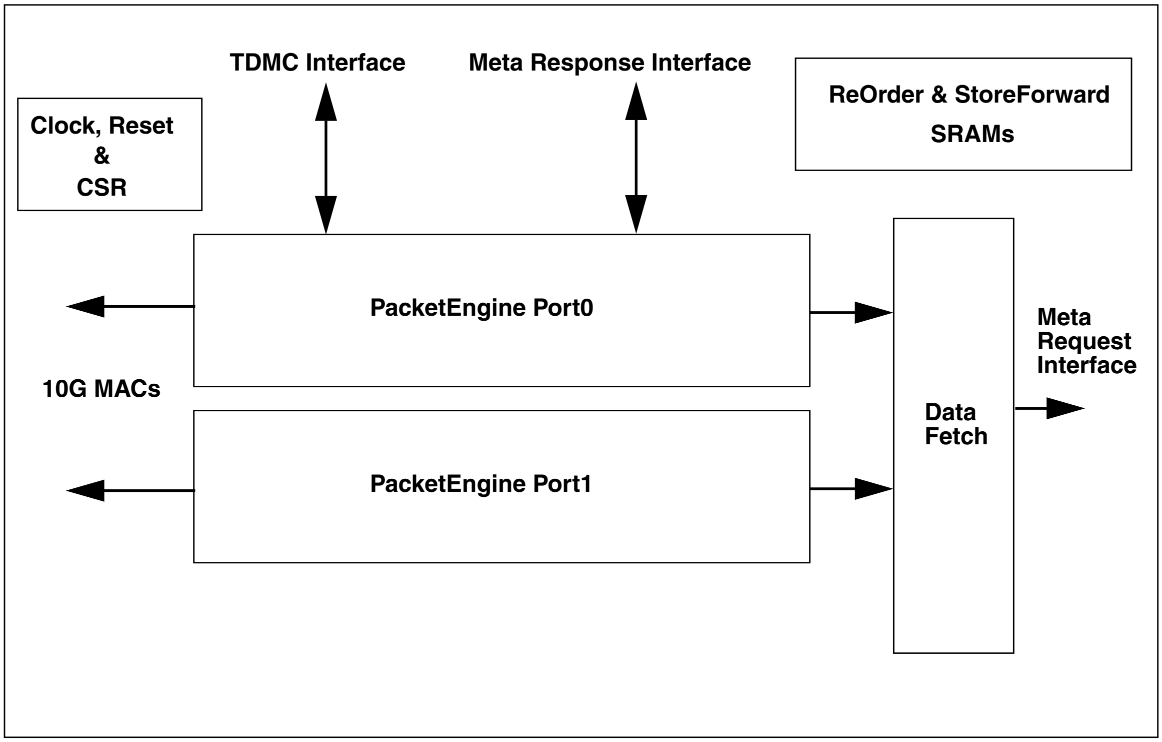


FIGURE 6-51 NIU_TXC Block Diagram

FIGURE 6-52 Packet Engine Block Diagram

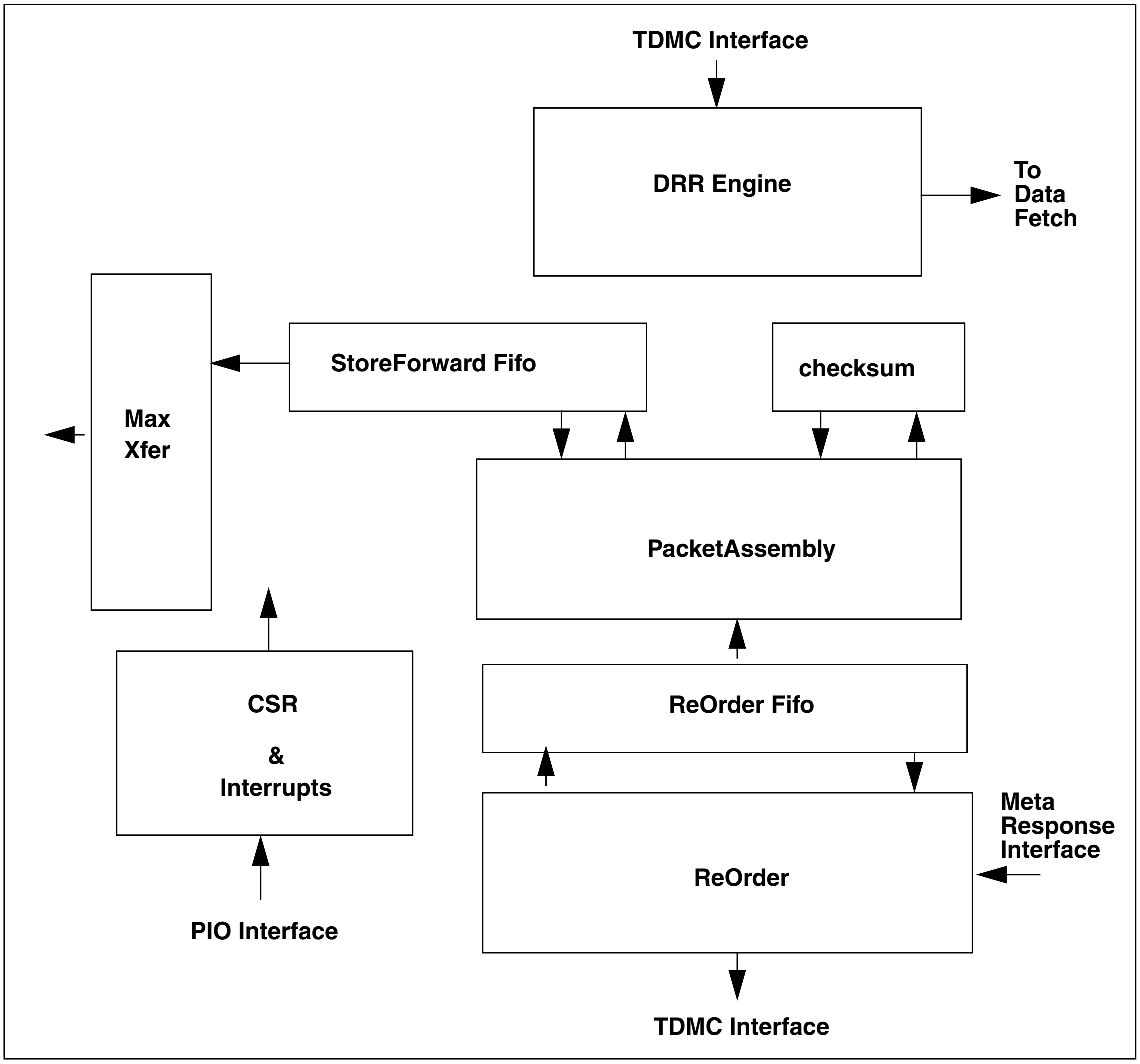


FIGURE 6-53 Deficit Round Robin Engine

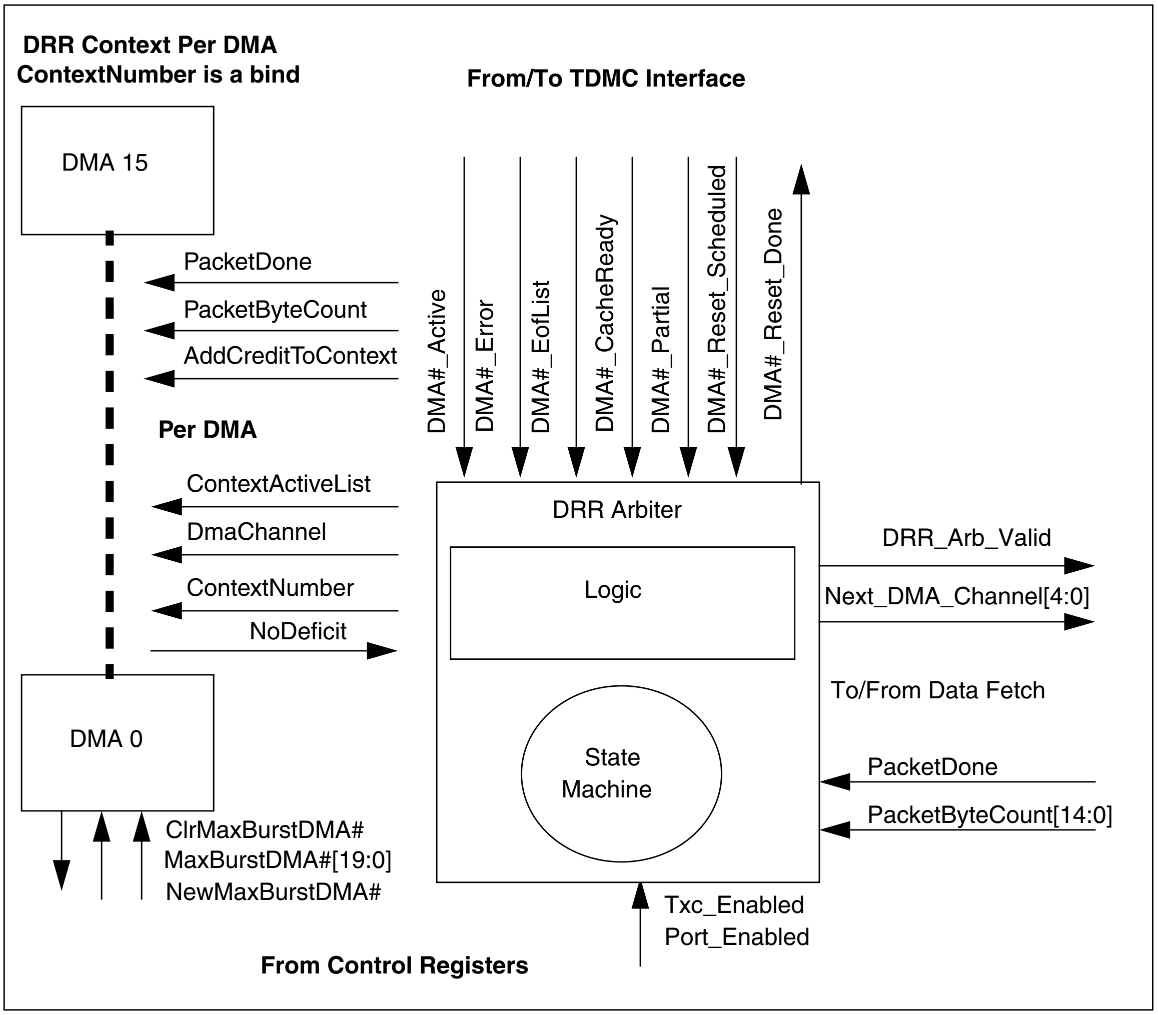


FIGURE 6-54 ReOrder Engine Block Diagram

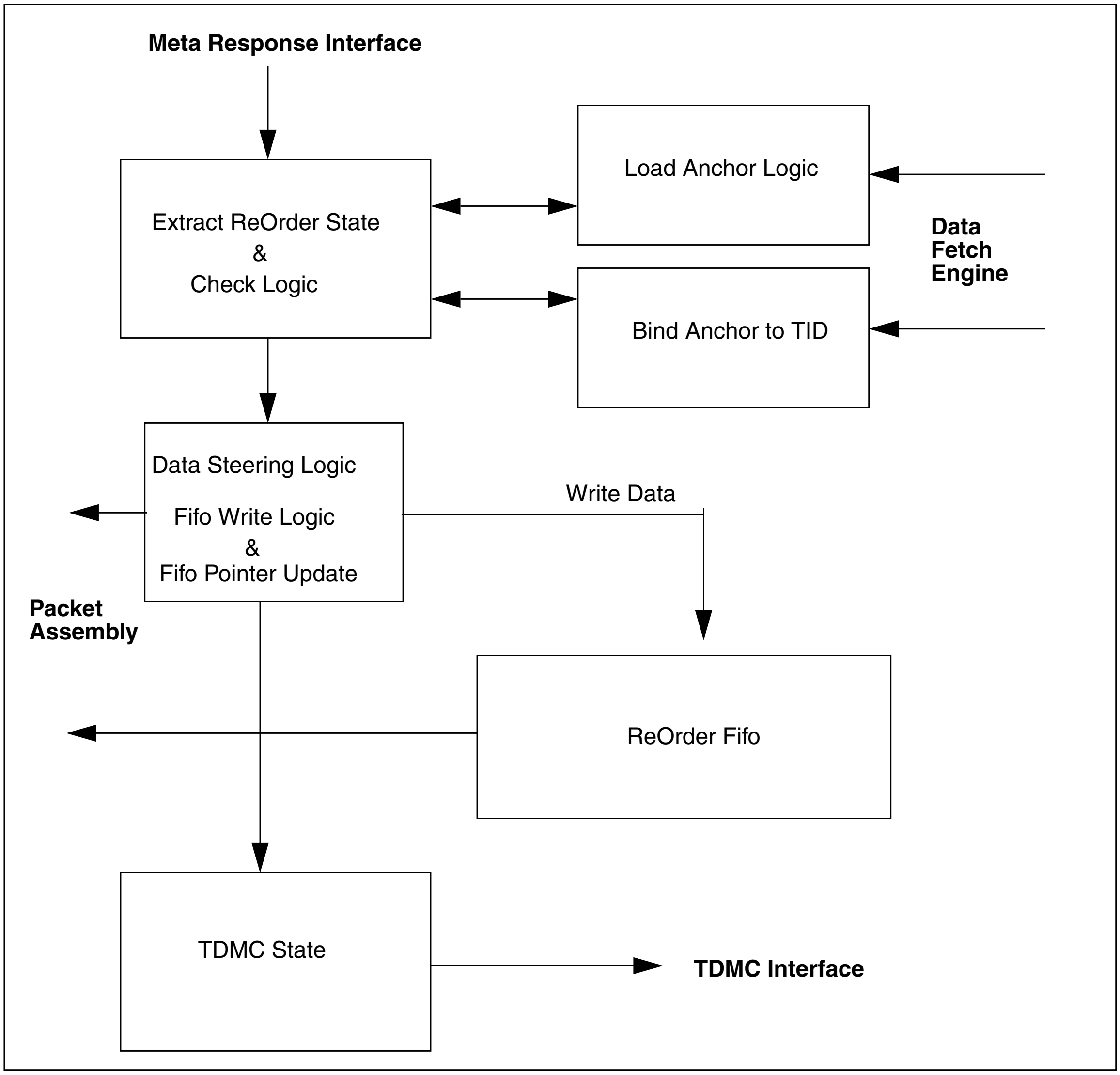


FIGURE 6-55 ReOrder State Management

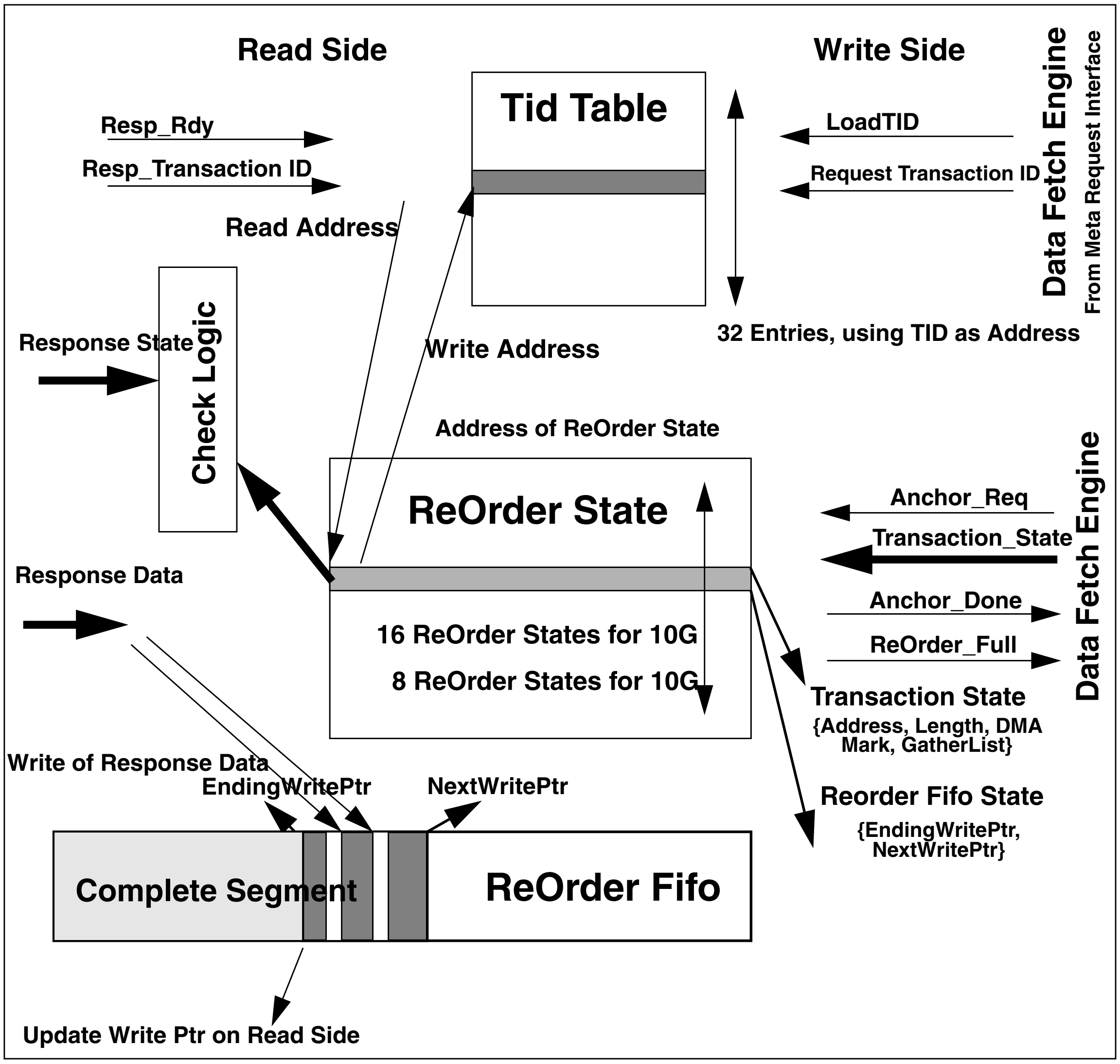
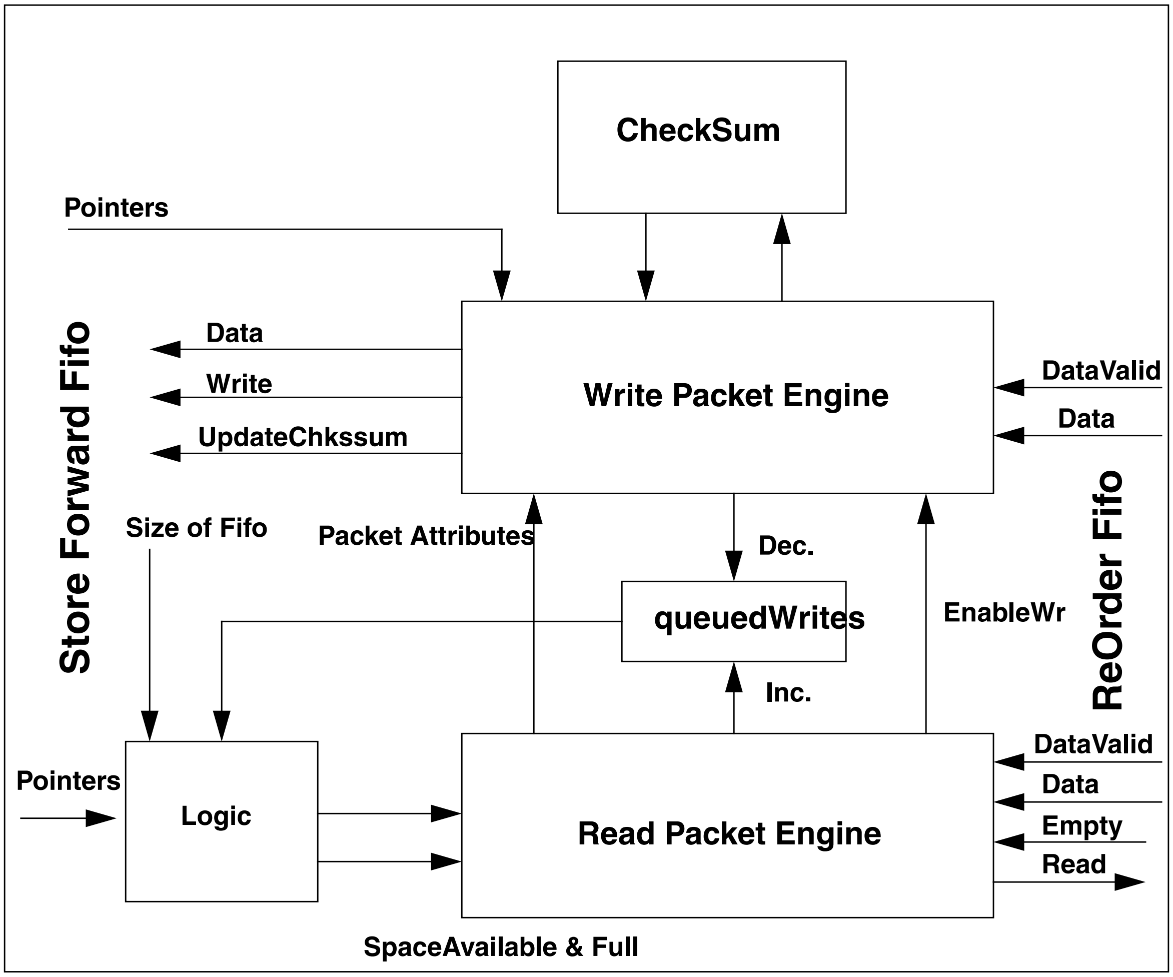


FIGURE 6-56 Packet Assembly Engine



6.24 Meta Arb Microarchitecture Specification

6.24.1 Meta Arb Overview

- Sits between NIU clients and Host Interface Module
- Arbitrates Read and Write Requests from NIU clients
  - Selected client is then served by Host Interface Module
- Two independent arbiters
  - Read arbiter; Simple Round Robin Arbiter arbitrating between the read
  - Requests to the Meta Interface issued by multiple clients (txc, tdmc, rdmc).
  - Write arbiter; Simple Round Robin Arbiter arbitrating between the write requests to the Meta Interface issued by multiple clients (tdmc and rdmc). txc does not issue writes.
- Read and Write Tag Manager
  - Snoops META response bus
  - Reclaim META tag when transaction is completed
  - Handles META tag if transaction is unsuccessful

FIGURE 6-57 Top Level View of META_ARB and Neighbor Blocks

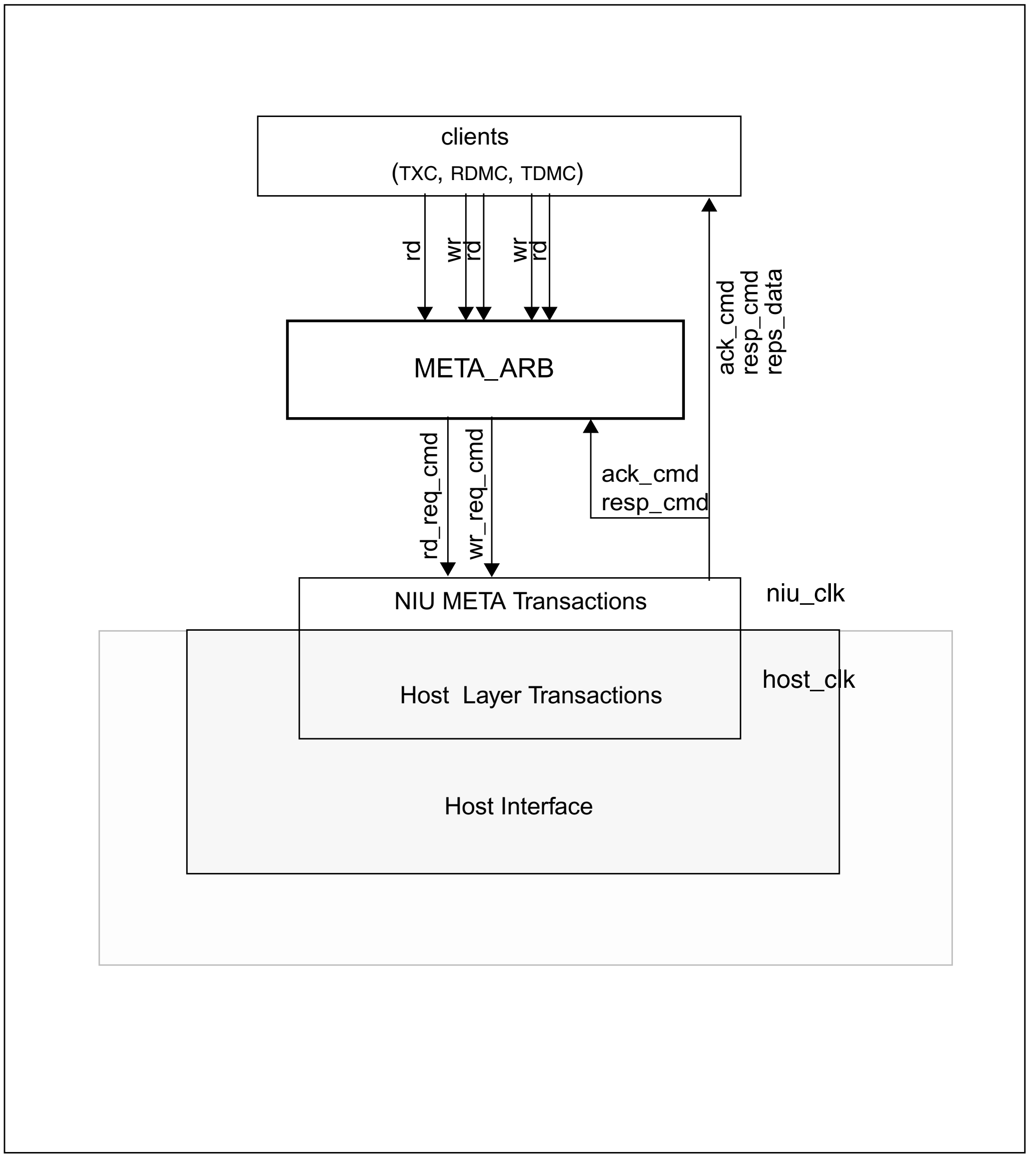
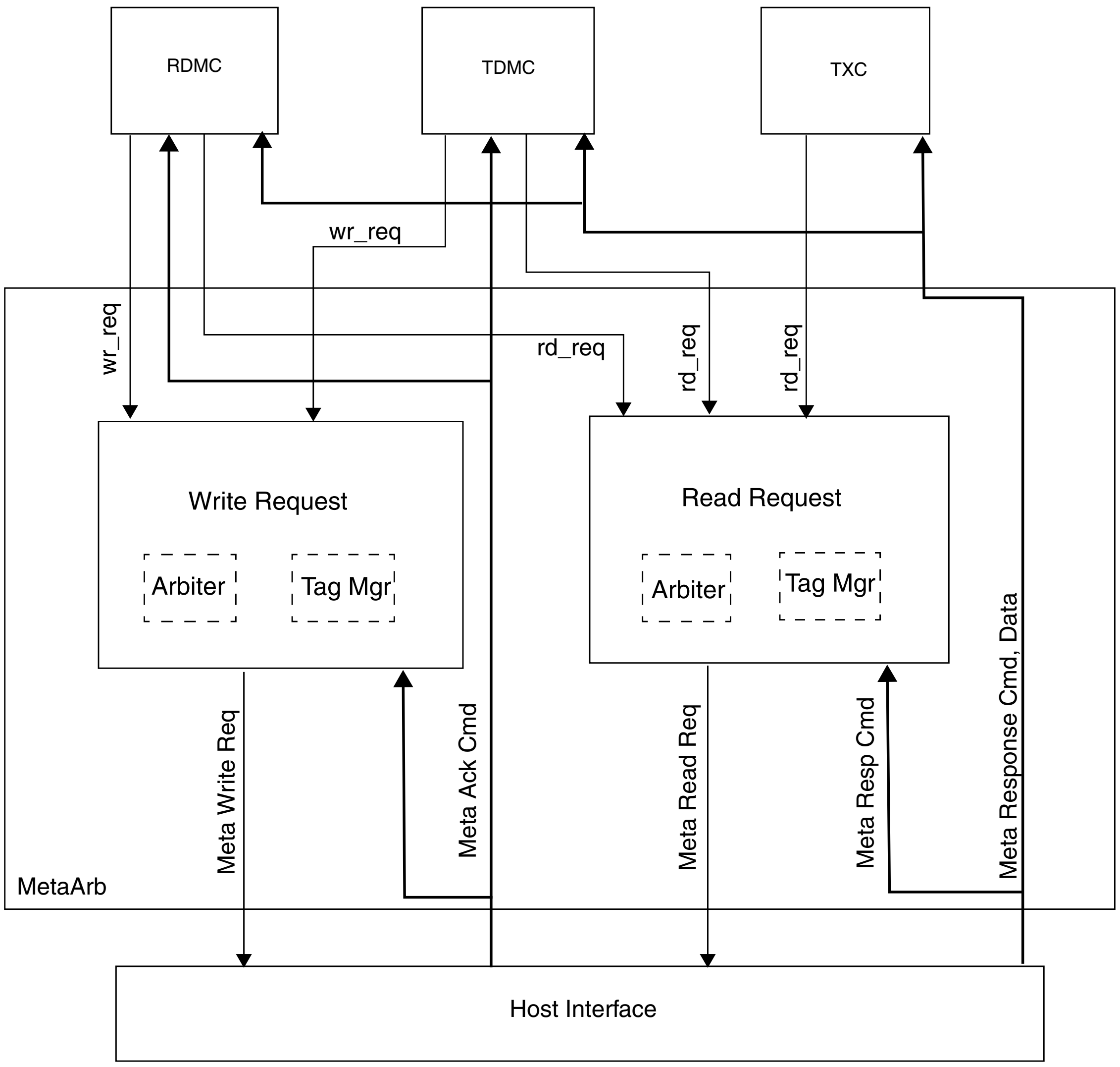


FIGURE 6-58 META ARB Top Level Flow



6.24.2 Meta Arb Interface Signals

The Meta Arbiter Interface uses the Meta protocol and is specified in the Meta Interface Specification document, please refer to Meta Interface Microarchitecture Specification for signal and timing diagram details.

TABLE 6-30 Write Request Signals

Signal Name	I/O	Size	From/To	Description
Request Control Path				
dmc_meta0_req	O	1	ARB->HOST	Send Queue Request
dmc_meta0_req_cmd	O	8	ARB->HOST	Command Requests: *Memory Read, Memory Write, Completion.*
dmc_meta0_req_address	O	64	ARB->HOST	64 bit Memory Address
dmc_meta0_req_transID	O	6	ARB->HOST	Transaction Identification
dmc_meta0_req_length	O	14	ARB->HOST	Data Length (byte units)
dmc_meta0_req_port_num	O	2	ARB->HOST	Port number corresponding to the request (to be returned with read response)
dmc_meta0_req_dma_num	O	5	ARB->HOST	Dma number corresponding to the request (to be returned with read response)
dmc_meta0_req_client	O	8	ARB->HOST	Requesting Client (vector, one-hot encoded)
meta_dmc0_req_accept	I	1	HOST->ARB	Grant Send Queue Request
Request Data Path				
meta_dmc0_data_req	I	1	HOST->ARB	Host Request for Burst Transfer.
dmc_meta0_data_valid	O	1	ARB->HOST	DMU sends data Ack with every cycle of valid data.
dmc_meta0_status	O	4	ARB->HOST	Packet Transfer Status: Complete, Abort.
dmc_meta0_data	O	128	ARB->HOST	Data.
dmc_meta0_req_byteenable	O	16	ARB->HOST	Contains the byteenables for each byte of data in the 16 byte data transfer. byteenable[N]==1 implies write data[8N + 7: 8N] is enabled (valid).
dmc_meta0_transfer_complete	O	1	ARB->HOST	Transfer complete. No additional data for this transaction. Asserted coincidental with last data.

FIGURE 6-59 Write Request - Command, Data Phase Illustration

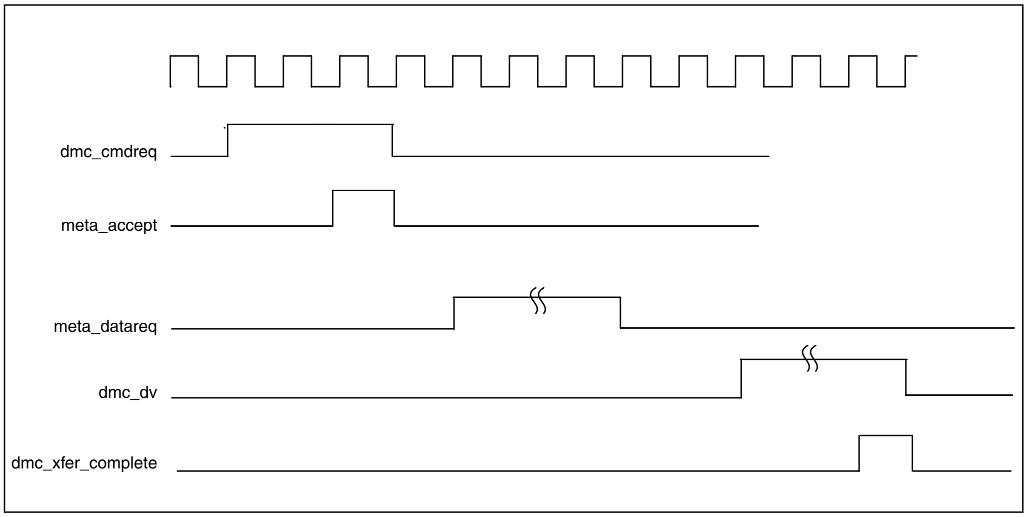


FIGURE 6-60 Write Request from the Same Client

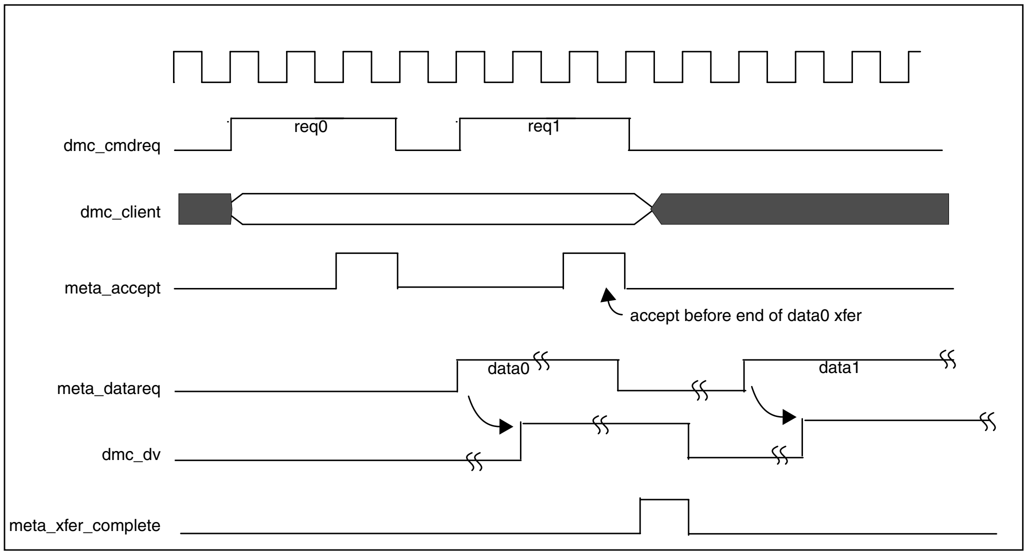


FIGURE 6-61 Write Request from Different Client

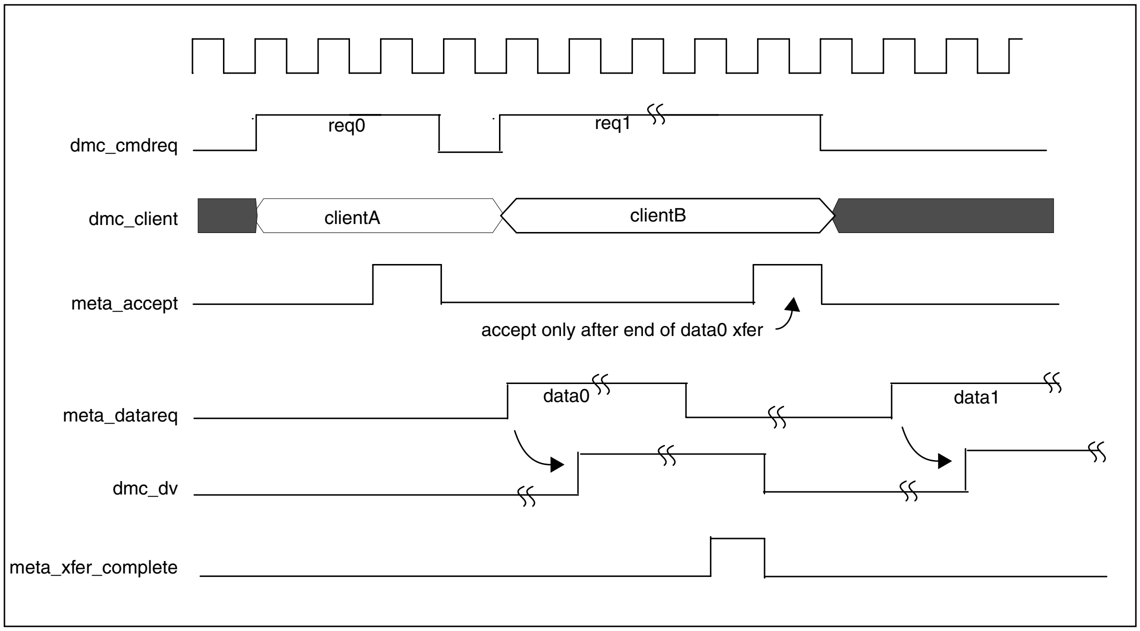


TABLE 6-31 Non-Posted Write Response Phase (Acknowledgement)

Signal Name	I/O	Size	From/To	Description
Non-Posted Write Acknowledgement Control Path				
meta_dmc_ack_ready	I	1	HOST->CLIENT	acknowledge ready
meta_dmc_ack_cmd_status	I	4	HOST->CLIENT	encoded command
meta_dmc_ack_dma_num	I	5	HOST->CLIENT	corresponding dma number
meta_dmc_ack_transID	I	6	HOST->CLIENT	Transaction Identification. (Meta Arbiter snoops this signal)
meta_dmc_ack_client	I	8	HOST->CLIENT	targeted client to receive this acknowledgement

TABLE 6-31 Non-Posted Write Response Phase (Acknowledgement) *(Continued)*

Signal Name	I/O	Size	From/To	Description
dmc_meta_ack_accept	O	8	CLIENT->ARB	Grant Receive Queue Request. (Meta Arbiter snoops this signal)
meta_dmc_ack_complete	I	8	HOST->CLIENT	single pulse to indicate acknowledgement to corresponding transaction
meta_dmc_ack_transfer_complete	I	8	HOST->CLIENT	Single pulse to indicate acknowledgement to corresponding completion of transaction (Meta Arbiter snoops this signal)

FIGURE 6-62 Non-Posted Write Acknowledgement

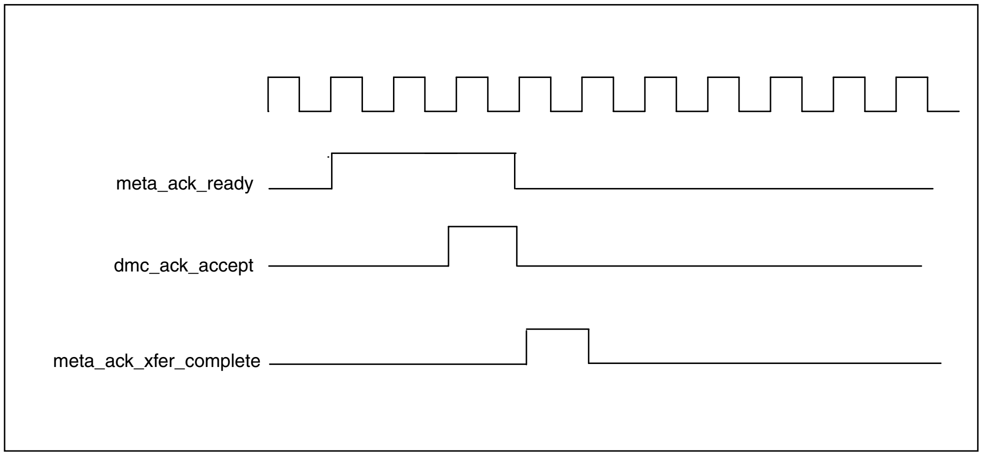


TABLE 6-32 Read Request Command Phase

Signal Name	I/O	Size	From/To	Description
Read Request				
Request Control Path				
dmc_meta1_req_cmd	O	8	ARB->HOST	Command Requests: *Memory Read*
dmc_meta1_req_address	O	64	ARB->HOST	64 bit Memory Address
dmc_meta1_req_transID	O	6	ARB->HOST	Transaction Identification
dmc_meta1_req_length	O	14	ARB->HOST	Data Length (byte units)

TABLE 6-32 Read Request Command Phase *(Continued)*

Signal Name	I/O	Size	From/To	Description
dmc_meta1_req_port_num	O	2	ARB->HOST	Port number corresponding to the request (to be returned with read response)
dmc_meta1_req_dma_num	O	5	ARB->HOST	Dma number corresponding to the request (to be returned with read response)
dmc_meta1_req_client	O	8	ARB->HOST	Requesting Client (vector, one-hot encoded)
dmc_meta1_req	O	1	ARB->HOST	Send Queue Request
meta_dmc1_req_accept	I	1	HOST->ARB	Grant Send Queue Request

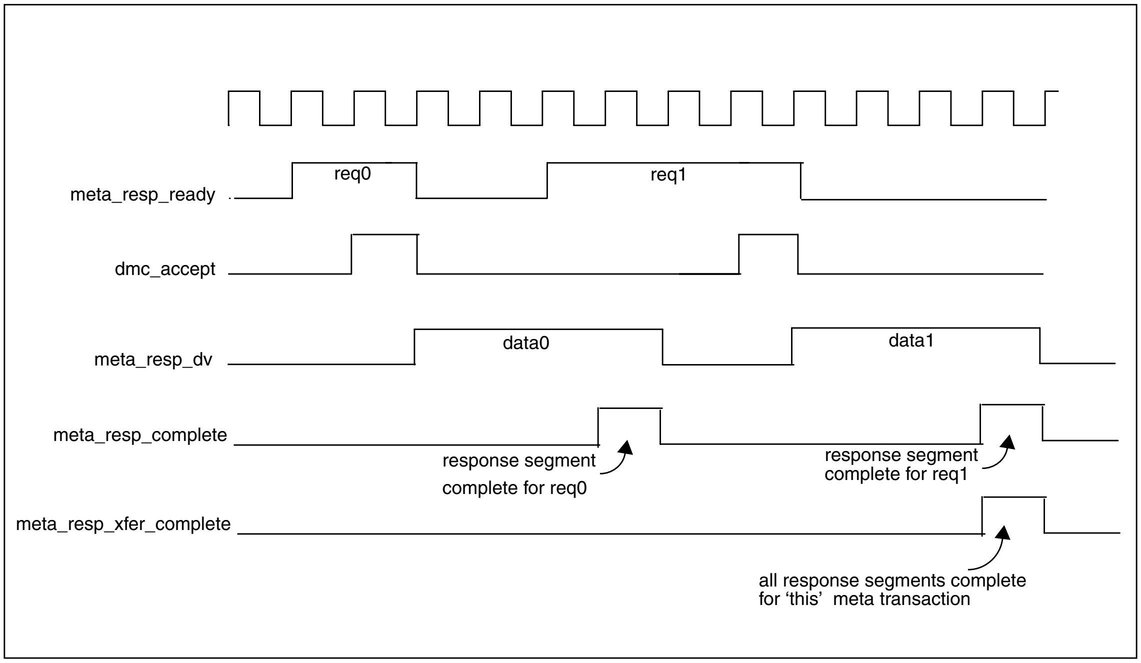
TABLE 6-33 Read Response Command and Data Phase

Signal Name	I/O	Size	From/To	Description
Resp Data Control Path				
meta_dmc_resp_ready	I	1	HOST->CLIENT	response ready
meta_dmc_resp_cmd	I	8	HOST->CLIENT	encoded respond command
meta_dmc_resp_cmd_status	I	4	HOST->CLIENT	0xf - transaction error
meta_dmc_resp_address	I	64	HOST->CLIENT	response address
meta_dmc_resp_length	I	14	HOST->CLIENT	length in bytes for this segment of response transfer
meta_dmc_resp_transID	I	6	HOST->CLIENT	Transaction Identification (Meta Arbiter snoops this signal)
meta_dmc_resp_port_num	I	2	HOST->CLIENT	port number corresponding to this response segment
meta_dmc_resp_dma_num	I	5	HOST->CLIENT	dma number corresponding to this response segment
meta_dmc_resp_client	I	8	HOST->CLIENT	targeted client to receive this response segment
dmc_meta_resp_accept	I	8	CLIENT->HOST	Grant Receive Queue Request (Meta Arbiter snoops this signal)
Response Data Path				
meta_dmc_data_valid	I	8	HOST->CLIENT	response data is valid
meta_dmc_data	I	128	HOST->CLIENT	response data
meta_dmc_resp_byteenable	I	16	HOST->CLIENT	byte enable per data byte

TABLE 6-33 Read Response Command and Data Phase *(Continued)*

Signal Name	I/O	Size	From/To	Description
meta_dmc_data_status	I	4	HOST->CLIENT	0x0 - good data 0xf - data error
meta_dmc_resp_complete	I	8	HOST->CLIENT	single pulse on the last data cycle to indicate that this segment of response transfer is complete
meta_dmc_transfer_complete	I	8	HOST->ARB	Single pulse on the last data cycle to indicate that all the read responses are returned from the host for the complete request i.e. the transaction is complete. (Meta Arbiter snoops this signal)

FIGURE 6-63 Read Response - Command, Data Phase


meta_resp_ready
req0
req1
dmc_accept
meta_resp_dv
data0
data1
meta_resp_complete
response segment complete for req0
response segment complete for req1
meta_resp_xfer_complete
all response segments complete for 'this' meta transaction

6.25 Meta Interface Microarchitecture Specification

6.25.1 Meta Interface Overview

The host Unit and the DMA Controllers communicate point-to point via the Meta Interface. Another unit, the host, also share the same point-to-point Meta Interface to the DMA Controllers. Note that only one point-to-point connection can be maintained given a particular mode of operation. This section describes the signal and timing diagrams of the Meta Interface. The Meta Interface is composed of three functional groups:

- REQUEST (separate read, write) queues,
- RESPONSE queue,
- and ACKNOWLEDGMENT queue.

The REQUEST, RESPONSE, and ACKNOWLEDGEMENT Meta interfaces can operate up to 300MHz in a 0.13um ASIC process.

The REQUEST Queue is responsible for issuing COMMAND request and transmitting payload to the Host module. The Host module translates the REQUEST Queue commands request into the appropriate host Transaction Layer Protocol (TLP) format, type, length, and transaction descriptor; and, depending on the type of command issued, pulls data in 16 byte chunks from the REQUEST Queue as needed for transmission.

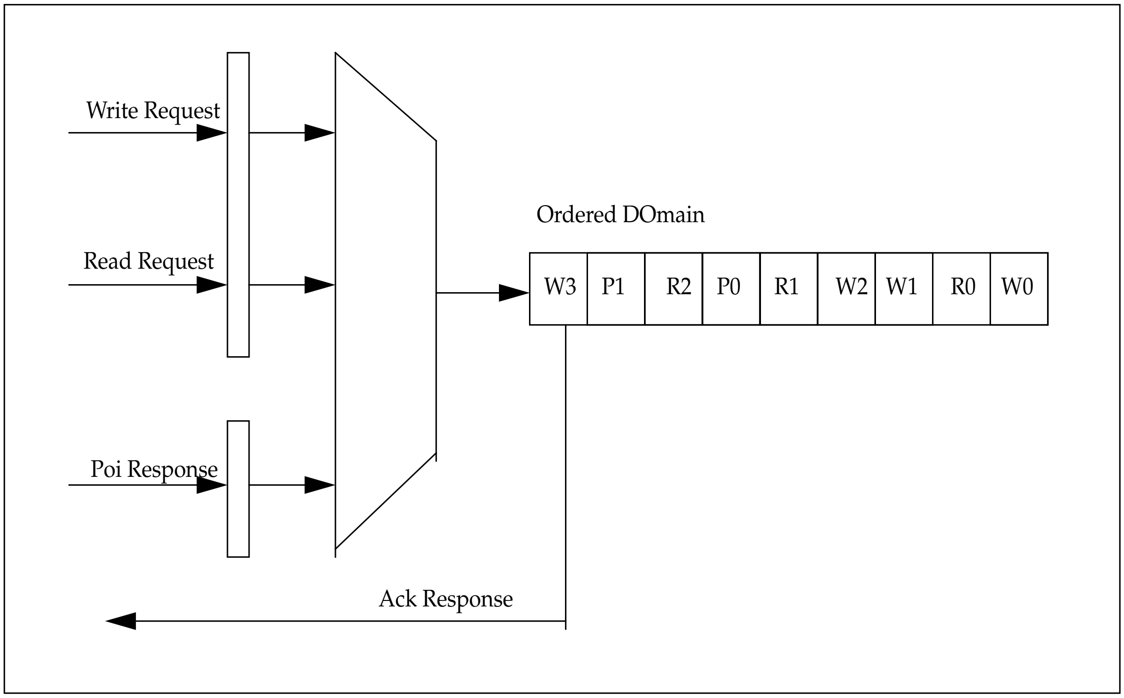
The RESPONSE Queue is responsible for receiving COMMAND request and payload from the host module. The host module translates TLP Packets into the appropriate RESPONSE Queue COMMAND, address, length, byte enables, and completion status; and, depending on the type of command issued, pulls data in 16 byte chunks from the host module as needed. The RESPONSE Credit is infinite.

The ACKNOWLEDGEMENT queue is responsible for indicating when posted write request have been completely serviced in the ordered domain. For example, a posted write of 1K bytes segment into four write request of 256 bytes will generate an ACKNOWLEDGEMENT when the last write request is scheduled to be dispatched on the wire.

6.25.2 Meta Request Interface Signals

The REQUEST Meta Interface is composed of two separate and independent buses, one for memory writes and the other for memory reads. The object of this bus design is to mitigate head-of-queue blocking of read REQUEST transactions during the service write REQUEST. The host shall support this requirement by interleaving read and write transaction layer packets (on packet boundaries) of concurrent read and write REQUESTS. PIO RESPONSE can also be interleaved.

FIGURE 6-64 Interleaved Read Write and PIO Packets



6.25.3 Write Request Interface Signals

TABLE 6-34 Write Request Meta Signals

Signal Name	Size	From/To	Description
Write REQUEST Transaction Type and Transaction Control			Nomenclature: Prefixes dmc_meta0 and meta_dmc0 signifies write interface signal.

meta_host0_req_cmd	8	Meta->host	Commands: *Memory Write, Memory Posted Write, and Flush.*
			 Command Request Encoding:
			 [7:6] Reserved;
			 [5] Posted=1, Non-Posted=0;
			 [4] Ordered=1, Non-Ordered=0;
			 [3] 64-bit addressing=1, 32-bit addressing=0;
			 [2:0] Memory Write=001; Reserved=1x1,x11,xx0.
			 Note: IO, Cfg, Msg, and Cpl type commands are not supported by the REQUEST Queue.

meta_host0_req_address	64	Meta->host	Memory Address: *64-bit address, 32-bit address.*
meta_host0_req_length	14	Meta->host	Data Length (bytes)
meta_host0_req_transID	6	Meta->host	Transaction Identification
meta_host0_req_port_num	2	Meta->host	Port Number
meta_host0_req_dma_num	5	Meta->host	Channel Number
meta_host0_req_client	8	Meta->host	Requesting Client (vector, one-hot encoded)
meta_host0_req	1	Meta->host	REQUEST Queue Request
meta_host0_transfer_complete	1	Meta->host	Transfer complete. No additional data for this transaction. Asserted coincides with last data.
host_meta0_req_accept	1	host->Meta	Grant REQUEST Queue request
Write REQUEST Data and Data Control			
host_meta0_data_req	1	host->Meta	Meta Request for memory line transfer.
meta_host0_data	128	Meta->host	Packet Data.
meta_host0_req_byteenable	16	Meta->host	First/Last Byte Enable
meta_host0_status	4	Meta->host	Packet Transfer Status: [3:0] Reserved Note: Data transfers are in memory-line units.
meta_host0_data_valid	1	Meta->host	DMU Acknowledges memory-line transfer.
Write REQUEST host Error Flag			
host_meta0_req_errors	1	host->Meta	Flag to report errors back to NIU. Flag is asynchronous with respect to write REQUEST events.

6.25.4 Write Request Interface Timing

FIGURE 6-65 Single Write Request

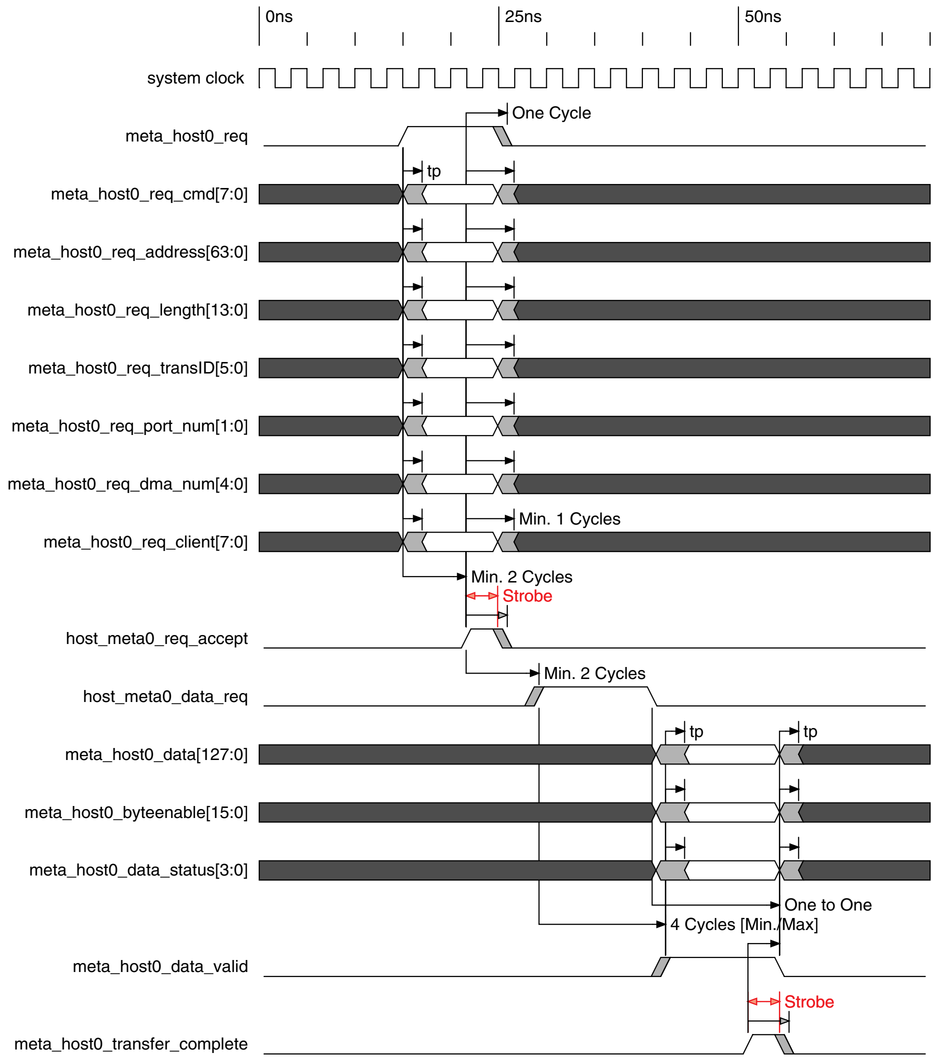

0ns
25ns
50ns
system clock
meta_host0_req
One Cycle
meta_host0_req_cmd[7:0]
tp
meta_host0_req_address[63:0]
meta_host0_req_length[13:0]
meta_host0_req_transID[5:0]
meta_host0_req_port_num[1:0]
meta_host0_req_dma_num[4:0]
Min. 1 Cycles
meta_host0_req_client[7:0]
Min. 2 Cycles
Strobe
host_meta0_req_accept
Min. 2 Cycles
host_meta0_data_req
tp
tp
meta_host0_data[127:0]
meta_host0_byteenable[15:0]
meta_host0_data_status[3:0]
One to One
4 Cycles [Min./Max]
meta_host0_data_valid
Strobe
meta_host0_transfer_complete

FIGURE 6-66 Single Write Request With Read Bubble‘

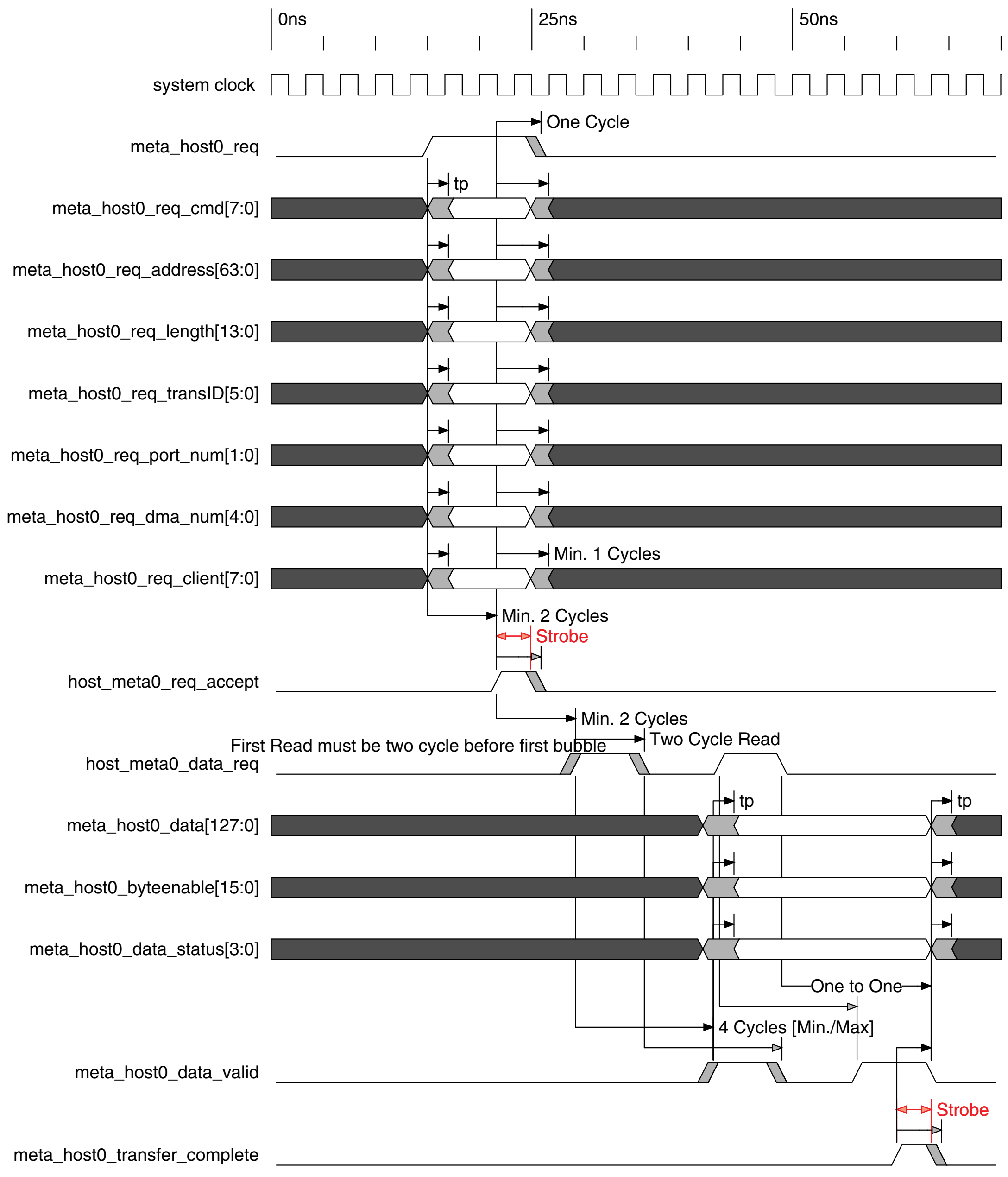

0ns
25ns
50ns
system clock
meta_host0_req
One Cycle
tp
meta_host0_req_cmd[7:0]
meta_host0_req_address[63:0]
meta_host0_req_length[13:0]
meta_host0_req_transID[5:0]
meta_host0_req_port_num[1:0]
meta_host0_req_dma_num[4:0]
Min. 1 Cycles
meta_host0_req_client[7:0]
Min. 2 Cycles
Strobe
host_meta0_req_accept
Min. 2 Cycles
Two Cycle Read
First Read must be two cycle before first bubble
host_meta0_data_req
tp
tp
meta_host0_data[127:0]
meta_host0_byteenable[15:0]
meta_host0_data_status[3:0]
One to One
4 Cycles [Min./Max]
meta_host0_data_valid
Strobe
meta_host0_transfer_complete

FIGURE 6-67 Two Write Requests

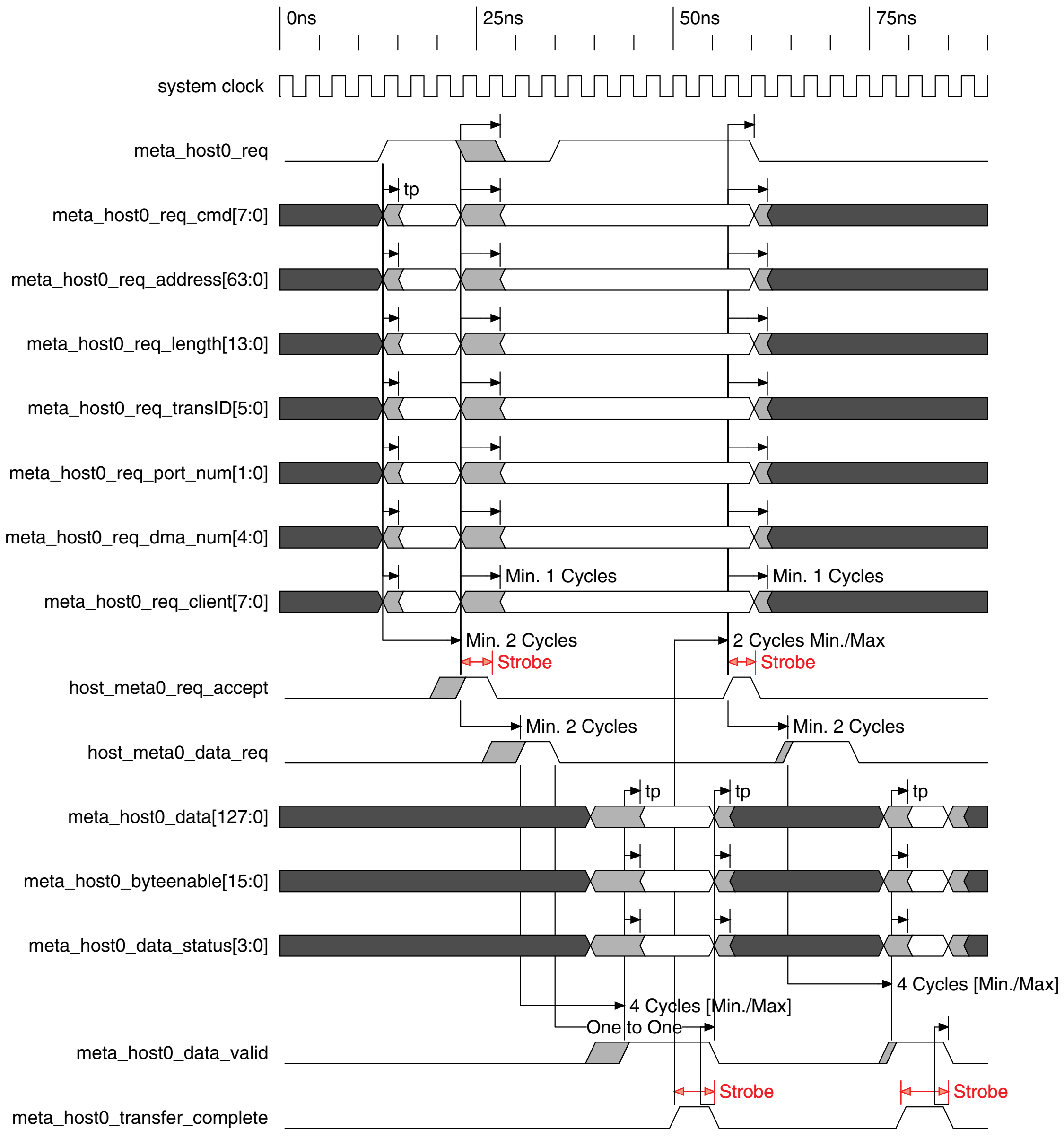
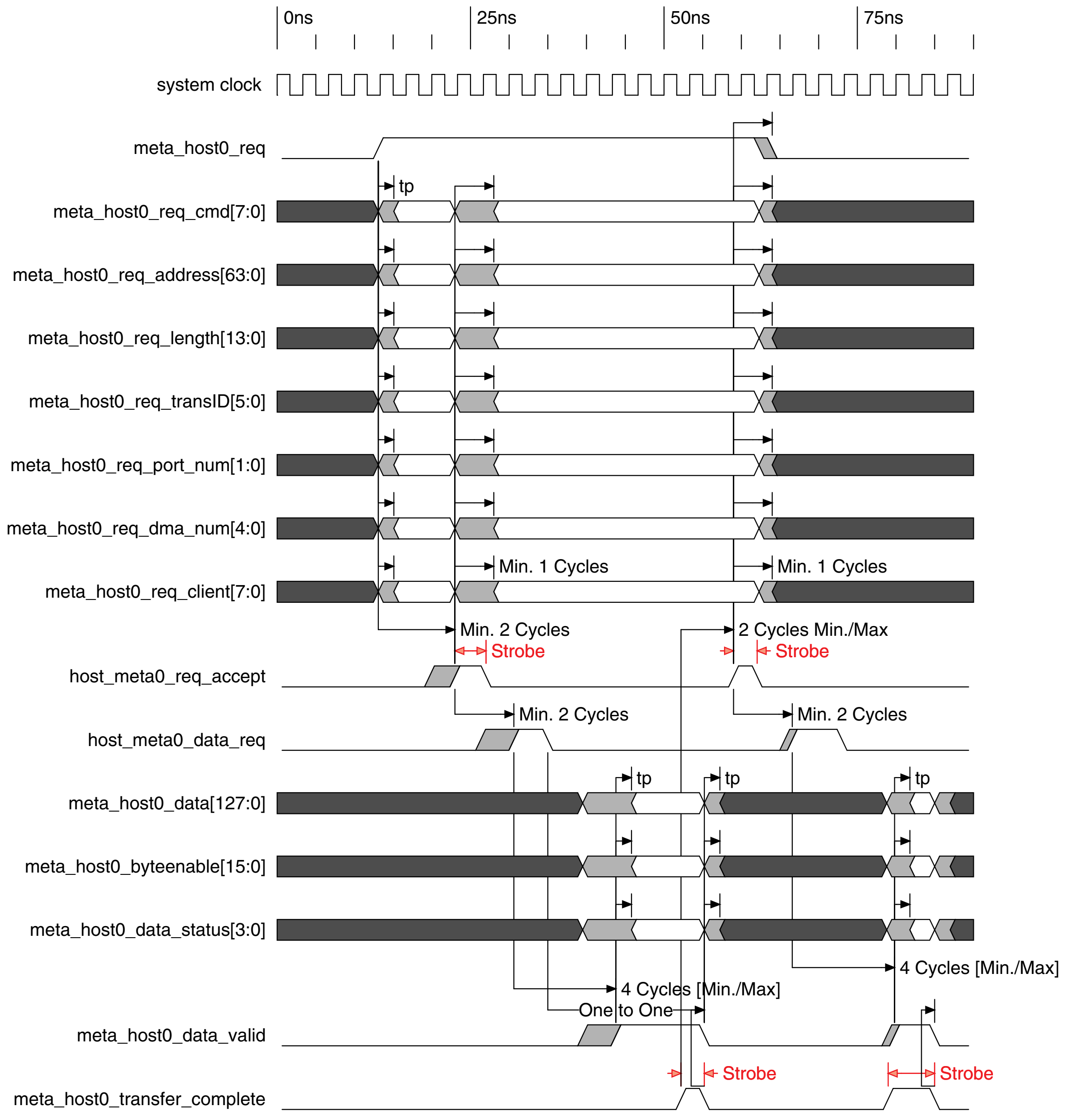


FIGURE 6-68 Back to Back Write Requests



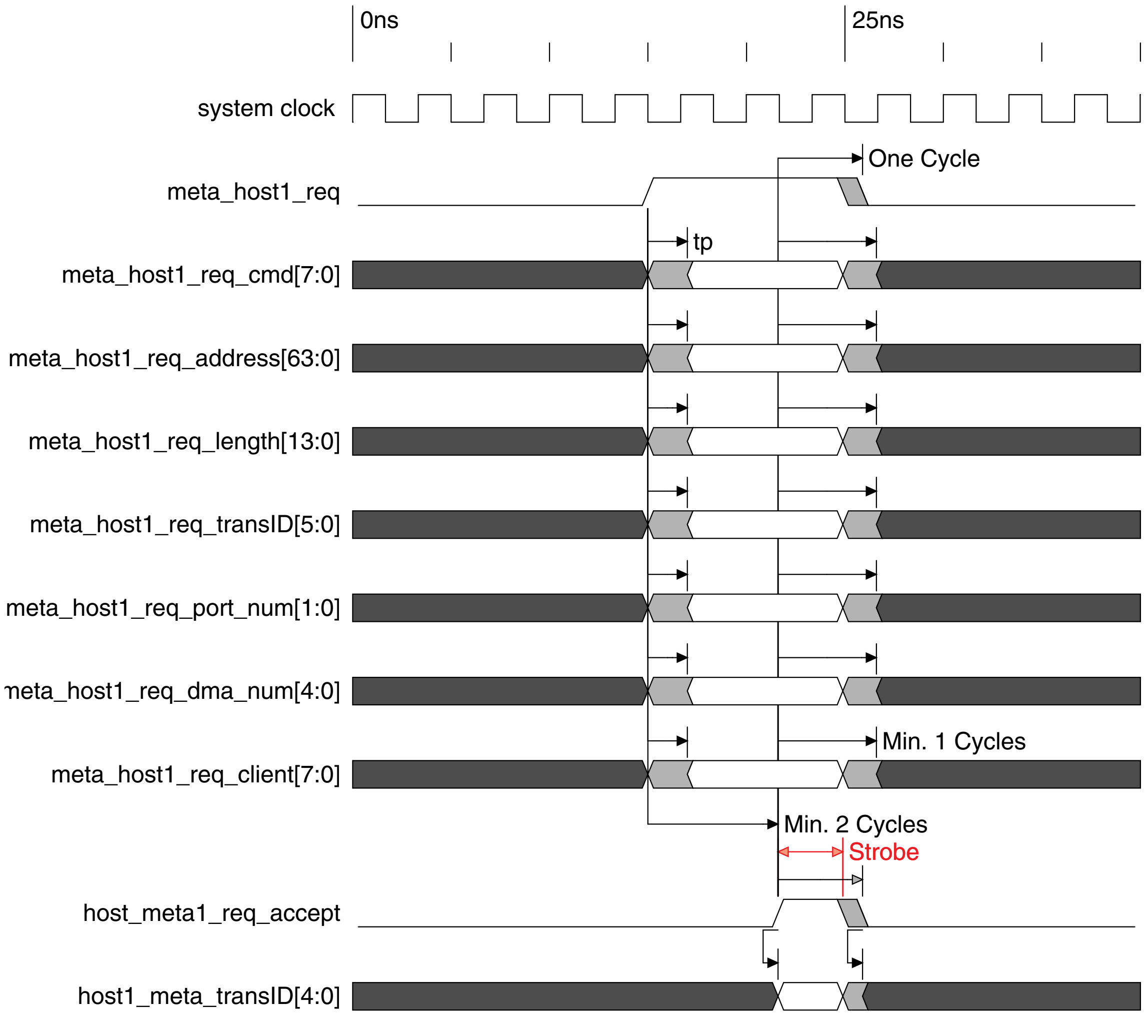
6.25.5 Read Request Interface Signals

TABLE 6-35 Read Request Meta Signals

Signal Name	Size	From/To	Description
Read REQUEST Transaction Type and Transaction Control			Nomenclature: Prefixes dmc_meta1 and meta_dmc1 signifies read interface signal.
meta_host1_req_cmd	8	Meta->host	Commands: *Memory Read.* Command Request Encoding: [7:5] Reserved; [4] Ordered=1, Non-Ordered=0; [3] 64-bit addressing=1, 32-bit addressing=0; [2:0] Memory Read=000; Reserved=1x0,x10,xx1. Note: IO, Cfg, Msg, and Cpl type commands are not supported by the REQUEST Queue.
meta_host1_req_address	64	Meta->host	Memory Address: *64-bit address, 32-bit address.*
meta_host1_req_length	14	Meta->host	Data Length (bytes)
meta_host1_req_transID	6	Meta->host	Transaction Identification
meta_host1_req_port_num	2	Meta->host	Port Number
meta_host1_req_dma_num	5	Meta->host	Channel Number
meta_host1_req_client	8	Meta->host	Requesting Client (vector, one-hot enc)
meta_host1_req	1	Meta->host	REQUEST Queue Request
host_meta1_req_accept	1	host->Meta	Grant REQUEST Queue Request
Read REQUEST Data and Data Control			
N/A			
Read REQUEST host Error Flag			
host_meta1_req_errors	1	host->Meta	Flag to report errors back to NIU. Flag is asynchronous with respect to read REQUEST events.

6.25.6 Read Request Interface Timing

FIGURE 6-69 Single Read Request



0ns
25ns
system clock
One Cycle
meta_host1_req
tp
meta_host1_req_cmd[7:0]
meta_host1_req_address[63:0]
meta_host1_req_length[13:0]
meta_host1_req_transID[5:0]
meta_host1_req_port_num[1:0]
meta_host1_req_dma_num[4:0]
Min. 1 Cycles
meta_host1_req_client[7:0]
Min. 2 Cycles
Strobe
host_meta1_req_accept
host1_meta_transID[4:0]

FIGURE 6-70 Two Read Requests

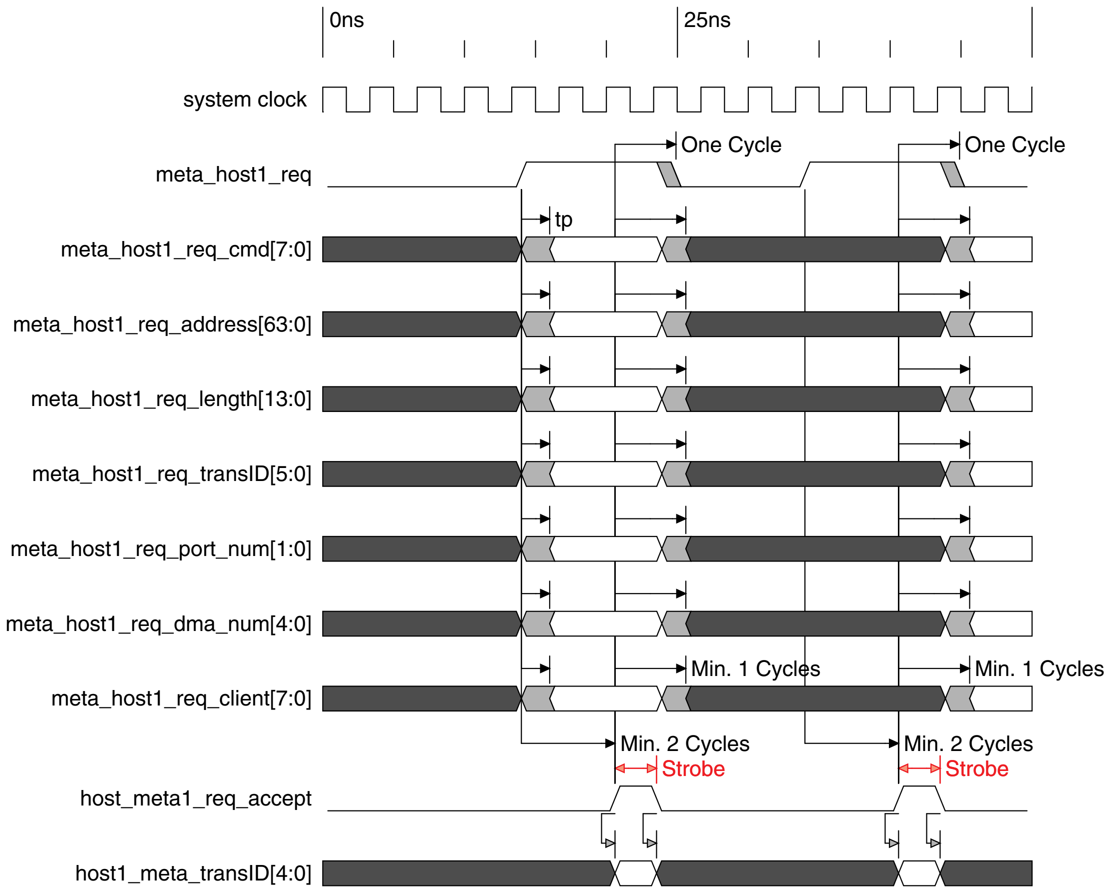


FIGURE 6-71 Back to Back Read Requests

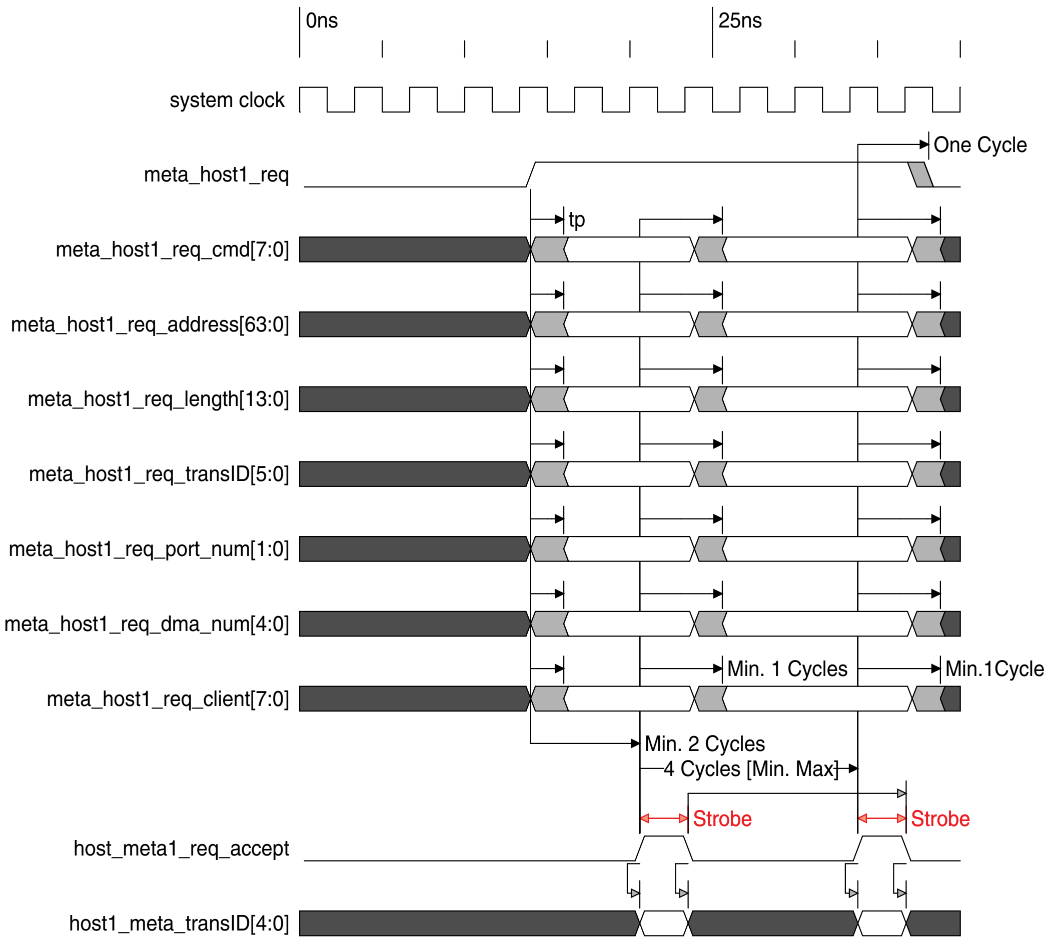


TABLE 6-36 Clock Cycles Between Conditional Event

Event$_{x-y}$	Minimum	Maximum	Comments
t_{1-2}	2		
t_{2-3}	2	2	
t_{2-a}	2	2	
t_{a-b}	4	4	
t_{b-c}	0	(Length/16)-1	
t_{c-2}	2	2	

6.25.7 Response Interface Signals

Example of response packet. Please note this is not meant to be an explicit implementation.

TABLE 6-37 Read Response Meta Signals

Signal Name	Size	From/To	Description
RESPONSE Transaction Type and Transaction Control			
host_meta_resp_cmd	8	host->Meta	Command Requests: *Completion with Data, and Completion without Data.* Command Request Encoding: [7:5] Reserved; [4:3] Error Type, [2:0] Completion with Data=101, Completion without Data=110. Note: Mem, IO, Cfg, and Msg type commands are not supported by the RESPONSE Queue.
host_meta_resp_address	64	host->Meta	Memory Address: *64-bit address, 32-bit address.*
host_meta_resp_length	14	host->Meta	Data Length (bytes)
host_meta_resp_transID	6	host->Meta	Transaction Identification
host_meta_resp_port_num	2	host->Meta	Port Number
host_meta_resp_dma_num	5	host->Meta	Channel Number
host_meta_resp_client	8	host->Meta	Requesting Client (vector, one-hot encoded)
host_meta_resp_ready	1	host->Meta	Validate RESPONSE Queue Request

TABLE 6-37 Read Response Meta Signals *(Continued)*

Signal Name	Size	From/To	Description
host_meta_resp_cmd_status	4	host->Meta	Indicates that at transaction time-out has occurred. Valid values are 4'h0 and 4'hF. 4'hF indicates a transaction time-out.
meta_host_resp_accept	8	Meta->host	Grant RESPONSE Queue Request. (Per client)
host_meta_resp_complete	8	host->Meta	Fragment complete. (Per client).
host_meta_transfer_cmpl	8	host->Meta	Transfer Complete. No additional transfers for this transaction. (Per client).
RESPONSE Data and Data Control			
host_meta_data	128	host->Meta	Packet Data.
host_meta_resp_byteenable	16	host->Meta	First/Last Byte Enable
host_meta_data_status	4	host->Meta	Packet Transfer Status: [3:2] Error Type, [1:0] Reserved. Note: Data transfers are in 128-bit units.
host_meta_data_valid	8	host->Meta	Meta Acknowledges Burst Transfer. (Per client).
RESPONSE NIU Error Flag			
N/A			
RESPONSE Flow Control			
N/A			Infinite Credits

6.25.8 Response Interface Timing

FIGURE 6-72 Read Response with Data

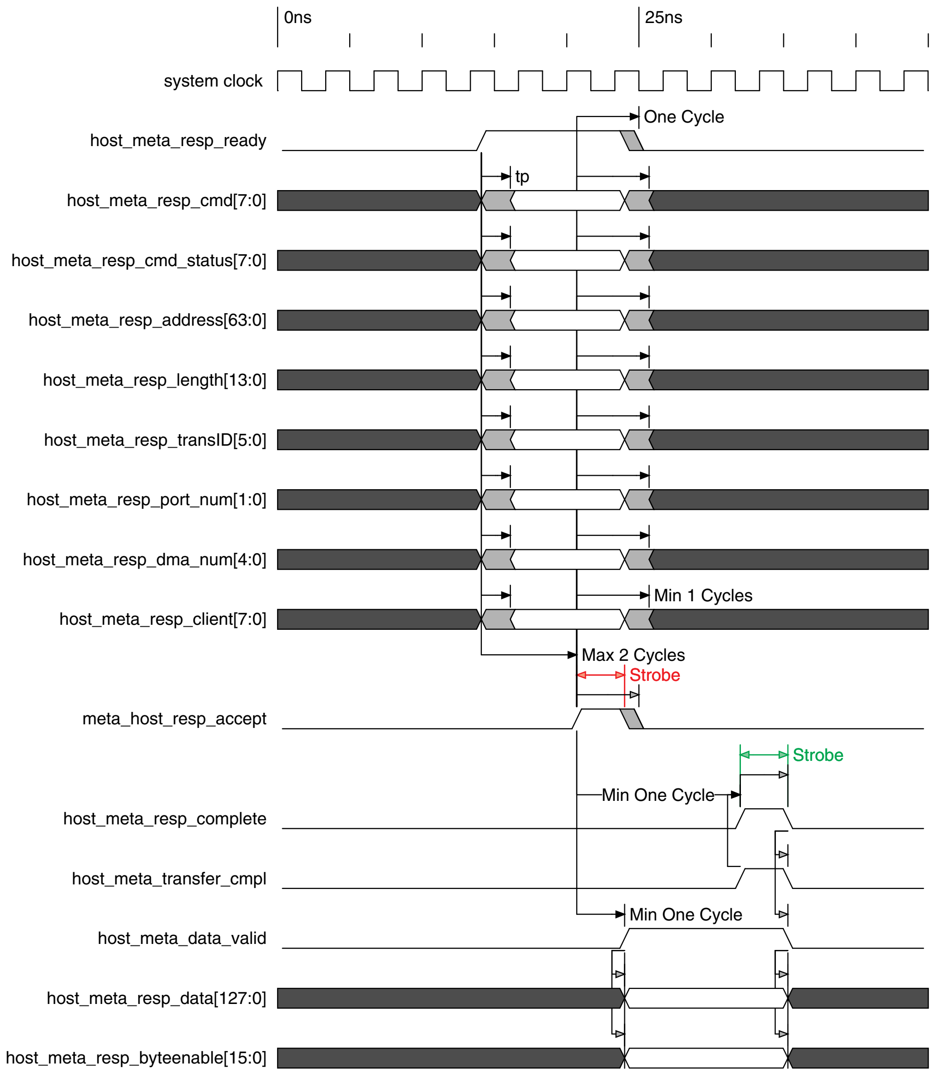

0ns
25ns
system clock
host_meta_resp_ready
One Cycle
tp
host_meta_resp_cmd[7:0]
host_meta_resp_cmd_status[7:0]
host_meta_resp_address[63:0]
host_meta_resp_length[13:0]
host_meta_resp_transID[5:0]
host_meta_resp_port_num[1:0]
host_meta_resp_dma_num[4:0]
Min 1 Cycles
host_meta_resp_client[7:0]
Max 2 Cycles
Strobe
meta_host_resp_accept
Strobe
Min One Cycle
host_meta_resp_complete
host_meta_transfer_cmpl
Min One Cycle
host_meta_data_valid
host_meta_resp_data[127:0]
host_meta_resp_byteenable[15:0]

FIGURE 6-73 Back to Back Read Responses

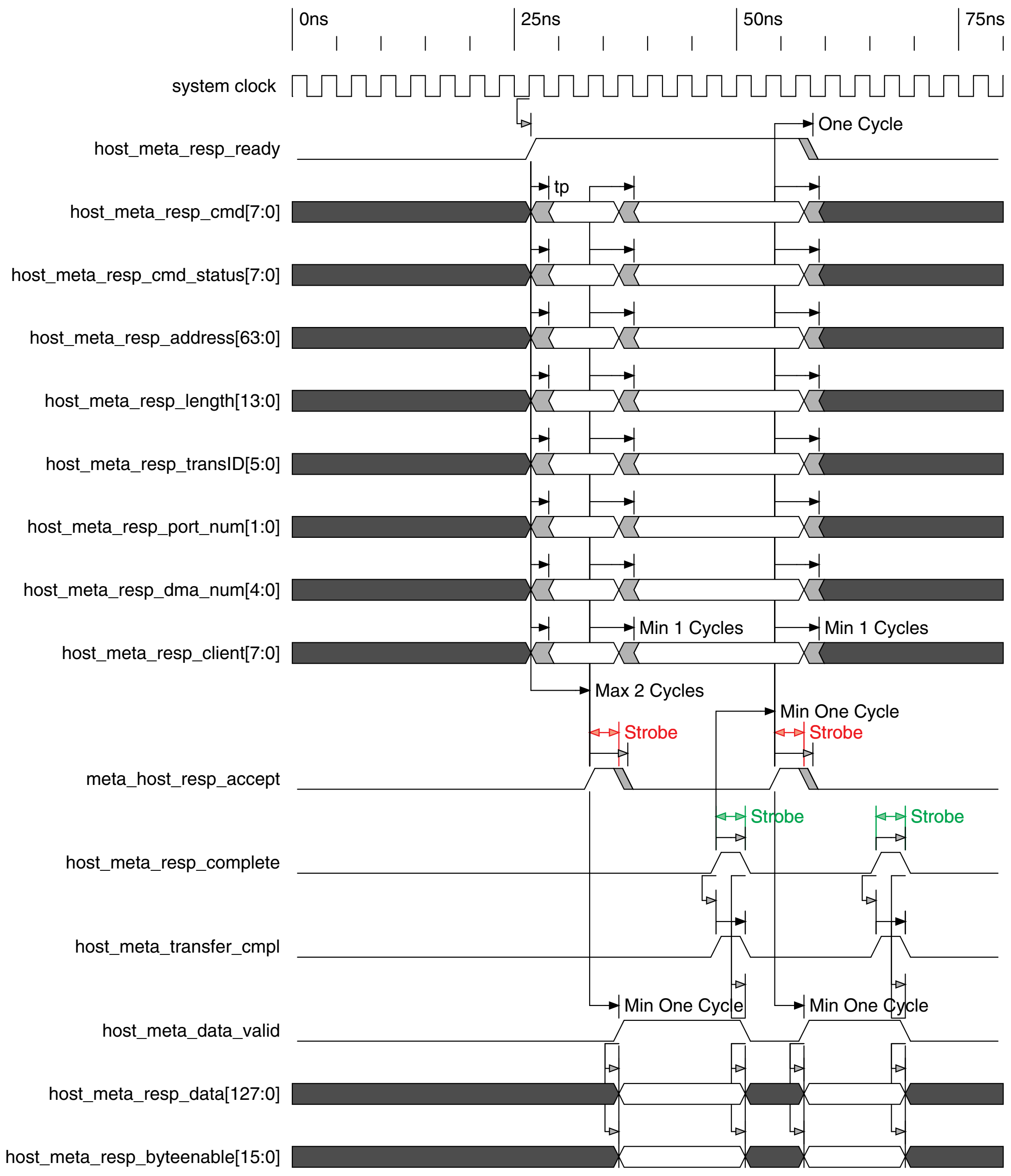


FIGURE 6-74 Segmented Read Responses

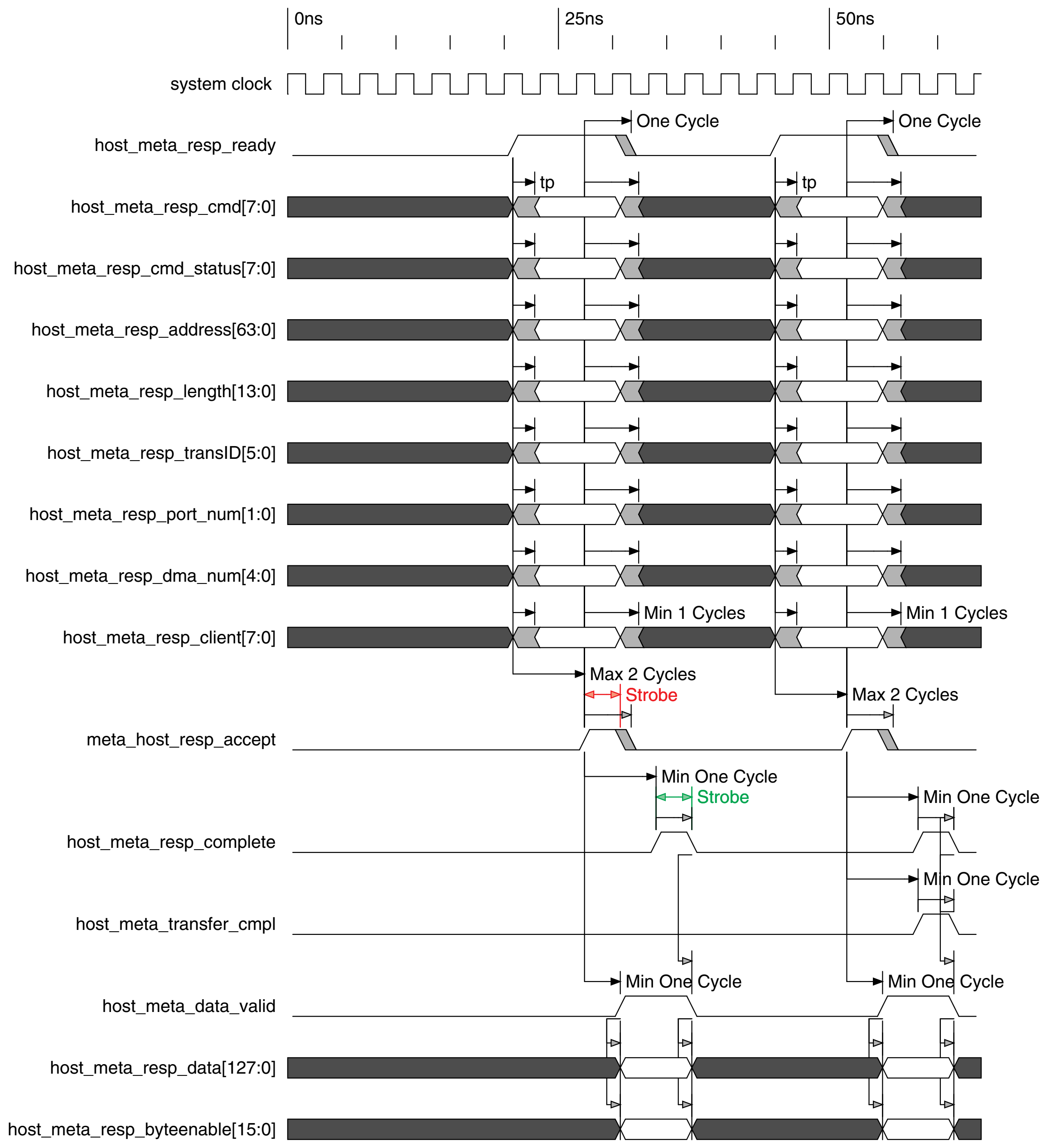


FIGURE 6-75 Transaction Time-out

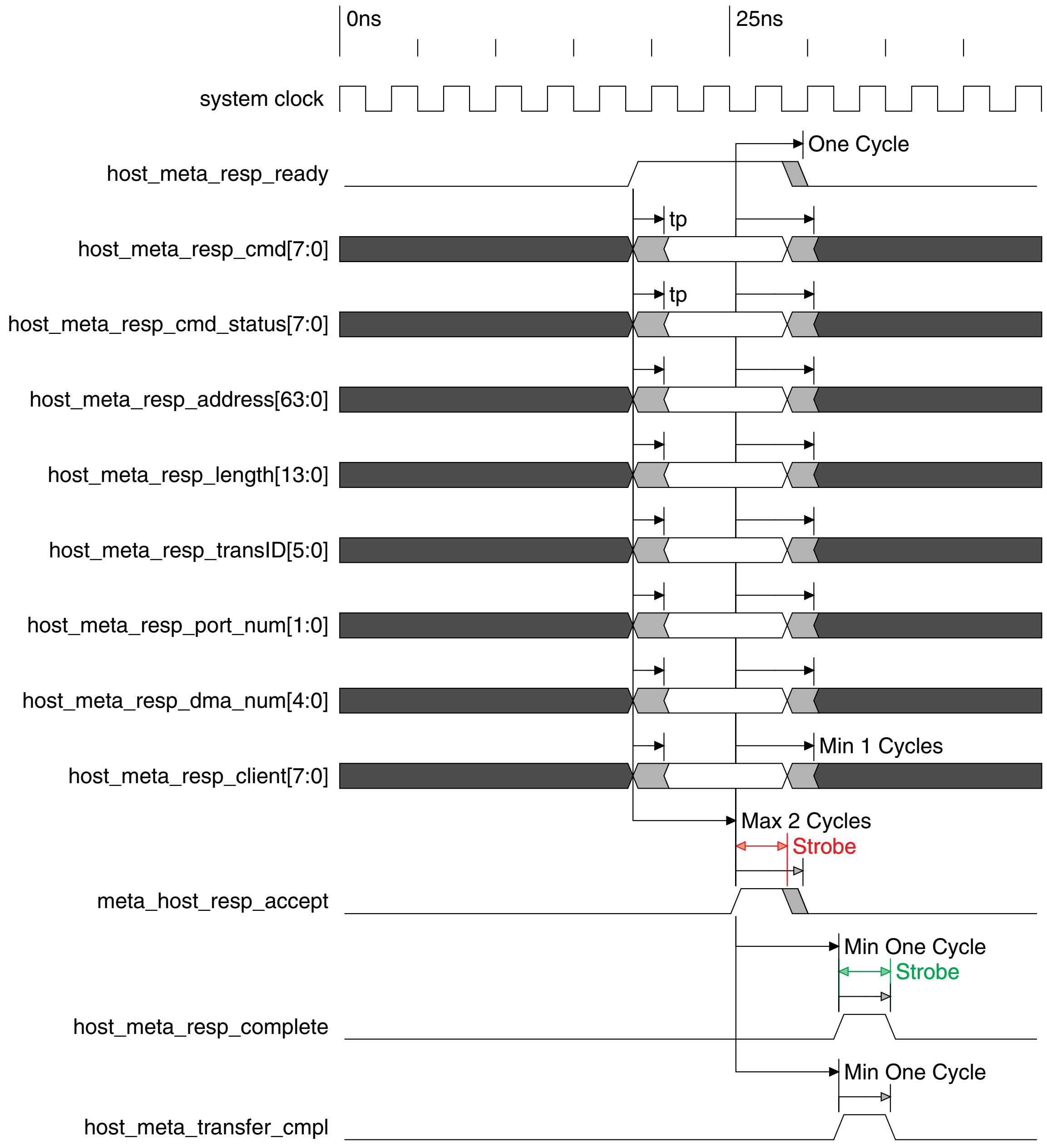
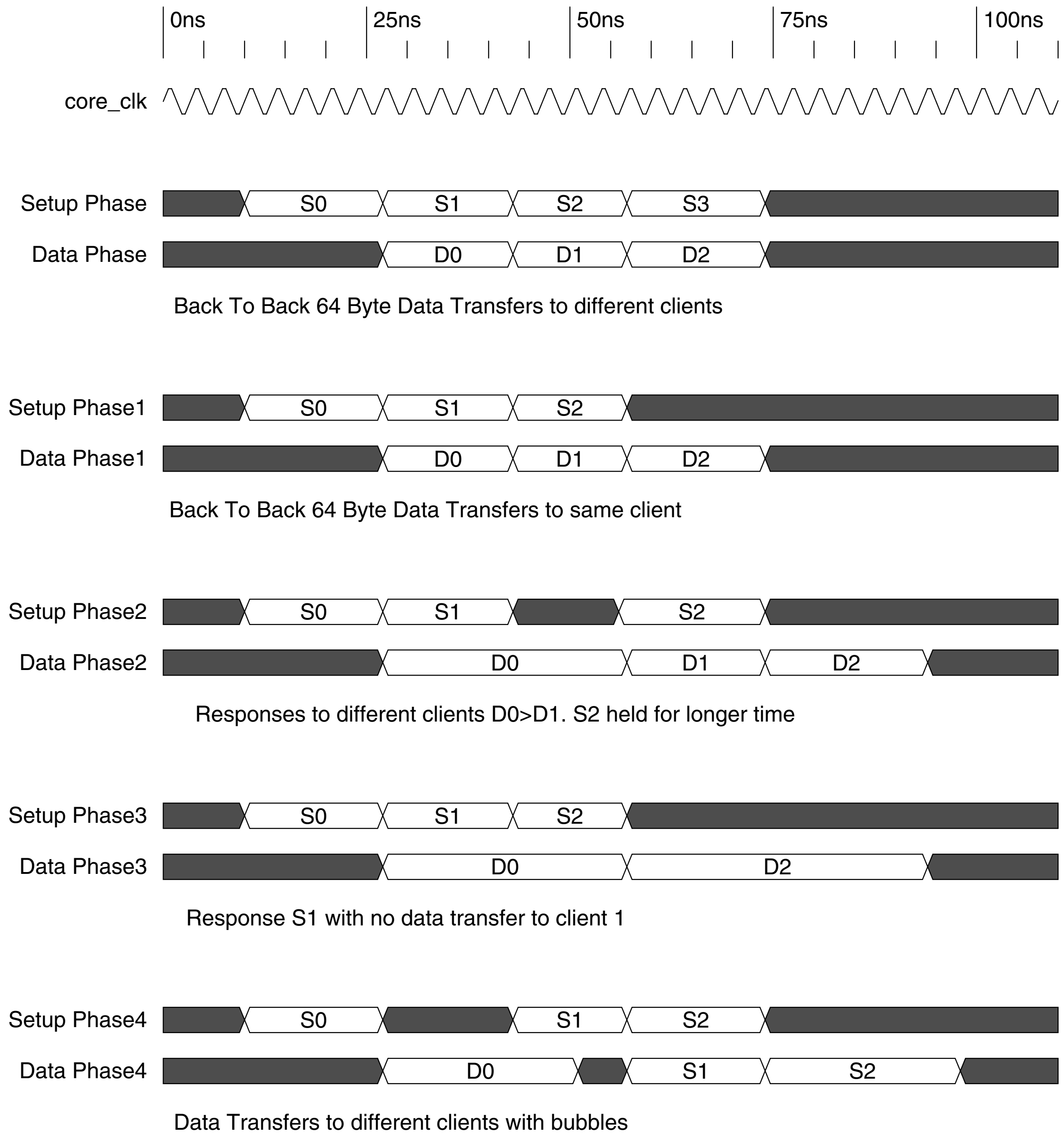


FIGURE 6-76 Example of Response Transactions


0ns
25ns
50ns
75ns
100ns
core_clk
Setup Phase
S0
S1
S2
S3
Data Phase
D0
D1
D2
Back To Back 64 Byte Data Transfers to different clients
Setup Phase1
S0
S1
S2
Data Phase1
D0
D1
D2
Back To Back 64 Byte Data Transfers to same client
Setup Phase2
S0
S1
S2
Data Phase2
D0
D1
D2
Responses to different clients D0>D1. S2 held for longer time
Setup Phase3
S0
S1
S2
Data Phase3
D0
D2
Response S1 with no data transfer to client 1
Setup Phase4
S0
S1
S2
Data Phase4
D0
S1
S2
Data Transfers to different clients with bubbles

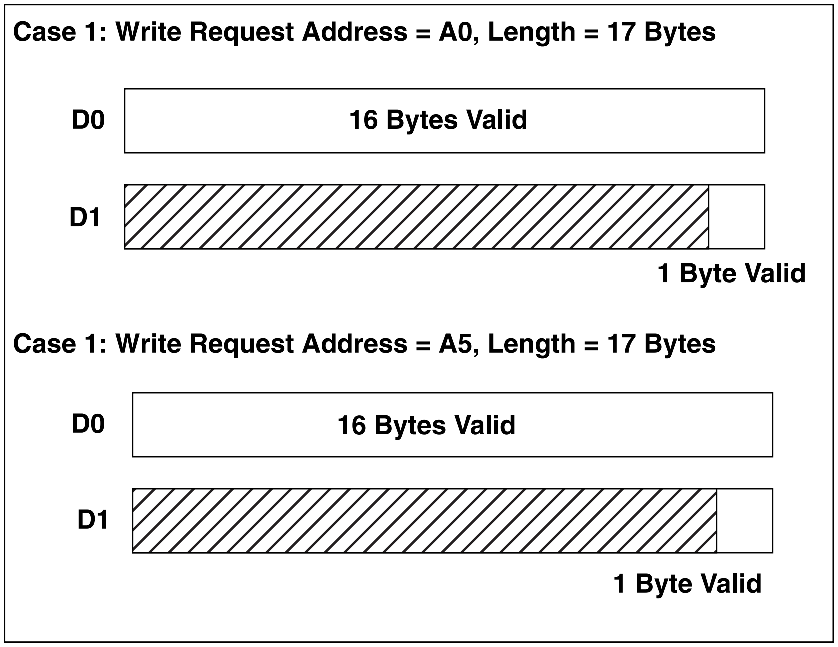
6.25.9 Alignment for Request and Response Data

FIGURE 6-77 and FIGURE 6-78 show the alignment of the internal data path of the NIU design for Request and Response data.

6.25.9.1 Request Data Format

In the case of Requests, data and Address are byte aligned, with byte enables only on the last data phase. Where D0 and D1 are the two successive data phases, with D1 being the last data phase.

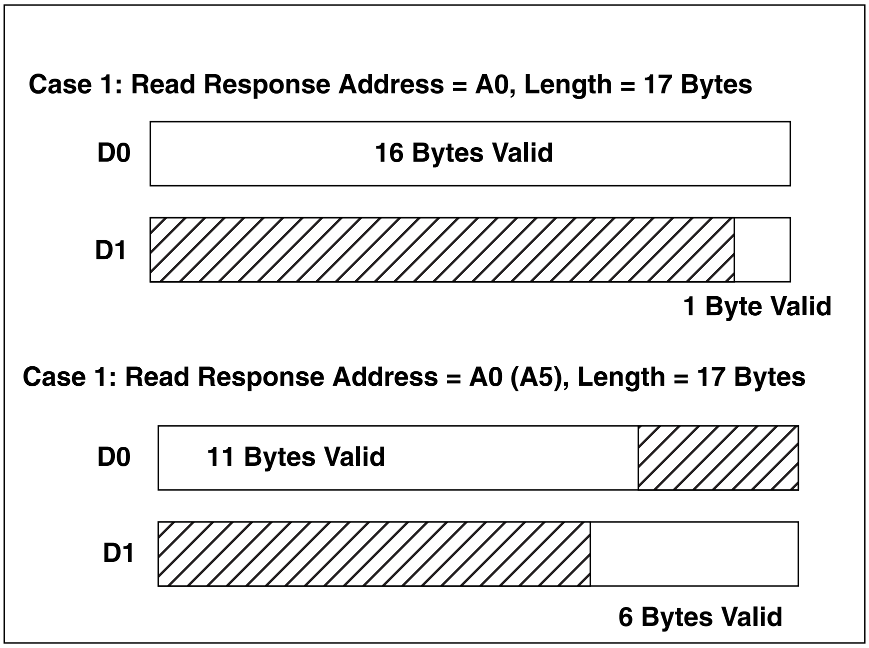
FIGURE 6-77 Request Data Format


Case 1: Write Request Address = A0, Length = 17 Bytes
D0
16 Bytes Valid
D1
1 Byte Valid
Case 1: Write Request Address = A5, Length = 17 Bytes
D0
16 Bytes Valid
D1
1 Byte Valid

6.25.9.2 Response Data Format

In the case of a response, the data is 16 byte aligned with the appropriate byte enable driven, as in FIGURE 6-78. Where D0 and D1 are the two successive data phases.

FIGURE 6-78 Response Data Format

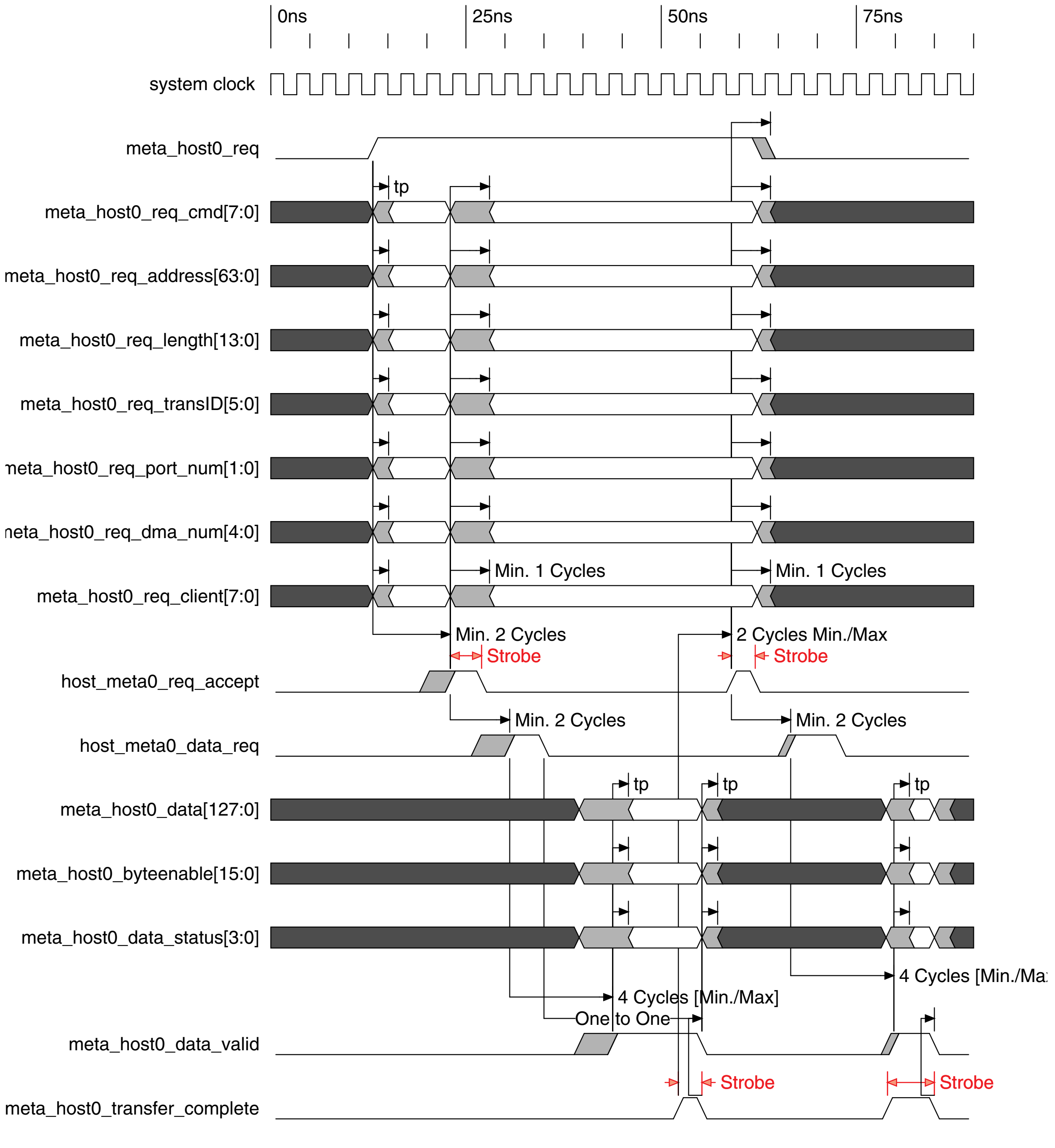

Case 1: Read Response Address = A0, Length = 17 Bytes
D0
16 Bytes Valid
D1
1 Byte Valid
Case 1: Read Response Address = A0 (A5), Length = 17 Bytes
D0
11 Bytes Valid
D1
6 Bytes Valid

6.25.10 Acknowledgment Signals

TABLE 6-38 Acknowledgement Queue Meta Signals

Signal Name	Size	From/To	Description
ACKNOWLEDGEMENT Transaction Type and Transaction Control			
host_meta_ack_cmd	8	host->Meta	Command Requests: Completion with Data, and Completion without Data. Command Request Encoding: [7:5] Reserved; [4:3] Error Type, [2:0] Completion with Data=101, Completion without Data=110. Note: Mem, IO, Cfg, and Msg type commands are not supported by the ACKNOWLEDGEMENT Queue.
host_meta_ack_address	64	host->Meta	Memory Address: *64-bit address, 32-bit address.*
host_meta_ack_length	14	host->Meta	Data Length (bytes)
host_meta_ack_transID	6	host->Meta	Transaction Identification
host_meta_ack_port_num	2	host->Meta	Port Number
host_meta_ack_dma_num	5	host->Meta	Channel Number
host_meta_ack_client	8	host->Meta	Requesting Client (vector, one-hot encoded)
host_meta_ack_ready	1	host->Meta	Validate ACKNOWLEDGEMENT Queue Request
meta_host_ack_accept	8	Meta->host	Grant ACKNOWLEDGEMENT Queue Request. (Per client)
host_meta_ack_complete	8	host->Meta	Fragment complete. (Per client).
host_meta_ack_transfer_comple te	8	host->Meta	Transfer Complete. No additional transfers for this transaction. (Per client).
ACKNOWLEDGEMENT Data and Data Control			
N/A			
ACKNOWLEDGEMENT NIU Error Flag			
N/A			
ACKNOWLEDGEMENT Flow Control			
N/A			Infinite Credits

FIGURE 6-79 Acknowledgement Waveform


0ns
25ns
50ns
75ns
system clock
meta_host0_req
meta_host0_req_cmd[7:0]
meta_host0_req_address[63:0]
meta_host0_req_length[13:0]
meta_host0_req_transID[5:0]
neta_host0_req_port_num[1:0]
neta_host0_req_dma_num[4:0]
meta_host0_req_client[7:0]
host_meta0_req_accept
host_meta0_data_req
meta_host0_data[127:0]
meta_host0_byteenable[15:0]
meta_host0_data_status[3:0]
meta_host0_data_valid
meta_host0_transfer_complete
tp
Min. 1 Cycles
Min. 1 Cycles
Min. 2 Cycles
Strobe
2 Cycles Min./Max
Strobe
Min. 2 Cycles
Min. 2 Cycles
tp
tp
tp
4 Cycles [Min./Ma
4 Cycles [Min./Max]
One to One
Strobe
Strobe

6.26 Interrupt Microarchitecture Specification

6.26.1 Interrupt Overview

The NIU supports up to 69 logic devices that can be separated into five categories:

Receive DMA Channels	LD[0:15]
Reserved	LD[16:31]
Transmit DMA Channels	LD[32:47]
Reserved	LD[48:62]
MDIO Interface (MIF)	LD[63]
Ethernet MACs	LD[64:65]
Reserved	LD[66:67]
System Errors.	LD[68]

Multiple interrupt sources can be associated with a single logic device. Each interrupt source can be individually masked by setting corresponding mask bits.

Each logic device has two associated interrupt flags i.e. the ldf[i] and ldf[j] flags. ldf[i] is used for normal datapath events such as packet arrivals or transmission completion events, while ldf[j] is used for error or non critical datapath events such as MAC and MIF interrupts.

Each logic device interrupt flag can be masked individually by setting the logic device ldf_mask[1:0] bits.

Each of the logic devices can be bound to one of 64 logic device groups sharing a system interrupt, by assigning a logic device group number.

The interrupt status of a logic device group is reflected by the Logic Device State Vector (LDSV[68:0]) associated with that group. Each bit of the LDSV represents the interrupt status of corresponding logic device bound to that group. The LDSV can be accessed by software to determine the pending interrupt status.

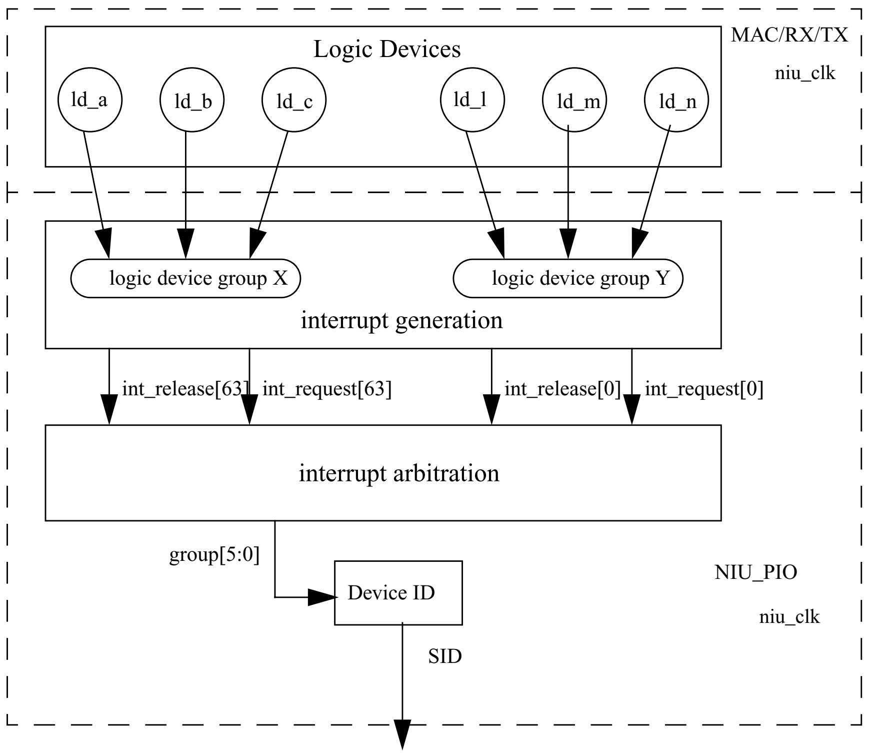
The interrupt logic implementation spans multiple NIU blocks.

The logic devices are implemented in the MAC/RXC/TXC blocks. The interrupt event generation an arbitration is implemented in the NIU_PIO block

FIGURE 6-80 is a high level representation of the Interrupt datapath.

The subsequent paragraphs describe the microarchitecture of the components of the interrupt logic in more detail.

FIGURE 6-80 Interrupt Datapath Block Diagram



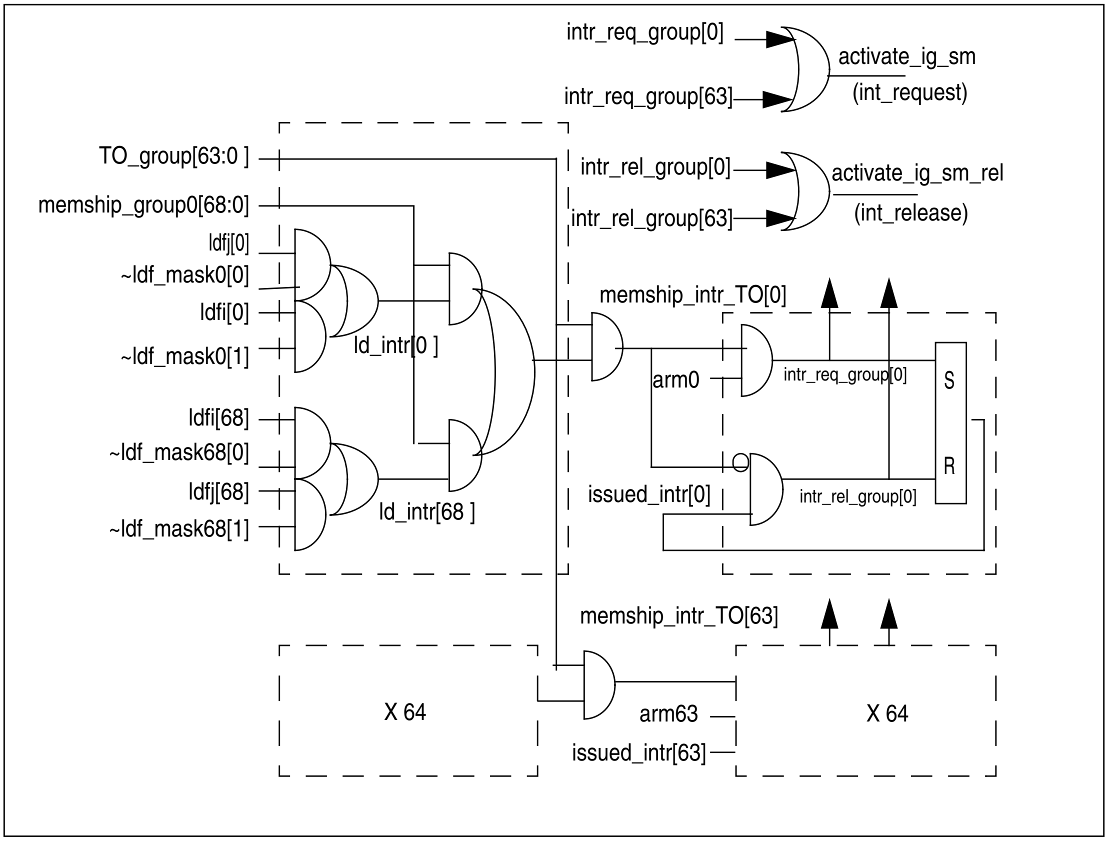
6.26.2 Interrupt Event Generation

A logic device group interrupt request event is generated when a least one logic device bound to that group flags an interrupt (flag bit set), the logic device mask bit is cleared, the logic device group interrupt timeout timer has expired and the logic device group arm bit is set.

A logic device group release request event is generated, when the logic device group’s arm bit is cleared (by hardware) and all the pending interrupts are serviced by software by either clearing the flag bits, setting the mask bit, and/or re-setting of the timeout counter.

When a logic device group has generated an interrupt event, (interrupt request or interrupt release request), the group arbitrates for access to the interrupt queue for further processing of the interrupt.

FIGURE 6-81 Interrupt Diagram



6.26.3 Interrupt Request Arbitration

The 64 interrupt groups can assert interrupt requests and release requests independently.

The interrupt arbitration logic arbitrates between the various requests for access to the interrupt request queue. The arbitration algorithm is a priority based scheme that assigns lowest priority to the group that has been serviced most recently. Interrupts requests have priority over interrupt release requests.

The arbitration logic generates the group number of the request that won the arbitration. The logic has to guarantee that the correct value of the group number is maintained throughout the arbitration sequence by latching all the interrupt information at the start of the sequence.

The interrupt arbitration state machine manages the dispatching of the interrupt request.

6.26.4 Interrupt SID Generation

Once the arbitration sequence has completed, the 6-bit group number associated with the request is used to index into the 64 entry DeviceID table that contains a unique SID value for each of the groups.

SID[6:5] contains the function number assigned to the group, and SID[4:0] represents the five bit SID vector. Note that only 32 unique SID values can be assigned per function. As a consequence, a maximum of 32 logic device groups can be assigned to a single function.

FIGURE 6-82 NIU Interrupt Arbitration Datapath

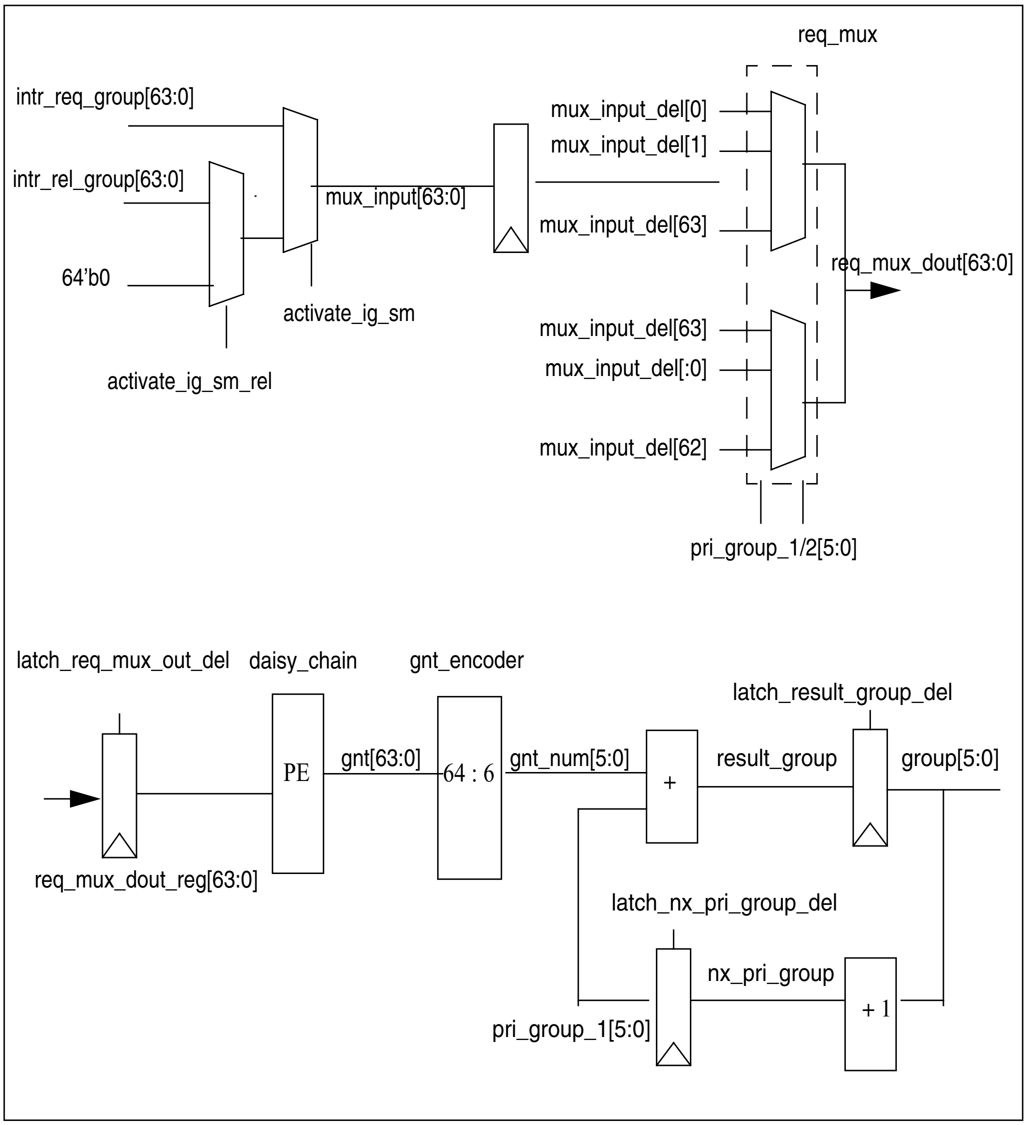
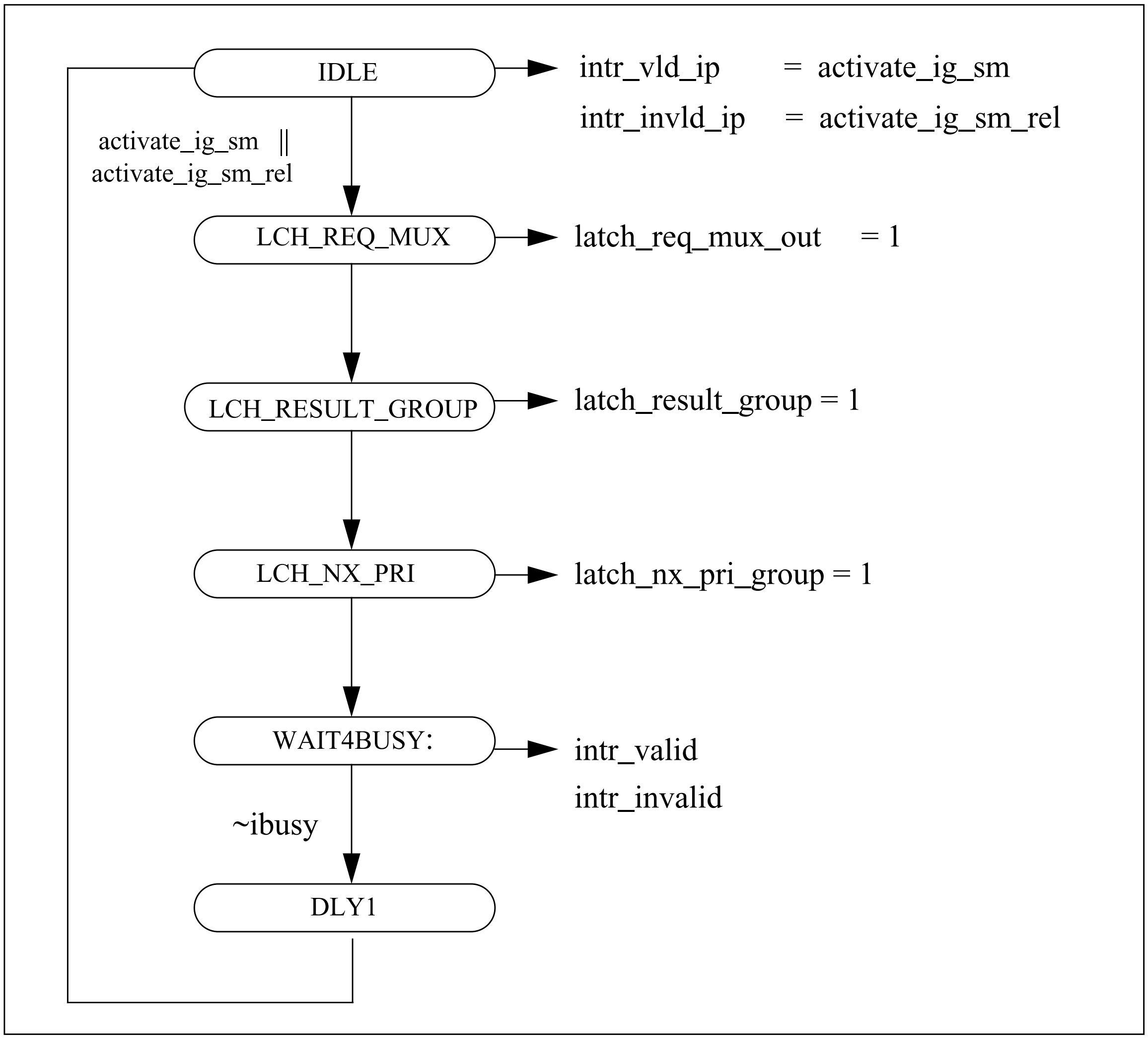


FIGURE 6-83 Interrupt Arbitration State Machine

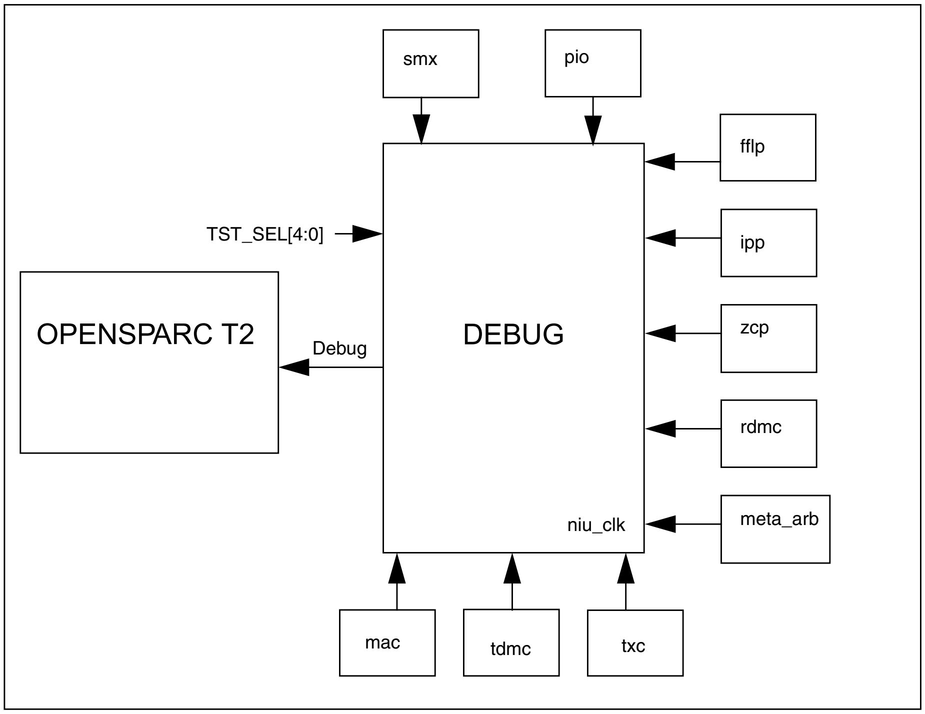


6.27 Debug Microarchitecture Specification

6.27.1 Overview

- Debug bus architecture that facilitates the observability of:
  - Block level control signals.
  - Internal Clock signals.
- The debug architecture is implemented at two levels:
  - Top level through RTST_SEL[4:0] pins.
  - Block level through debug select register

FIGURE 6-84 Top-level Debug Diagram



6.27.2 Debug Port

The debug architecture is implemented at two levels, block level and top level.

At the block level, a debug select register is programmed by software to select a group of block level debug control signals that are placed on the block debug bus. Details on the programming and selection of the block level control signal groups are described in the PRM chapters of the individual blocks.

At the top level, the external RTST_SEL[4:0] pins select which one of the block level debug busses are propagated to the debug port data bus:

TABLE 6-39 Debug Data Port

RTST_SEL[4:0]	Debug Port Data
5'h11	txc_debug_port
5'h12	tdmc_debug_port
5'h13	rdmc_debug_port
5'h14	zcp_debug_port
5'h15	ipp_debug_port
5'h16	fflp_debug_port
5'h17	pio_debug_port
5'h18	mac_debug_port
5'h19	zero
5'h1a	meta_arb_debug_port
5'h1b	smx_debug_port_port

Default debug port value is zero when the select ranges from 5'h1to 5'10.

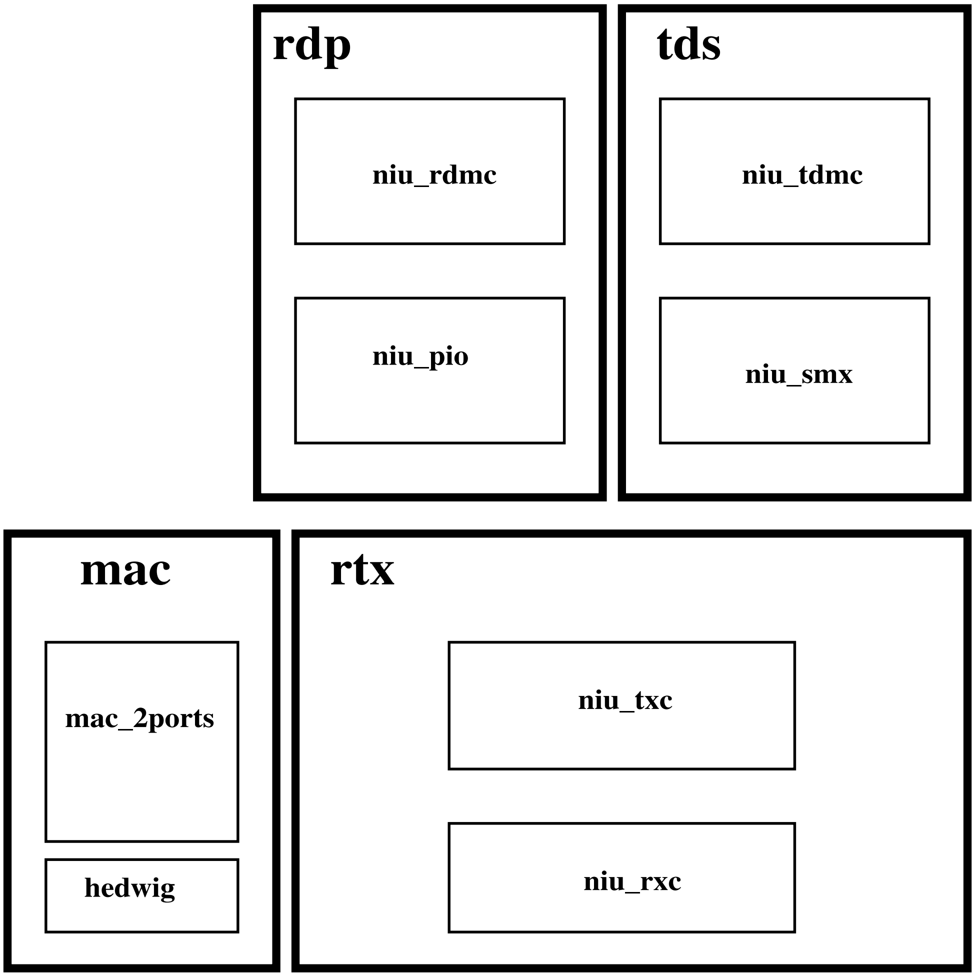
In addition to debug control signals, internal clock signals can be observed shown in TABLE 6-40.

TABLE 6-40 Debug Internal Clock Signals

RTST_SEL[4:0]	Debug_clock0	Debug_clock1
5'h0:	1'b0	1'b0
5'h8:	mac_debug_clock0_divby8	mac_debug_clock1_divby8
5'h9:	1'b0	1'b0
default:	niu_clk_div8	niu_clk_div8

6.28 N2 NIU Design for Test

FIGURE 6-85 NIU Block Diagram



6.28.1 Membist Block Diagrams

FIGURE 6-86 Membist TDS Block Diagram

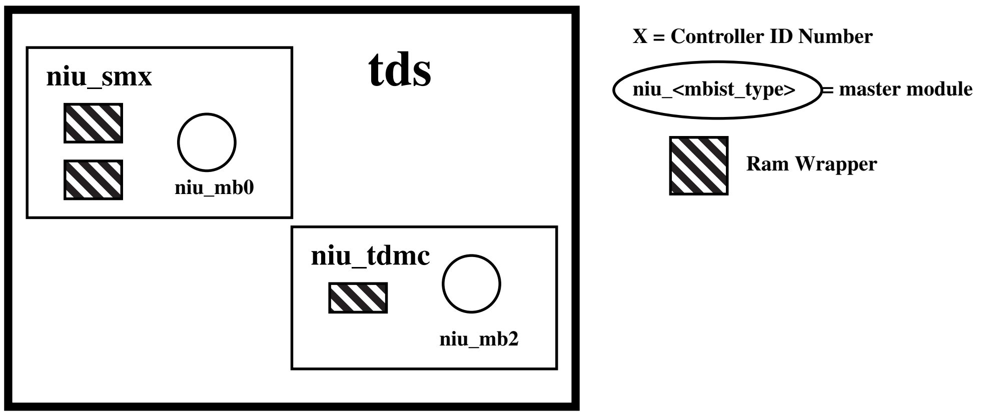


FIGURE 6-87 Membist RDP Block Diagram

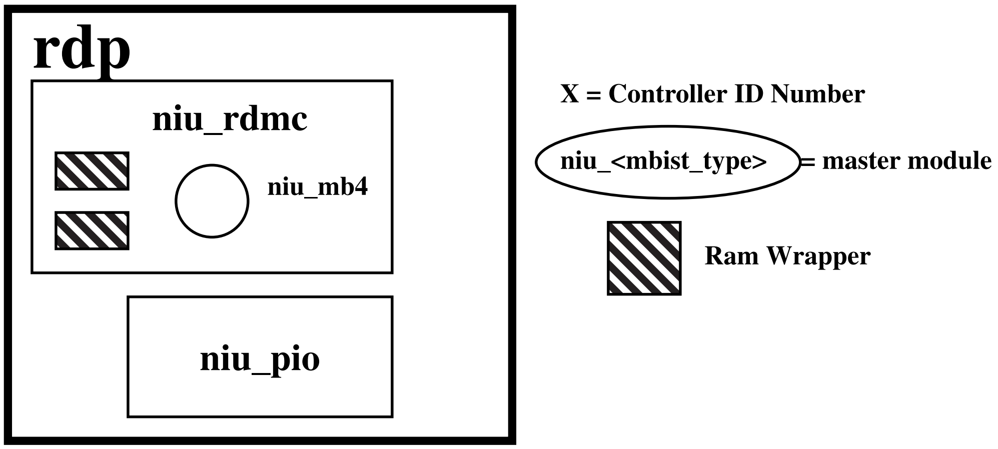
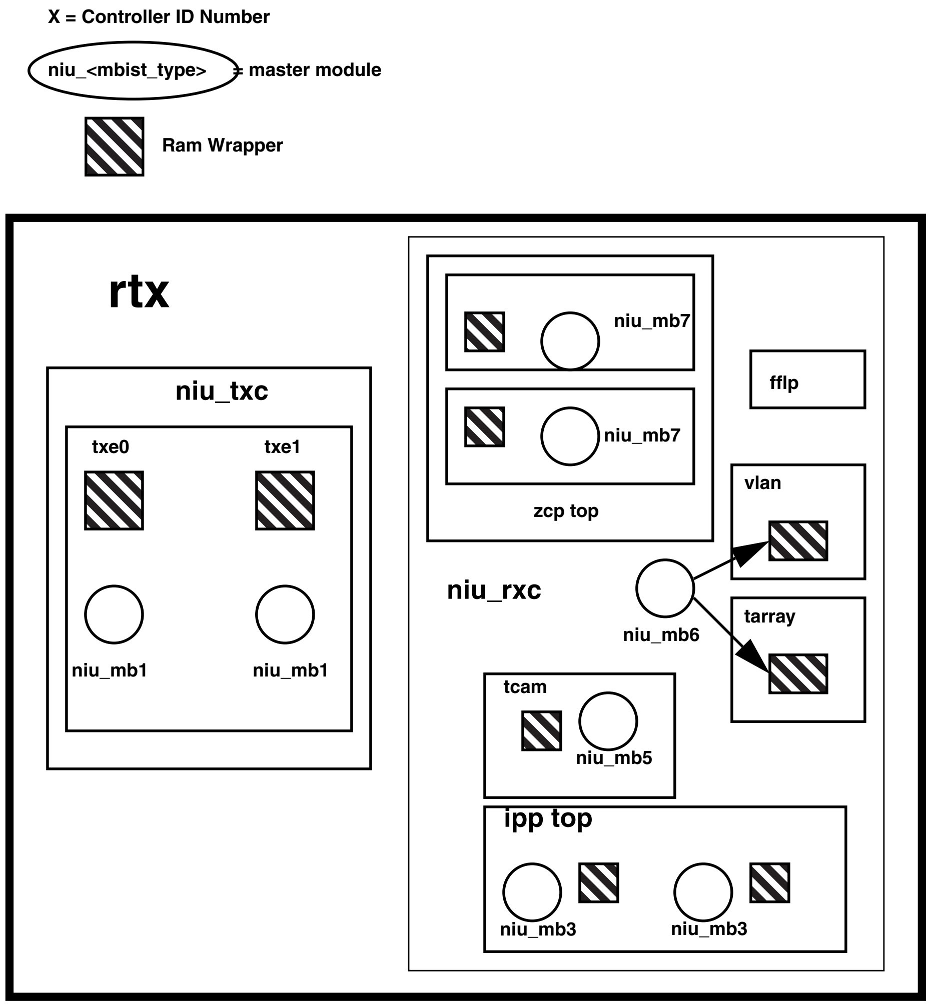


FIGURE 6-88 Membist RTX Block Diagram



6.28.2 MAC Wrapper DFT Clocks

TABLE 6-41 MAC DFT Clock Ports

MAC DFT Clock Port Name	Type	Description	Mux with Functional Clock
mac_312rx_test_clk	Input	312 MHz RX DFT clock	rbc0_a,b,c,d
mac_312tx_test_clk	Input	312 MHz TX DFT clock	tx_clk0_312, tx_clk1_312
mac_156rx_test_clk	Input	156 MHz RX DFT clock	rx_clk0, rx_clk1
mac_156tx_test_clk	Input	156 MHz TX DFT clock	tx_clk0, tx_clk1
mac_125rx_test_clk	Input	125 MHz TX DFT clock	rx_nbclk0, rx_nbclk1
mac_125tx_test_clk	Input	125 MHz TX DFT clock	tx_nbclk0, tx_nbclk1

TABLE 6-42 MAC Functional vs. DFT Clock Groupings

Functional Clock	External DFT Clock	Comment
rbc0_a0, rbc0_b0, rbc0_c0, rbc0_d0, rbc0_a1, rbc0_b1, rbc0_c1, rbc0_d1	mac_312rx_test_clock	312 Mhz RX clocks
rx_clk0, rx_clk1	mac_156rx_test_clock	156 Mhz RX clocks
rx_nbclk0, rx_nbclk1	mac_125rx_test_clock	125 Mhz RX clocks
tx_clk0, tx_clk1	mac_156tx_test_clock	156 Mhz TX clocks
tx_nbclk0, tx_nbclk1	mac_125tx_test_clock	125 Mhz TX clocks
tx_clk0_312, tx_clk1_312	mac_312tx_test_clock	312 Mhz TX clocks NIU Lane 1 transmit clk.

6.28.3 MAC Wrapper DFT Port Names

TABLE 6-43 MAC DFT Scan Ports

MAC Scan Port Names	Type	Description
tcu_scan_mode	Input	(FUNC_MODE) Mux select for dft clocking = 1
tcu_scan_en	Input	Shift/Capture select
tcu_aclk	Input	Shift clock stage 1
tcu_bclk	Input	Shift clock stage 2
tcu_mac_io_clk_stop	Input	Stop Clock

TABLE 6-43 MAC DFT Scan Ports *(Continued)*

MAC Scan Port Names	Type	Description
tcu_pce_ov	Input	Pulse Clock Enable Override
scan_in	Input	Scan input
scan_out	Output	Scan output

6.28.4 RDP Wrapper DFT Port Names

TABLE 6-44 RDP DFT Ports

RDMC Scan Port Names	Type	Description	
tcu_scan_en	Input	Shift/Capture select	
tcu_aclk	Input	Shift clock stage 1	
tcu_bclk	Input	Shift clock stage 2	
tcu_rdp_io_clk_stop	Input	Stop Clock	
tcu_pce_ov	Input	Pulse Clock Enable Override	
scan_in	Input	Scan input	
scan_out	Output	Scan output	
tcu_array_wr_inhibit	Input		
tcu_mbist_bisi_en	Input		
tcu_se_scancollar_in	Input		
tcu_se_scancollar_out	Output		
tcu_rdp_rdmc_mbist_start	Input		
rdp_rdmc_tcu_mbist_fail	Output		
rdp_rdmc_tcu_mbist_done	Output		
rdp_rdmc_mbist_scan_in	Input	Mbist scan input	
rdp_rdmc_mbist_scan_out	Output	Mbist scan output	
rdp_tcu_dmo_data_out[39:0]	Output	40 bit DMO data bus	New

6.28.5 TDS Wrapper DFT Port Names

TABLE 6-45 TDS DFT Ports

TDS Scan Port Names	Type	Description
tcu_scan_en	Input	Shift/Capture select
tcu_aclk	Input	Shift clock stage 1
tcu_bclk	Input	Shift clock stage 2
tcu_tds_io_clk_stop	Input	Stop Clock
tcu_pce_ov	Input	Pulse Clock Enable Override

TABLE 6-45 TDS DFT Ports *(Continued)*

TDS Scan Port Names	Type	Description
scan_in	Input	Scan input
scan_out	Output	Scan output
tcu_array_wr_inhibit	Input	
tcu_mbist_bisi_en	Input	
tcu_se_scancollar_in	Input	
tcu_se_scancollar_out	Output	
tcu_tds_smx_mbist_start	Input	
tds_smx_tcu_mbist_fail	Output	
tds_smx_tcu_mbist_done	Output	
tcu_tds_tdmc_mbist_start	Input	
tds_tdmc_tcu_mbist_fail	Output	
tds_tdmc_tcu_mbist_done	Output	
tds_mbist_scan_in	Input	Mbist scan input
tds_mbist_scan_out	Output	Mbist scan output
tds_tcu_dmo_data_out[39:0]	Output	40 bit DMO data bus to RDP New

6.28.6 RTX Wrapper DFT Port Names

TABLE 6-46 RTX DFT Ports

RTX Scan Port Names	Type	Description
tcu_scan_en	Input	Shift/Capture select
tcu_aclk	Input	Shift clock stage 1
tcu_bclk	Input	Shift clock stage 2
tcu_rtx_io_clk_stop	Input	Stop Clock
tcu_pce_ov	Input	Pulse Clock Enable Override
scan_in	Input	Scan input
scan_out	Output	Scan output
tcu_array_wr_inhibit	Input	
tcu_mbist_bisi_en	Input	
tcu_array_bypass	Input	
tcu_se_scancollar_in	Input	
tcu_se_scancollar_out	Output	
tcu_rtx_txe0_mbist_start	Input	
rtx_txe0_tcu_mbist_fail	Output	
rtx_txe0_tcu_mbist_done	Output	
tcu_rtx_txe1_mbist_start	Input	
rtx_txe1_tcu_mbist_fail	Output	
rtx_txe1_tcu_mbist_done	Output	
tcu_rtx_rxc_ipp0_mbist_start	Input	
rtx_rxc_ipp0_tcu_mbist_fail	Output	
rtx_rxc_ipp0_tcu_mbist_done	Output	
tcu_rtx_rxc_ipp1_mbist_start	Input	
rtx_rxc_ipp1_tcu_mbist_fail	Output	
rtx_rxc_ipp1_tcu_mbist_done	Output	
tcu_rtx_rxc_mb5_mbist_start	Input	
rtx_rxc_mb5_tcu_mbist_fail	Output	
rtx_rxc_mb5_tcu_mbist_done	Output	
tcu_rtx_rxc_mb6_mbist_start	Input	

TABLE 6-46 RTX DFT Ports *(Continued)*

RTX Scan Port Names	Type	Description	
rtx_rxc_mb6_tcu_mbist_fail	Output		
rtx_rxc_mb6_tcu_mbist_done	Output		
tcu_rtx_rxc_zcp0_mbist_start	Input		
rtx_rxc_zcp0_tcu_mbist_fail	Output		
rtx_rxc_zcp0_tcu_mbist_done	Output		
tcu_rtx_rxc_zcp1_mbist_start	Input		
rtx_rxc_zcp1_tcu_mbist_fail	Output		
rtx_rxc_zcp1_tcu_mbist_done	Output		
rtx_mbist_scan_in	Input	Mbist scan input	
rtx_mbist_scan_out	Output	Mbist scan output	
tcu_rtx_dmo_ctl[2:0]	Input	3 bit select from TCU	New
rtx_tcu_dmo_data_out[39:0]	Output	40 bit DMO data bus to RDP	New

6.28.7 SMX Module DFT Port Names

TABLE 6-47 SMX DFT Ports

SMX Scan Port Names	Type	Description
tcu_scan_en	Input	
tcu_aclk	Input	Shift clock stage 1
tcu_bclk	Input	Shift clock stage 2
tcu_clk_stop	Input	Stop Clock
tcu_array_wr_inhibit	Input	
tcu_mbist_bisi_en	Input	
tcu_pce_ov	Input	Pulse Clock Enable Override
tcu_se_scancollar_in	Input	
tcu_se_scancollar_out	Output	
tcu_tds_smx_mbist_start	Input	
tds_smx_tcu_mbist_fail	Output	
tds_smx_tcu_mbist_done	Output	
tds_smx_mbist_scan_in	Input	Mbist scan input
tds_smx_mbist_scan_out	Output	Mbist scan output

6.28.8 TDMC Module DFT Port Names

TABLE 6-48 TDMC DFT Ports

TDMC Scan Port Names	Type	Description
tcu_scan_en	Input	
tcu_aclk	Input	Shift clock stage 1
tcu_bclk	Input	Shift clock stage 2
tcu_clk_stop	Input	Stop Clock
tcu_pce_ov	Input	Pulse Clock Enable Override
tcu_array_wr_inhibit	Input	
tcu_mbist_bisi_en	Input	
tcu_se_scancollar_in	Input	
tcu_se_scancollar_out	Output	

TABLE 6-48 TDMC DFT Ports *(Continued)*

TDMC Scan Port Names	Type	Description	
tcu_tds_tdmc_mbist_start	Input		
tds_tdmc_tcu_mbist_fail	Output		
tds_tdmc_tcu_mbist_done	Output		
tds_tdmc_mbist_scan_in	Input	Mbist scan input	
tds_tdmc_mbist_scan_out	Output	Mbist scan output	
tds_tcu_dmo_data_out[39:0]	Output	Mbist DMO output	New

6.28.9 RDMC Module DFT Port Names

TABLE 6-49 RDMC DFT Ports

RDMC Scan Port Names	Type	Description	
tcu_scan_en	Input		
tcu_aclk	Input	Shift clock stage 1	
tcu_bclk	Input	Shift clock stage 2	
tcu_clk_stop	Input	Stop Clock	
tcu_array_wr_inhibit	Input		
tcu_mbist_bisi_en	Input		
tcu_array_bypass	Input		
tcu_pce_ov	Input	Pulse Clock Enable Override	
tcu_se_scancollar_in	Input		
tcu_se_scancollar_out	Output		
tcu_rdp_rdmc_mbist_start	Input		
rdp_rdmc_tcu_mbist_fail	Output		
rdp_rdmc_tcu_mbist_done	Output		
rdp_rdmc_mbist_scan_in	Input	Mbist scan input	
rdp_rdmc_mbist_scan_out	Output	Mbist scan output	
rdp_tcu_dmo_data_out[39:0]	Output	Mbist DMO output	New

6.28.10 TXC Module DFT Port Names

TABLE 6-50 TXC DFT Ports

TXC Scan Port Names	Type	Description
tcu_scan_en	Input	
tcu_aclk	Input	Shift clock stage 1
tcu_bclk	Input	Shift clock stage 2
tcu_clk_stop	Input	Stop Clock
tcu_pce_ov	Input	Pulse Clock Enable Override
tcu_array_wr_inhibit	Input	
tcu_mbist_bisi_en	Input	

TABLE 6-50 TXC DFT Ports *(Continued)*

TXC Scan Port Names	Type	Description	
tcu_se_scancollar_in	Input		
tcu_se_scancollar_out	Output		
tcu_rtx_txc_txe0_mbist_start	Input		
rtx_txc_txe0_tcu_mbist_fail	Output		
rtx_txc_txe0_tcu_mbist_done	Output		
rtx_txc_txe0_dmo_dout	Output	Mbist DMO output	New
tcu_rtx_txc_txe1_mbist_start[39:0]	Input		
rtx_txc_txe1_tcu_mbist_fail	Output		
rtx_txc_txe1_tcu_mbist_done	Output		
rtx_txc_txe1_dmo_dout[39:0]	Output	Mbist DMO output	New
rtx_txc_txe_mbist_scan_in	Input	Mbist scan input	
rtx_txc_txe_mbist_scan_out	Output	Mbist scan output	

6.28.11 RXC Module DFT Port Names

TABLE 6-51 RXC DFT Ports

RXC Scan Port Names	Type	Description	
tcu_scan_en	Input		
tcu_aclk	Input	Shift clock stage 1	
tcu_bclk	Input	Shift clock stage 2	
tcu_clk_stop	Input	Stop Clock	
tcu_pce_ov	Input	Pulse Clock Enable Override	
tcu_array_wr_inhibit	Input		
tcu_mbist_bisi_en	Input		
tcu_array_bypass	Input		
tcu_se_scancollar_in	Input		
tcu_se_scancollar_out	Output		
tcu_rtx_rxc_ipp0_mbist_start	Input		
rtx_rxc_ipp0_tcu_mbist_fail	Output		
rtx_rxc_ipp0_tcu_mbist_done	Output		
rtx_rxc_ipp0_mb3_mbist_scan_in	Input	Mbist scan input	
rtx_rxc_ipp0_mb3_mbist_scan_out	Output	Mbist scan output	
rtx_rxc_ipp0_mb3_dmo_dout[39:0]	Output	Mbist DMO output	New
tcu_rtx_rxc_ipp1_mbist_start	Input		
rtx_rxc_ipp1_tcu_mbist_fail	Output		
rtx_rxc_ipp1_tcu_mbist_done	Output		
rtx_rxc_ipp1_mb3_mbist_scan_in	Input	Mbist scan input	
rtx_rxc_ipp1_mb3_mbist_scan_out	Output	Mbist scan output	
rtx_rxc_ipp1_mb3_dmo_dout[39:0]	Output	Mbist DMO output	New
tcu_rtx_rxc_mb5_mbist_start	Input		
rtx_rxc_mb5_tcu_mbist_fail	Output		
rtx_rxc_mb5_tcu_mbist_done	Output		
rtx_rxc_tcam_cntrl_mbist_scan_in	Input	Mbist scan input	
rtx_rxc_tcam_cntrl_mbist_scan_out	Output	Mbist scan output	
tcu_rtx_rxc_mb6_mbist_start	Input		

TABLE 6-51 RXC DFT Ports *(Continued)*

RXC Scan Port Names	Type	Description	
rtx_rxc_mb6_tcu_mbist_fail	Output		
rtx_rxc_mb6_tcu_mbist_done	Output		
rtx_tcam_vlan_mbist_scan_in	Input	Mbist scan input	
rtx_tcam_vlan_mbist_scan_out	Output	Mbist scan output	
rtx_rxc_vlan_mb6_dmo_dout[39:0]	Output	Mbist DMO output	New
tcu_rtx_rxc_zcp0_mbist_start	Input		
rtx_rxc_zcp0_tcu_mbist_fail	Output		
rtx_rxc_zcp0_tcu_mbist_done	Output		
rtx_rxc_zcp0_mb7_mbist_scan_in	Input	Mbist scan input	
rtx_rxc_zcp0_mb7_mbist_scan_out	Output	Mbist scan output	
rtx_rxc_zcp0_mb7_dmo_dout[39:0]	Output	Mbist DMO output	New
tcu_rtx_rxc_zcp1_mbist_start	Input		
rtx_rxc_zcp1_tcu_mbist_fail	Output		
rtx_rxc_zcp1_tcu_mbist_done	Output		
rtx_rxc_zcp1_mb7_mbist_scan_in	Input	Mbist scan input	
rtx_rxc_zcp1_mb7_mbist_scan_out	Output	Mbist scan output	
rtx_rxc_zcp1_mb7_dmo_dout[39:0]	Output	Mbist DMO output	New

6.28.12 Controller to SRAM Mapping

Controller Type	SRAMs	Depth X Width Physical *** App. Depth/Width in ()	< Array Name>	2X Wrapper Clk Name	e_fuse
niu_mb0	SMX Read/Write Table	64 X 148(146)	smx_table	TDS	none
	SMX Store/Forward	32 X 148(146)	smx_store	TDS	none
niu_mb1	Port 0 Transmit Store & Forward	1024(640)X 152(152)	xmit_store	2X RTX	efuhdr1a_p0
	Port 0 Transmit Re-align Buffer	1024 X 152(152)	xmit_realign	2X RTX	efuhdr1b_p0
niu_mb1	Port 1 Transmit Store & Forward	1024(640) X 152(152)	xmit_store	2X RTX	efuhdr1a_p1
	Port 1 Transmit Re-align Buffer	1024 X 152(152)	xmit_realign 2X	RTX	efuhdr1b_p1
niu_mb2	TX DMA Descriptor Cache	256 X 152(148)	tx_dma_desc 2X	TDS	efuhdr2
niu_ mb3	Port 0 Receive Data Fifo	1024 X 152(146)	rx_data_fifo 2X	RTX	efuhdr3_p0
	Port 0 Pre-Buffer Header	64 X 152(146)	prebuf_header	RTX	none
niu_mb3	Port 1 Receive Data Fifo	1024 X 152(146)	rx_data_fifo	2X RTX	efuhdr3_p1
	Port 1 Pre-Buffer Header	64 X 148(146)	prebuf_header	RTX none	
niu_ mb4	RX DMA Descriptor Cache	256 X 152(148)	rx_dma_desc	2X RDP	efuhdr4a
	RX DMA Completion Shadow	256 X 152(148)	rx_dma_comp	2X RDP	efuhdr4a
niu_mb5	TCAM Controller	128 X 200(200)	tcam_cntrl	RTX	none
niu_mb6	TCAM Array	128 X 42(42)	tcam_array	RTX	none
	VLAN Table	4096 X 9 (9)	vlan	RTX	efuhdr6
niu_mb7	Port 0 Control Fifo (ZCP)	512 X 152(146)	cntrl_fifo_zcp	2X RTX	efuhdr7_p0
niu_mb7	Port 1 Control Fifo (ZCP)	512 X 152(146)	cntrl_fifo_zcp	2X RTX	efuhdr7_p1

*** Ram wrapper has application width for wiring.

6.28.13 Scan and MEMBIST Signals for NIU SRAMs

Name	Type	Description
tcu_scan_en	input	(NIU top level only)
tcu_ack	input	(Top level and Propagate to RAM hierarchy)
tcu_bclk	input	(Top level and Propagate to RAM hierarchy)
tcu_se_scancollar_in	input	(Top level and Propagate to RAM hierarchy)
tcu_se_scancollar_out	input	(Needed for SRAM output flops...none in NIU)
tcu_clk_stop	input	(Top level and Propagate to RAM hierarchy)
tcu_pce_ov	input	(Top level and Propagate to RAM hierarchy)
tcu_arrary_wr_inhibit	input	(Top level and Propagate to RAM hierarchy)
tcu_arrary_bypass	input	(Top level and Propagate to RAM hierarchy)
tcu_mbist_bisi_en	input	(Top level and Propagate to RAM hierarchy)
<mbist_name>_<array_name>_scan_in	input	(Top level and Propagate to RAM hierarchy)
<mbist_name>_<array_name>_scan_out	output	(Propagate from RAM hierarchy to top level)
tcu_niu_mbist_start	input	(Top level and Propagate to RAM hierarchy)
niu_tcu_mbist_fail	output	(Propagate from Controller to top level)
niu_tcu_mbist_done	output	(Propagate from Controller to top level)

6.28.14 SRAM Array Signal Names

TABLE 6-52 Array Signals

Description	Type	Name Format	NIU *
Read_Enable	Input	<cluster_name_mbi_<array_name>_rd_en	mbi_rd_en
Write_Enable	Input	<cluster_name_mbi_<array_name>_wr_en	mbi_wr_en
Address	Input	<cluster_name>_mbi_addr	mbi_adr
Wdata	Input	<cluster_name>_mbi_wdata	mbi_wdata
Run	Input	<cluster_name>_mbi_run	mbi_run

6.28.15 Membist Controller Port Names

TABLE 6-53 NIU_MB0 SMX Mbist Controller Ports

Description	Type	Signal Name	Changed
Read_Enable	Output	niu_mb0_smx_table_rd_en	
Write_Enable	Output	niu_mb0_smx_table_wr_en	
Read_Enable	Output	niu_mb0_smx_store_rd_en	
Write_Enable	Output	niu_mb0_smx_store_wr_en	
Address	Output	niu_mb0_addr[5:0]	
Wdata	Output	niu_mb0_wdata[7:0]	
Run	Output	sniu_mb0_run	
Fail	Output	niu_tcu_mbist_fail_0	
Done	Output	niu_tcu_mbist_done_0	
Start	Input	tcu_niu_mbist_start_0	
Data_Out	Input	niu_mb0_smx_table_data_out[145:0]	
Data_Out	Input	niu_mb0_smx_store_data_out[145:0]	
scan_in	Input	mb0_scan_in	
scan_out	Output	mb0_scan_out	

TABLE 6-54 TXC NIU_MB1 Mbist Controller Ports

Description	Type	Signal Name	Changed
Read_Enable	Output	niu_mb1_xmit_store_rd_en	
Write_Enable	Output	niu_mb1_xmit_store_wr_en	
Read_Enable	Output	niu_mb1_xmit_realign_rd_en	
Write_Enable	Output	niu_mb1_xmit_realign_wr_en	
Address	Output	niu_mb1_addr[11:0]	
Wdata	Output	niu_mb1_wdata[7:0]	
Run	Output	niu_mb1_run	
Done	Output	niu_tcu_mbist_done_1	
Fail	Output	niu_tcu_mbist_fail_1	
Start	Input	tcu_niu_mbist_start_1	
Data_Out	Input	niu_mb1_xmit_store_data_out[151:0]	
Data_Out	Input	niu_mb1_xmit_realign_data_out[151:0]	

TABLE 6-54 TXC NIU_MB1 Mbist Controller Ports *(Continued)*

Description	Type	Signal Name	Changed
scan_in	Input	mb1_scan_in	
scan_out	Output	mb1_scan_out	
dmo_dout	Output	mb1_dmo_dout[39:0]	New

TABLE 6-55 TDMC NIU_MB2 Mbist Controller Ports

Description	Type	Signal Name	Changed
Read_Enable	Output	niu_mb2_rd_en	
Write_Enable	Output	niu_mb2_wr_en	
Address	Output	niu_mb2_addr[7:0]	
Wdata	Output	niu_mb2_wdata[7:0]	
Run	Output	niu_mb2_run	
Fail	Output	niu_tcu_mbist_fail_2	
Done	Output	niu_tcu_mbist_done_2	
Start	Input	tcu_niu_mbist_start_2	
Data_Out	Input	niu_mb2_tdmc_data_out[147:0]	
scan_in	Input	mb2_scan_in	
scan_out	Output	mb2_scan_out	
dmo data out	Output	mb2_dmo_dout[39:0]	New

TABLE 6-56 RDMC NIU_MB4 Mbist Controller Ports

Description	Type	Signal Name	Changed
Read_Enable	Output	niu_mb4_desc_rd_en	
Write_Enable	Output	niu_mb4_desc_wr_en	
Read_Enable	Output	niu_mb4_comp_rd_en	
Write_Enable	Output	niu_mb4_comp_wr_en	
Address	Output	niu_mb4_addr[7:0]	
Wdata	Output	niu_mb4_wdata [7:0]	
Run	Output	niu_mb4_run	
Fail	Output	niu_tcu_mbist_fail_4	
Done	Output	niu_tcu_mbist_done_4	

TABLE 6-56 RDMC NIU_MB4 Mbist Controller Ports *(Continued)*

Description	Type	Signal Name	Changed
Start	Input	tcu_niu_mbist_start_4	
Data_Out	Input	niu_rdmc_desc_data_out[147:0]	
Data_Out	Input	niu_rdmc_comp_data_out[147:0]	
scan_in	Input	mb4_scan_in	
scan_out	Output	mb4_scan_out	
dmo data out	Output	mb4_dmo_dout[39:0]	New

TABLE 6-57 RXC NIU_MB3 IPP Mbist Controller Ports

Description	Type	Signal Name	Changed
Read_Enable	Output	niu_mb3_rx_data_fifo_rd_en	
Write_Enable	Output	niu_mb3_rx_data_fifo_wr_en	
Read_Enable	Output	niu_mb3_prebuf_header_rd_en	
Write_Enable	Output	niu_mb3_prebuf_header_wr_en	
Address	Output	niu_mb3_addr[9:0]	
Wdata	Output	niu_mb3_wdata[7:0]	
Run	Output	niu_mb3_run	
Fail	Output	niu_tcu_mbist_fail_3	
Done	Output	niu_tcu_mbist_done_3	
Start	Input	tcu_niu_mbist_start_3	
Data_Out	Input	niu_mb3_rx_data_fifo_data_out[145:0]	
Data_Out	Input	niu_mb3_prebuf_header_data_out[145:0]	
scan_in	Input	mb3_scan_in	
scan_out	Output	mb3_scan_out	
dmo data out	Output	mb3_dmo_dout[39:0]	New

TABLE 6-58 RXC NIU_MB5 TCAM Mbist Controller Signals

Description	Type	Signal Name	Changed
Read_Enable	Output	niu_mb5_tcam_cntrl_rd_en	
Write_Enable	Output	niu_mb5_tcam_cntrl_wr_en	
Address	Output	rxc_mb5_addr[6:0]	

TABLE 6-58 RXC NIU_MB5 TCAM Mbist Controller Signals

Description	Type	Signal Name	Changed
Run	Output	rxc_mb5_run	
Fail	Output	niu_tcu_mbist_fail_5	
Done	Output	niu_tcu_mbist_done_5	
Start	Input	tcu_niu_mbist_start_5	
scan_in	Input	mb5_scan_in	
scan_out	Output	mb5_scan_out	
cam_haddr	Input	niu_mb5_cam_haddr[6:0]	
cam_compare	Output	niu_mb5_cam_compare	
cam_hit	Input	niu_mb5_cam_hit	
data_inp	Output	niu_mb5_data_inp[199:0]	
pio_sel	Output	niu_mb5_pio_sel	
msk_dat_out	Input	niu_mb5_msk_dat_out[199:0]	
cam_valid	Input	niu_mb5_cam_valid	
pio_rd_vld	Input	niu_mb5_rd_vld	

TABLE 6-59 RXC NIU_MB6 Mbist Controller Signals

Description	Type	Signal Name
Read_Enable	Output	niu_mb6_tcam_array_rd_en
Write_Enable	Output	niu_mb6_tcam_array_wr_en
Read_Enable	Output	niu_mb6_vlan_rd_en
Write_Enable	Output	niu_mb6_vlan_wr_en
Address	Output	niu_mb6_addr[11:0]
Wdata	Output	niu_mb6_wdata [7:0]
Run	Output	niu_mb6_run
Fail	Output	niu_tcu_mbist_fail_6
Done	Output	niu_tcu_mbist_done_6
Start	Input	tcu_niu_mbist_start_6
Data_Out	Input	niu_mb6_tcam_array_data_out[41:0]
Data_Out	Input	niu_mb6_vlan_data_out[8:0]

TABLE 6-59 RXC NIU_MB6 Mbist Controller Signals *(Continued)*

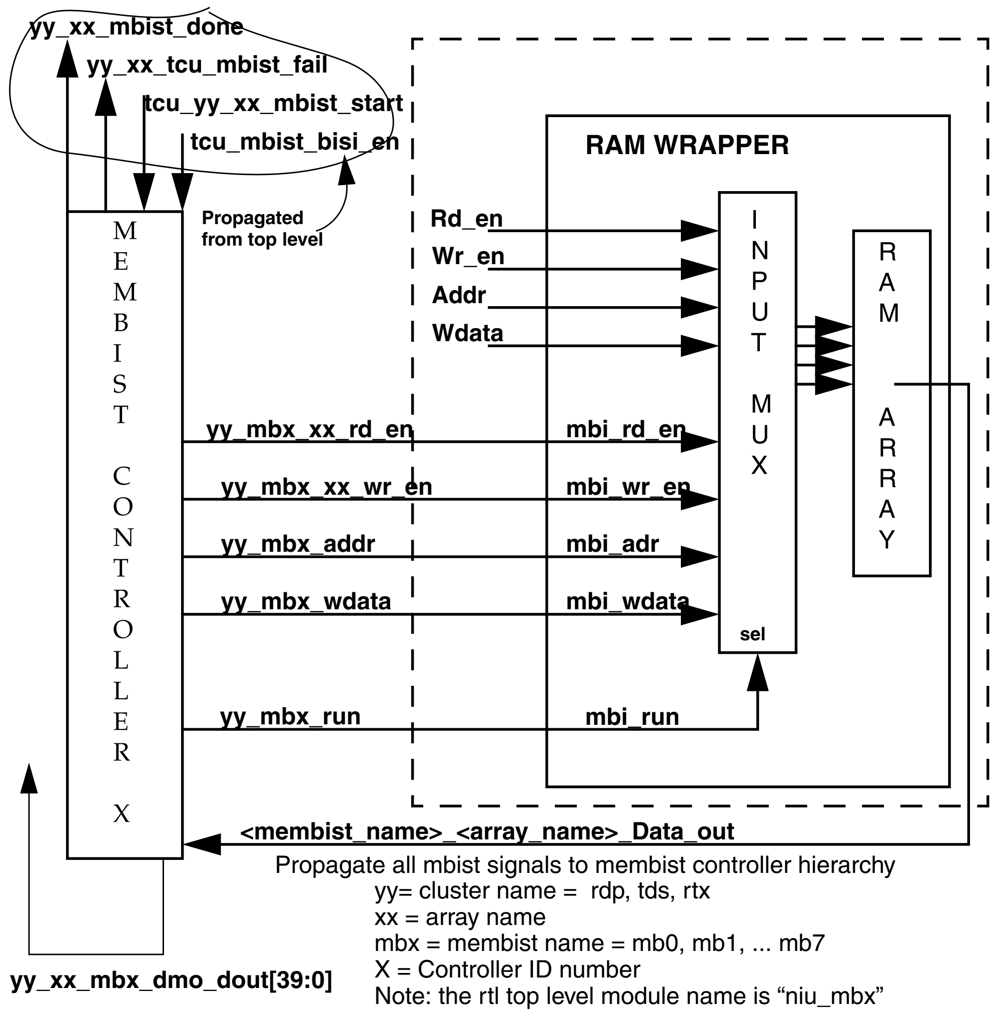
Description	Type	Signal Name	
scan_in	Input	mb6_scan_in	
scan_out	Output	mb6_scan_out	
dmo data out	Output	mb7_dmo_dout[39:0]	New

TABLE 6-60 RXC NIU_MB7 ZCP Mbist Controller Signals

Description	Type	Signal Name	
Read_Enable	Output	niu_mb7_cntrl_fifo_zcp_rd_en	
Write_Enable	Output	niu_mb7_cntrl_fifo_zcp_wr_en	
Address	Output	niu_mb7_addr[8:0]	
Wdata	Output	niu_mb7_wdata[7:0]	
Run	Output	niu_mb7_run	
Fail	Output	niu_tcu_mbist_fail_7	
Done	Output	niu_tcu_mbist_done_7	
Start	Input	tcu_niu_mbist_start_7	
Data_Out	Input	niu_mb7_cntrl_fifo_zcp_data_out[145:0]	
scan_in	Input	mb7_scan_in	
scan_out	Output	mb7_scan_out	
dmo data out	Output	mb7_dmo_dout[39:0]	New

6.28.16 RAM vs. Membist Controller Connectivity

FIGURE 6-89 Ram vs. Membist Controller Connectivity



Propagate all mbist signals to membist controller hierarchy
yy= cluster name = rdp, tds, rtx
xx = array name
mbx = membist name = mb0, mb1, ... mb7
X = Controller ID number
Note: the rtl top level module name is “niu_mbx”

6.29 SMX Microarchitecture

The SMX module is designed OpenSPARC T2 and is the bus bridge from the Meta Interface (which is NIU's internal protocol) to the System Interface Unit (SIU) bus interface.

The SMX module has the following functionality:

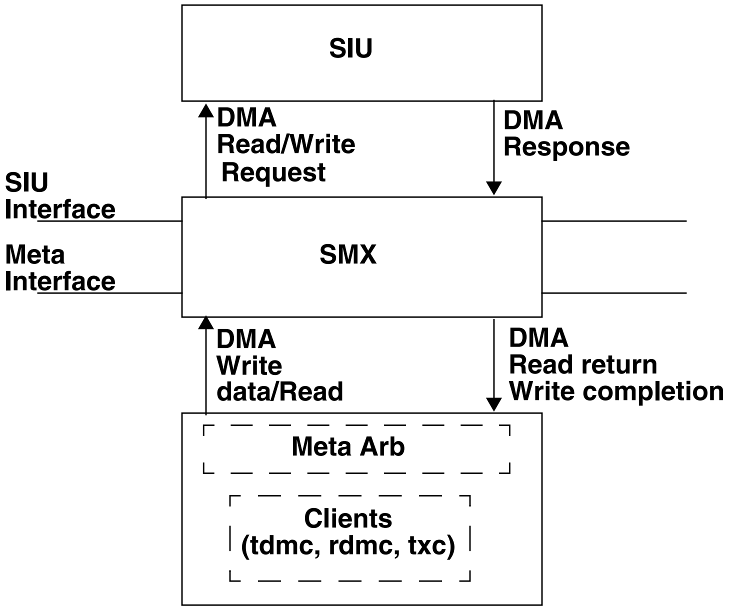
- The DMA read/write data is routed through this bridge Interface.
- Performs Flow Control
- Support the DMA function of the DMU module by fetching read and write descriptors and reading and writing data from host memory in 64 byte aligned blocks.
- Converts larger data read and write request payloads on the Meta interface to 64 byte aligned requests, with a fixed payload of 64 bytes (One cacheline).
- The maximum payload coming from the DMU block will be 8K+ bytes on the Meta. The SIU block supports a fixed 64 bytes data payload.

The SMX module comprises of the following sub-modules:

1. Meta Read/Write state m/c
2. SIU Read/Write state m/c, Credit based Flow control logic
3. Command/Data Buffering on the Write path and the Read return paths
4. Two levels of buffering to store two commands and corresponding 64byte data on the outbound and inbound path of the Translator.

6.29.1 Block Diagram

FIGURE 6-90 SMX Module Block Diagram



6.29.2 SMX Data Flow Diagram

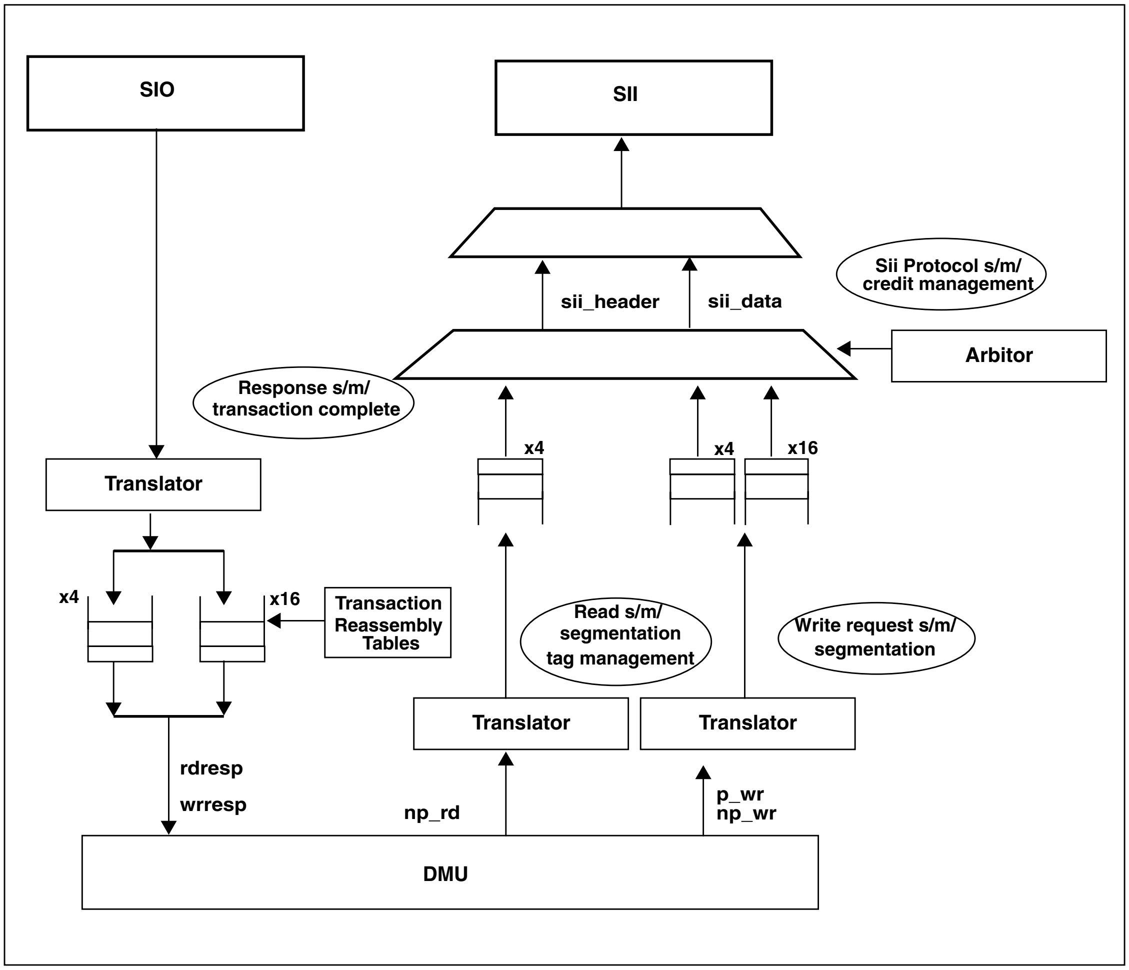

SIO
SII
Sii Protocol s/m/ credit management
sii_header
sii_data
Arbitor
Response s/m/ transaction complete
x4
x4
x16
Translator
x4
x16
Transaction Reassembly Tables
Read s/m/ segmentation tag management
Write request s/m/ segmentation
Translator
Translator
rdresp
wrresp
np_rd
p_wr
np_wr
DMU

FIGURE 6-91 SMX Data Flow

6.29.3 Description of Bus Interfaces to the SMX Module

6.29.3.1 SIU Interface

The SIU-NIU Interface is specified in the SIU Microarchitecture document, please refer to the SIU Volume 1, Chapter 6 for timing diagrams and Header format

TABLE 6-61 SII-SMX Interface

Signal Name	I/O	Size	From/To	Description
niu_sii_hdr_vld	O	1	SMX->SII	Asserted for one cycle (header cycle) to indicate that the header packet is being sent on the data pins(niu_sii_data.).
niu_sii_reqbypass	O	1	SMX->SII	Indicated that NIU is sending packet to SII's (bypass) Queue. Asserted in the header valid cycle.
niu_sii_datareq	O	1	SMX->SII	Indicates that header has a payload following the header valid cycle, this is also asserted in the header cycle.
niu_sii_oqdq	I	1	SII->SMX	Dequeue signal for the ordered queue.
niu_sii_bqdq	I	1	SII->SMX	Dequeue signal for the bypass queue.
niu_sii_data	O	128	SMX->SII	Contains the header in the header cycle (1st cycle) and the payload in the following data cycles. Data is in big endian format.
niu_sii_parity	O	8	SMX->SII	Contains the parity for each 16 bit of data. See RAS documentation.

TABLE 6-62 SIO-SMX Interface

Signal Name	I/O	Size	From/To	Description
sio_niu_hdr_vld	I	1	SIO->SMX	Asserted for one cycle (header cycle) to indicate that the header packet is being sent on the data pins(sio_niu_data.).
sio_niu_datareq	I	1	SIO->SMX	Indicates that header has a payload following the header valid cycle, this is also asserted in the header cycle.
sio_niu_data	I	128	SIO->SMX	Contains the header in the header cycle (1st cycle) and the payload in the following data cycles. Data is in big endian format.
sio_niu_parity	I	8	SIO->SMX	Contains the parity for each 16 bit of data. See SOC RAS doc
nu_sio_dq	O	1	SMX->SIO	Dequeue signal for inbound queue; data flow from sio to smx is flow control with 4 credits;

TABLE 6-63 Meta Protocol Signals

Signal Name	I/O	Size	From/To	Description
Write REQUEST				
dmc_meta0_req_cmd	I	8	DMU->SMX	Command Requests: Memory Read, Memory Write, Completion.
dmc_meta0_req_address	I	64	DMU->SMX	64 bit Memory Address
dmc_meta0_req_transID	I	6	DMU->SMX	Transaction Identification
dmc_meta0_req_length	I	14	DMU->SMX	Data Length (byte units)
dmc_meta0_req_port_num	I	2	DMU->SMX	Port number corresponding to the request (to be returned with read response)
dmc_meta0_req_dma_num	I	5	DMU->SMX	Dma number corresponding to the request (to be returned with read response)
dmc_meta0_req_client	I	8	DMU->SMX	Requesting Client (vector, one-hot encoded)
dmc_meta0_req	I	1	DMU->SMX	Send Queue Request
meta_dmc0_req_accept	O	1	SMX->DMU	Grant Send Queue Request
Write REQUEST Data and Data Control				
meta_dmc0_data_req	O	1	SMX->DMU	SMX Request for Burst Transfer.
dmc_meta0_data_valid	I	1	DMU->SMX	DMU sends data Ack with every cycle of valid data.
dmc_meta0_status	I	4	DMU->SMX	Packet Transfer Status: Complete, Abort.
dmc_meta0_data	I	128	DMU->SMX	Data.
dmc_meta0_req_byteenable	I	16	DMU->SMX	Contains the byteenables for each byte of data in the 16 byte data transfer. byteenable[N]==1 implies write data[8N + 7: 8N] is enabled (valid).
dmc_meta0_transfer_comple te	I	1	SMX->DMU	Transfer complete. No additional data for this transaction. Asserted coincidental with last data.
Write REQUEST Error Flag				
meta_dmc0_req_errors	O	1	SMX->DMU	Flag to report errors back to NIU. Flag is asynchronous with respect to write REQUEST events.
Read REQUEST				
dmc_meta1_req_cmd	I	8	DMU->SMX	Command Requests: Memory Read
dmc_meta1_req_address	I	64	DMU->SMX	64 bit Memory Address
dmc_meta1_req_transID	I	6	DMU->SMX	Transaction Identification
dmc_meta1_req_length	I	14	DMU->SMX	Data Length (byte units)

TABLE 6-63 Meta Protocol Signals *(Continued)*

Signal Name	I/O	Size	From/To	Description
dmc_meta1_req_port_num	I	2	DMU->SMX	Port number corresponding to the request (to be returned with read response)
dmc_meta1_req_dma_num	I	5	DMU->SMX	Dma number corresponding to the request (to be returned with read response)
dmc_meta1_req_client	I	8	DMU->SMX	Requesting Client (vector, one-hot encoded)
Read REQUEST Data and Data Control				
N/A				
Read REQUEST Error Flag				
meta_dmc1_req_errors	O	1	SMX->DMU	Flag to report errors back to NIU. Flag is asynchronous with respect to write REQUEST events.
RESPONSE Transaction Type and Transaction Control				
meta_dmc_resp_cmd	O	8	SMX->DMU	Command Requests: Completion.
meta_dmc_resp_cmd_ststus	O	4	SMX->DMU	Response command status: 4'hf - timeout error
meta_dmc_resp_address	O	64	SMX->DMU	64 bit Memory Address
meta_dmc_resp_transID	O	6	SMX->DMU	Transaction Identification
meta_dmc_resp_length	O	14	SMX->DMU	Data Length (byte units)
meta_dmc_resp_port_num	O	2	SMX->DMU	Port number corresponding to the request (to be returned with read response)
meta_dmc_resp_dma_num	O	5	SMX->DMU	Dma number corresponding to the request (to be returned with read response)
meta_dmc_resp_client	O	8	SMX->DMU	Requesting Client (vector, one-hot encoded)
meta_dmc_resp_ready	O	1	SMX->DMU	SMX to DMU command response ready
dmc_meta_resp_accept	I	8	DMU->SMX	Grant Receive Queue Request
RESPONSE Data and Data Control				
meta_dmc_data_valid	O	1	SMX->DMU	Data Ack validating every cycle of valid data.
meta_dmc_data_status	O	4	SMX->DMU	Packet Transfer Status: Complete, Abort.
meta_dmc_data	O	128	SMX->DMU	Packet Data.
meta_dmc_resp_byteenable	O	16	SMX->DMU	Contains the byteenables for each byte of data in the 16 byte data transfer. byteenable[N]==1 implies write data[8N + 7: 8N] is enabled (valid).

TABLE 6-63 Meta Protocol Signals *(Continued)*

Signal Name	I/O	Size	From/To	Description
meta_dmc_resp_complete	O	1	SMX->DMU	Single pulse on the last data cycle to indicate that the segmented response is complete.
meta_dmc_resp_transfer_cmpl	O	1	SMX->DMU	Single pulse on the last data cycle to indicate that all the read responses are returned from the host for the complete request i.e. the transaction is complete.
Write Completion Transaction Type				
meta_dmc_ack_cmd	O	8	SMX->DMU	Command Requests: Completion.
meta_dmc_ack_address	O	64	SMX->DMU	64 bit Memory Address
meta_dmc_ack_transID	O	6	SMX->DMU	Transaction Identification
meta_dmc_ack_length	O	14	SMX->DMU	Data Length (byte units)
meta_dmc_ack_port_num	O	2	SMX->DMU	Port number corresponding to the request (to be returned with read response)
meta_dmc_ack_dma_num	O	5	SMX->DMU	Dma number corresponding to the request (to be returned with read response)
meta_dmc_ack_client	O	8	SMX->DMU	Requesting Client (vector, one-hot encoded)
meta_dmc_ack _ready	O	1	SMX->DMU	SMX to DMU command response ready
dmc_meta_ack_accept	I	8	DMU->SMX	Grant Receive Queue Request
Write Completion without Data				
meta_dmc_ack_complete	O	1	SMX->DMU	Single pulse on the last data cycle to indicate that the segmented response is complete.
meta_dmc_ack_transfer_cmpl	O	1	SMX->DMU	Single pulse on the last data cycle to indicate that all the read responses are returned from the host for the complete request i.e. the transaction is complete

TABLE 6-64 Command Opcode (RECEIVE and SEND command)

Bits	Name	Usage
7:6	Reserved	
5	1=Posted, 0=Non-Posted	SIU: Supported. Reads are always Non-Posted Commands.

TABLE 6-64 Command Opcode (RECEIVE and SEND command) *(Continued)*

Bits	Name	Usage
4	1=Ordered,0=Un-Ordered	SIU: To send the request to the non-ordered queue (bypass queue), assert reqbypass signal on the SIU Header cycle.
3	Reserved	
2: 0	Memory Read=000,Memory Write = 001,Completion with Data = 101Completion without Data = 110	SIU: Supported

6.29.4 Meta-SIU Header Translation

The header encoding for a DMA Read/Write from NIU is shown in TABLE 6-65.

TABLE 6-65 SIU header/Meta Command Translation: DMA Read/Write Request

Header Cycle	Name	Meta Command bits	niu_sii_data[127:122]	Command
127=Response bit	126=Posted bit	125=Read bit	124=Write ByteMask Active	123=L2 bit
122=NCU bit		dmc_smx_cmd_req[3]	dmc_smx_cmd_req[5]	dmc_smx_cmd_req[0]
Set to 0	Set to 1	Set to 0	niu_sii_data[121:85]	Reserved
Must Be Zero	niu_sii_data[84:83]	AP[1:0]	Address Parity: AP[0]: parity for the even bits of PA, AP[1]: parity for the odd bits of PA. (i.e. PA[1], PA[3], PA[5].)	niu_sii_data[82]
TimeOutError	1=This packet had Timed Out	Set to Zero	niu_sii_data[81]	UnmappedAddressError
1=This packet's address mapped to an nonexistent, reserved, or erroneous address (Set to Zero)	niu_sii_data[80]	Uncorrectable Error	1=data payload has uncorrectable error	niu_sii_data[79:64]
ID[15:0]	Transaction ID;{dmc_meta_transID[5:0], segment}	niu_sii_data[63]	Reserved	Must be Zero

TABLE 6-65 SIU header/Meta Command Translation: DMA Read/Write Request

Header Cycle	Name	Meta Command bits	niu_sii_data[127:122]	Command
niu_sii_data[62]	CP	Command Parity: Parity for bits 127-122	niu_sii_data[61:56]	CtagEcc[5:0]
6-bit SEC-DED check bits for the 16-bit Ctag field.	niu_sii_data[55:40]	Reserved		niu_sii_data[39:0]
PA[39:0]	Segment start address	niu_sii_reqbypass	Control signal to indicate write should be directed to bypass queue	!dmc_smx_cmd_req[4]

6.29.5 Functional Description of Sub-blocks

6.29.5.1 Meta Request Sub-module

The meta request sub-module handles the protocol to interface with the SEND Queue signals on the Meta interface. The Meta Command is an 8-bit encode to specify the type of Memory Read/Write operation to be performed on the SIU bus. The Command is decoded and translated to the Sii HEADER and placed in the Sii FIFO. The Sii FIFO is two Commands deep.

On the data path the dmc_smx_length signal is latched in with the Command and data is read from the DMC SEND Queue in bursts of 4 data cycles, which is 64 bytes (16 Dwords) and placed in the Sii FIFO. As soon as this data fifo is non empty the header and data is sent out on the Sii bus. The Translated Sii HEADER is inserted after 4 data cycles, with the 64 byte aligned address for the data. The meta request state machine will stay in loop processing the Write request till all the data is converted to 64 byte chunks and read out of the DMC SEND Queue based on the dmc_smx_length latched in with the Command. The data length can be a non-multiple of 64 bytes (One cachline). At the trailing end of the data is not in a 64 bytes, 0s are padded in the data to make it a 64 bytes in length.

The byte enables are always set to 1s when in OpenSPARC T2 as zero-copy is not supported in T2. Byte enables are removed from the sii and sio bus as the data is always 64 byte aligned and it is OK to write memory with bad data at the start and end of the packet.

The TransactionID is 6 bit on the Meta and is 16 bits on the SIU interface. The SMX manages 64 outstanding requests and sends a new tag for each on the SII bus.

6.29.5.2 Sii Request Sub-module

The Sii_request state machine looks at the Sii FIFO signals, if the header FIFO is not empty and it is a read header it will initiate a read header cycle on the Sii bus. The Command buffering is 4 deep and a Command can be written by the Meta request/Translator block while one Command is being sent on the SIU bus. In the case of writes it waits for the SII data FIFO to be non-empty and then send the header followed by 4 cycles of data.

6.29.5.3 Sio Response Sub-module

Has two levels of buffering for the Header and data coming from SIU. This module is responsible for sending the dequeue signals for doing the Credit based flow control with the SIU block and has the state machine to support the Sio protocol.

6.29.5.4 Meta Response Sub-module

The Meta Response block handles the protocol timing for communicating with the RECEIVE QUEUE in the DMU and passed the Read/Write response data back to the DMU. It also computes the byte enable information from address and length and drives the byteenable lines to the Clients.

In addition it asserts a completion signal with the last response for each transaction ID. The Meta interface tracks the last response from the HOST for each Meta Read request (each request has a unique transaction ID). This will require the SMX to maintain a table of Transaction ID with a corresponding Outstanding request length information for each request.

6.29.6 Transaction Table

Each meta request which expects responses (with or without data) is kept track via transaction ID. SMX supports 64 transaction ID. For each meta request launched, the corresponding packet info is stored in the transaction table. The transaction ID is later extracted from the tagID field of sio packet header (responses). This is then used to index the transaction table.