PCIO

Peripheral Component Interconnect
Input Output Controller

THE NETWORK IS THE COMPUTER™

Sun Microelectronics
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-7837-01
March 1997

Please
Recycle

Contents

Figures

Tables

CHAPTER 1

Introduction

The PCIO is a high integration, high performance single chip IO subsystem, interfacing to the PCI Local Bus. Off a single PCI bus load, it integrates high speed Ethernet and EBus2, a generic, slave-DMA bus to which are connected off-the-shelf peripherals implementing the rest of the Sun core IO system.

1.1 Objectives

The objectives for PCIO are a reflection of the systems it is targeted to. These are primarily cost-sensitive, high performance, single and multi processor desktops. The objectives include:

- Low cost
- Time-to-volume/time-to-market
- Complete IO subsystem: Ethernet, keyboard, mouse, serial ports, parallel port, stereo audio, floppy disk, boot PROM and time-of-day/non-volatile RAM
- High performance: high bandwidth, low interrupt overhead, tolerant of relatively high interrupt and bus latencies
- Modularity, at the architectural, design and test levels
- Full compliance with Sun and industry standards
- Suitability for licensing and third party marketing

1.2 Key Device Features

The following functions are built-in:

- PCI Local Bus master/slave interface, compliant with *PCI Local Bus Specification, Revision 2.0*[1]
- 10baseT (802.3) and 100baseT (802.30) Ethernet, using derivative of MAC, with fully buffered transmit and receive channels; media-independent interface (MII)
- Expansion bus interface (EBus2), supporting eight external devices and four buffered slave-DMA channels
- Oscillator for 40 MHz SCSI clock, and free running 10 MHz real-time clock
- IEEE 1149.1 JTAG compliant test architecture

The following functions are implemented with off-the-shelf devices, interfacing directly to the EBus2 interface:

- National Semiconductors™ PC87303VLJ Super IO, integrating 82077-compatible floppy controller with DMA, parallel port, P1284-compliant, with ECP and EPP with DMA and two 16C550 serial controllers with 16-byte FIFOs, for keyboard and mouse
- Two high performance sync/async serial ports, using Siemens' SAB82532. 460.8 KBaud async, 384 Kbaud sync.
- Sun-compatible NVRAM, MK48T59, with alarm clock interrupt for power management
- EPROM or flash EPROM, 8-bit wide, up to 16 Mbyte, for boot or Fcode
- CS4231 Audio CODEC
- Access to USC and DSC EBus control port
- Auxiliary IO ports, for power supply control, temperature sensor, frequency calibration and other miscellaneous functions

1.3 Intended Applications

PCIO is being designed as part of the Ultra AX program. Ultra AX is a single processor, UltraSPARC based desktop using the PCI bus as its IO bus.

In addition, PCIO can be used on any system with a standard PCI bus. In 'motherboard' mode it makes the PROM available for boot code; in 'add-in' mode it can be used in card adapters or as a secondary motherboard design.

1. Although designed to the PCI 2.0 specification, PCIO is compatible with the 2.1 specification as well. However, certain 2.1-recommended features, such as pseudo-split reads and a tighter interpretation of the initial latency rule are not implemented

CHAPTER 2

Functional Overview

This chapter identifies and provides a brief description of each of the major component blocks in PCIO.

2.1 Major Component Blocks

FIGURE 2-1 contains the block diagram for PCIO, showing the major component blocks. PCIO is built around an internal bus, the Channel Engine Interface, which provides the key to its modularity. Above the Channel Engine Interface, the Bus Adapter connects to the PCI bus. The two identical ports on the Channel Engine Interface are used for each of PCIO's functional units: Ethernet and Ebus2. Each of these has its own set of control and status registers, data buffers and the core logic function.

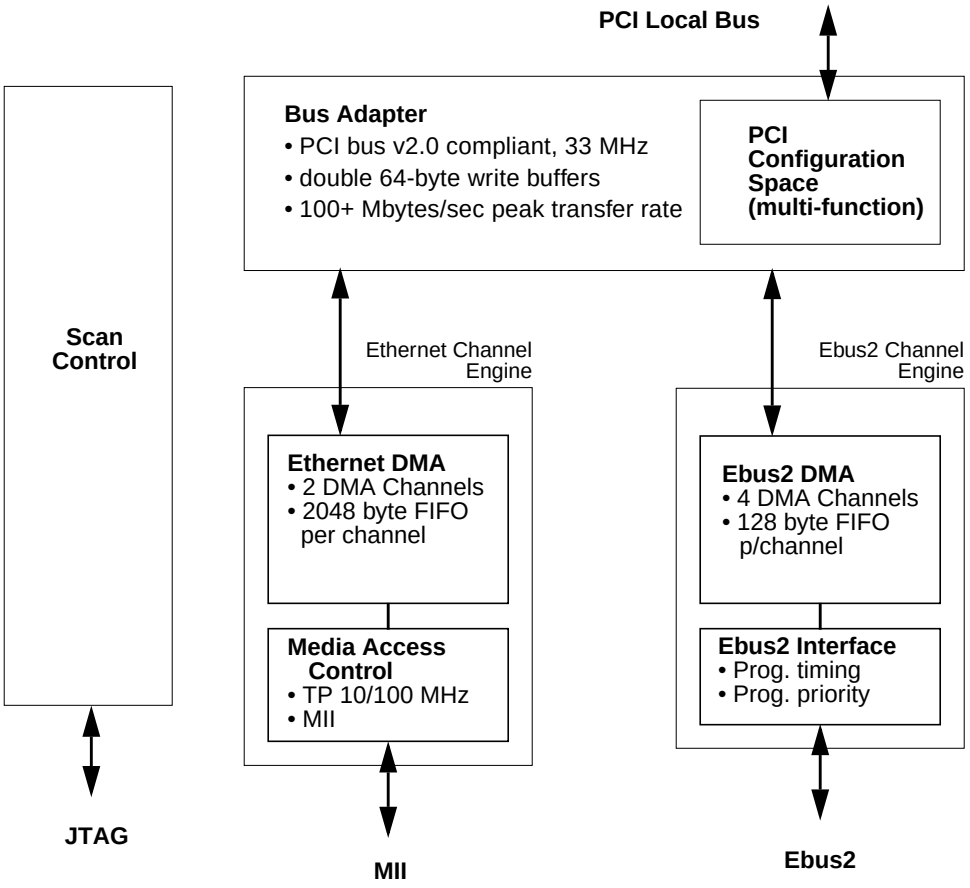

FIGURE 2-1 PCIO Block Diagram

2.1.1 Bus Adapter

The Bus Adapter provides a bus-independent layer between the channel engines and the PCI bus. The PCI bus interface is 32-bit and 33 MHz, fully compliant with the *PCI Local Bus Specification, Revision 2.0*. As a master, it is capable of 64-byte (8 word) bursts. DMA writes are buffered in the Bus Adapter to support back to back transactions.

The Bus Adapter also contains the PCI bus Configuration Space. PCIO presents itself to PCI as a multi-function device: Ebus2 (a bridge) and Ethernet. Each function has its own area in the configuration space.

2.1.2 Channel Engine Interface

The Bus Adapter contains two identical Channel Engine Interface ports, one for each channel engine. The Channel Engine Interface is a bus independent interface, resulting in a high level of modularity at the design and test level.

2.1.3 EBus2 Channel Engine

The EBus2 Channel Engine interfaces standard off-the-shelf devices to PCIO. Up to eight single or multi-function Intel-style 8-bit devices can be accommodated with a minimum of glue logic. Four internal DMA engines can be attached to any of these devices, buffering data streams in 128-byte FIFOs for each channel.

The standard set of IO devices is: PC87303VLJ Super IO (integrates 82077 floppy controller, dual 16C550 serial controllers for keyboard and mouse and ECP/EPP P1284 parallel port), SAB82532 serial communications controller, CS4231 audio CODEC, MK48T59 NVRAM with alarm clock, boot PROM and USC/DSC control port.

The EBus2 Channel Engine provides access to several general purpose IO lines (*a.k.a.* AUXIO), used to control miscellaneous system functions.

2.1.4 Ethernet Channel Engine

The Ethernet Channel Engine provides a buffered full duplex DMA engine and a Media Access Control function based on MAC. The descriptor-based DMA engine contains independent transmit and receive channels, each with 2048 bytes of on-chip buffering. The MAC provides a 10 or 100 Mbps CSMA/CD protocol based network interface conforming to IEEE 802.3, proposed IEEE 802.30 and Ethernet specifications.

2.1.5 Scan Control Block

The Scan Control contains a tap controller.

CHAPTER 3

Programmer's Model

3.1 Address Map

3.1.1 PCI Bus Configuration Space

PCIO's PCI Configuration space complies with the PCI Bus Specification v2.0. Unless otherwise indicated, all the configuration space is accessible as bytes, half-word and word, and is read / write.

PCIO contains two functions within a single device: EBus2 and Ethernet. These are implemented as functions 0 and 1 respectively. PCIO responds to configuration cycles for functions 2 to 7 indicating they are not implemented (i.e. read zeroes from all locations.)

Detailed information on the PCI configuration space can be found in Chapter 6 of the PCI bus specification.

Note – All multi-byte configuration fields are in LITTLE ENDIAN format

Table 3-1 PCIO PCI Configuration Space

Offset	Size	R/W	Name
EBus2			
0x000 - 0x001	16-bit	RO	Vendor ID = 0x108E
0x002 - 0x003	16-bit	RO	Device ID = 0x1000

Table 3-1 PCIO PCI Configuration Space *(Continued)*

Offset	Size	R/W	Name
0x004 - 0x005	16-bit	RW	Command Register
0x006 - 0x007	16-bit	RW	Status Register
0x008	8-bit	RO	Revision ID = 0x01
0x009 - 0x00B	24-bit	RO	Class Code 0x00B Base Class = 0x06 Bridge Device 0x00A Sub Class = 0x80 Other Bridge 0x009 Prog i/f = 0x00
0x00C	8-bit	R/W	Cache Line Size, in units of 32-bit words; resets to 0x00
0x00D	8-bit	R/W	Latency Timer
0x00E	8-bit	RO	Header Type = 0x80, identifies multi-function device, standard header type
0x00E	8-bit	RO	BIST = 0x00, not capable
0x010 - 0x013	32-bit	RW	Base Address Register, Boot ROM, 16 Mbyte space: bits [23:0] read as zero; reset value depends on setting of boot[1:0] pins: 00 reset to 0x3000.0000 01 reset to 0x7000.0000 10 reset to 0xB000.0000 11 reset to 0xF000.0000
0x014 - 0x017	32-bit	RW	Base Address Register, EBus2 Channel Engine, 8 Mbyte space; bits [22:0] read as zero; resets to 0xF100.0000
0x018 - 0x02F			Reserved, read as zero
0x030 - 0x033	32-bit	RW	Expansion ROM Base Address, 16 Mbyte space: bits [23:0] read as zero; resets to 0x0000.0000
0x034 - 0x03B			Reserved, read as zero
0x03C	8-bit	RW	Interrupt Line
0x03D	8-bit	RO	Interrupt Pin = 0x01, use INTA# if ADD-IN mode = 0x00, if MOTHERBOARD mode
0x03E	8-bit	RO	Min_Gnt = 0x0A, in units of 1/4 of a microsecond = 2.5 µS
0x03F	8-bit	RO	Max_Lat = 0x19, in units of 1/4 of a microsecond = 6.25 µS
0x040	32-bit	RW	Diagnostics Register

Table 3-1 PCIO PCI Configuration Space *(Continued)*

Offset	Size	R/W	Name
0x044-0x0FF			Reserved, read as zero
Ethernet			
0x100 - 0x101	16-bit	RO	Vendor ID = 0x108E
0x102 - 0x103	16-bit	RO	Device ID = 0x1001
0x104 - 0x105	16-bit	RW	Command Register
0x106 - 0x107	16-bit	RW	Status Register
0x108	8-bit	RO	Revision ID = 0x01
0x109 - 0x10B	24-bit	RO	Class Code 0x00B Base Class = 0x02 Network Device 0x00A Sub Class = 0x00 Ethernet 0x009 Prog i/f = 0x00
0x10C	8-bit	R/W	Cache Line Size, in units of 32-bit words; resets to 0x00
0x10D	8-bit	R/W	Latency Timer
0x10E	8-bit	RO	Header Type = 0x80, identifies multi-function device, standard header type
0x10E	8-bit	RO	BIST = 0x00, not capable
0x110 - 0x113	32-bit	RW	Base Address Register, Ethernet Channel Engine, 32 Kbyte space; bits [14:0] read as zero; resets to 0x0000.0000
0x114 - 0x117			Reserved, read as zero
0x118 - 0x12F			Reserved, read as zero
0x130 - 0x133	32-bit	RW	Expansion ROM Base Address, 16 Mbyte space: bits [23:0] read as zero; resets to 0x0000.0000
0x134 - 0x13B			Reserved, read as zero
0x13C	8-bit	RW	Interrupt Line
0x13D	8-bit	RO	Interrupt Pin = 0x02, use INTB# if ADD-IN mode = 0x00, if MOTHERBOARD mode
0x13E	8-bit	RO	Min_Gnt = 0x0A, in units of 1/4 of a microsecond = 2.5 µS
0x13F	8-bit	RO	Max_Lat = 0x05, in units of 1/4 of a microsecond = 1.25 µS

3.1.1.1 Command Register

The command register provides coarse control over a function's ability to generate and respond to PCI cycles. When a 0 is written to this register, the function is logically disconnected from the PCI bus for all accesses except configuration accesses. Each function in PCIO has its own command register.

Note – Please refer to §6.2.2 of the PCI bus specification for more information on the command register

---

Table 3-2 Command Register Bits

Bit	Usage
0	IO Space — Not implemented, read back as zero
1	Memory Space — Controls a function's response to memory space accesses: when set, allows the function to respond to memory space accesses. Reset to zero for Ethernet and for EBus2 in ADD-IN mode; reset to one for EBus2 in MOTHERBOARD mode
2	Bus Master — Controls a function's ability to act as a master on the PCI bus: when set, allows the device to behave as a bus master. Reset to zero
3	Special Cycles — Not implemented, read back as zero
4	Memory Write and Invalidate Enable — Controls whether a master can generate the Memory Write and Invalidate command (when set.) Reset to zero.
5	VGA Palette Snoop — Not implemented, read back as zero
6	Parity Error Response — This bit controls the function's response to parity errors.
7	Wait Cycle Control — Not implemented, read back as zero
8	SERR# Enable — This bit is an enable for the SERR# driver: when set, the SERR# pin driver is enabled
9	Fast Back-to-Back Enable — Not implemented, read back as zero
15 - 10	Reserved, read back as zero

3.1.1.2 Status Register

The status register is used to record information for PCI bus related events. Reads to this register behave normally; during writes, bits can only be reset, but not set. A bit is reset whenever the register is written, and the data in the corresponding bit location is a 1. Each function in PCIO has its own status register.

Note – Please refer to §6.2.3 of the PCI bus specification for more information on the status register

Table 3-3 Status Register Bits

Bit	Usage
6 - 0	Reserved, read back as zero
7	Fast Back-to-Back Capable — Read-only, set to 1, indicates PCIO is capable of accepting fast back-to-back transactions
8	Data Parity Detected — Set when three conditions are met: 1) PERR# was asserted or observed asserted, 2) function was the bus master for the transaction in which the error occurred, 3) Parity Error Response bit in command register is set
10 - 9	DEVSEL Timing — Read-only, set to 01 (medium)
11	Signaled Target Abort — When set, indicates the function terminated a transaction with a target-abort; this bit is implemented only in the EBus2 configuration space, it is read back as zero in Ethernet and SCSI
12	Received Target Abort — When set, indicates the function had a transaction terminated by a target-abort
13	Received Master Abort — When set, indicates the function had a transaction terminated by a master-abort
14	Signaled System Error — Set when any of the functions asserts SERR#; this bit is implemented only in the EBus2 configuration space, it is read back as zero in Ethernet
15	Detected Parity Error — Set when a function detects a parity error and the Parity Error Response in the command register is set

3.1.1.3 Expansion ROM

Although both function units have Expansion ROM Base Address registers, the two spaces map to the same physical ROM device, which is the same device used as boot ROM. Offset 0x00 from the either Expansion ROM space or Boot ROM space will read the first byte of the EPROM.

3.1.1.4 Diagnostics Register

The diagnostics register in the EBus2 configuration space provides a means to force a variety of error and exception conditions into the system. Although this register is read-write, most of the bits are self-clearing: they will reset themselves after generating the desired condition.

In order to write to the diagnostics register, the enable_diag_reg bit (bit 31) must be written first. Writes with this bit reset are ignored.

All bits in this register reset to zero.

Table 3-4 Diagnostics Register Bits

Bit	Name	Usage
0	bad_addr_parity_master	Computes wrong parity during address phases, when PCIO is bus master; will cause SERR# to be asserted by other devices in the bus (note: PCIO will not assert SERR# under this condition)
1	bad_addr_parity_slave	Computes wrong parity during address phases, when PCIO is not the bus master; will cause SERR# to be asserted by PCIO
2	bad_data_parity_master	Computes wrong parity during data phases, when PCIO is doing DMA writes; will cause PERR# to be asserted by the agent receiving data
3	bad_data_parity_slave	Computes wrong parity during data phases, when PCIO is data receiver (PIO write and DMA read); will cause PCIO to assert PERR#
4	Reserved	[Write Buffer Enable] - not implemented
5		Reserved, read back as zero
6	gen_disconnect	Causes PCIO to disconnect during a slave transaction, without returning any data
7	gen_target_abort	Causes PCIO to generate a target-abort during a slave transaction
8	use_byte_holes	Causes PCIO to use the 4-bit mask in bits [12:9] as the byte enables in the next transaction when PCIO is a master
12 - 9	byte_holes[3:0]	Byte enable mask; used with use_byte_holes bit to generate byte holes in DMA streams

Table 3-4 Diagnostics Register Bits *(Continued)*

Bit	Name	Usage
13	arb_write	Arbitrate PIO Writes, when set will force PIO writes to arbitrate for the CEI before proceeding; normally, arbitration is not needed, but setting this bit is required for running the diagnostic loopback from one DMA engine to another in EBus; reset to zero
14	force_swap	This bit, when set, will cause any descriptor transfer in the CEI to undergo byte swapping, rather than passing straight through. Reset to zero.
30 - 15		Reserved, read back as zero
31	enable_diag_reg	enable_diag_reg:

3.1.2 EBus2 Channel Engine

The address map for registers in the EBus2 channel engine can be found in Table 7-1.

3.1.3 Ethernet Channel Engine

Addresses on the Ethernet Channel Engine are offset from the Base Address Register 0 in the Ethernet section of the configuration space (Configuration Space address 0x110.) The address map can be found in Table 6-23.

CHAPTER 4

Bus Adapter

4.1 Introduction

The Bus Adapter provides the layer between the bus-independent Channel Engine Interface and the PCI Local Bus. Its main features are as follows:

- Single time domain operates at the PCI Bus frequency
- PCI Local Bus Revision 2.0 compatibility, 32-bit only
- Full master and slave capabilities
- 64-byte bursts as initiator
- Multi-function configuration space, with independent address decoders for each function
- Dual-buffered DMA WRITE path
- Two Channel Engine Interface ports, 32/64-bit wide
- Interrupt router for PCI Bus add-in card or motherboard modes

4.2 Address Map

There are no addressable registers in the Bus Adapter accessible during normal device operation. The PCI Bus Configuration Space is only available during PCI configuration cycles and contains registers which set up PCIO's basic operating functions. The address map for the Configuration Space is in the "Address Map" section of Chapter 3, "Programmer's Model."

4.3 Bus Adapter Blocks

4.3.1 Block Diagram

FIGURE 4-1 contains the block diagram for the Bus Adapter. The Input Datapath contains pipeline registers and a 32-bit parity checker. The Output Datapath contains dual DMA write buffers, address registers for pending DMA transactions, and a 32-bit parity generator. The Configuration Space contains the registers defined by the PCI Bus Specifications, and address decoders. The Interrupt Router directs the different channel engine interrupts to independent interrupt pins or to PCI INTA, B, C and D. Finally, the Control Logic is composed of a PCI Bus Control, dealing with the PCI Bus and one end of both input and output datapaths, and the CEI Control, dealing with the Channel Engine Interface and the other end of the input and output datapaths.

4.3.2 Input Datapath

A more detailed view of the Input Datapath is provided in FIGURE 4-2. The input datapath contains input pipeline registers on the PCI Bus signals, to minimize timing constraints. The pipelined data and command/byte enable bits are then checked for parity errors.

Address is sent to the Configuration Space where it is decoded. When any of the channel engines is decoded as the target for a write command, the pre-decoded address and data is stored in the write buffers, to accelerating PIO access. DMA read data received is sent to the CEI directly.

The input datapath also performs the conversion from the little endian PCI Bus to the big endian format of the channel engines. This conversion is “intelligent”, based on the data object size (byte, half word, word or double word) and whether the data object is part of a DMA data stream or is PIO or descriptor data. A pseudo-code description of this conversion is shown in TABLE 4-1.

Finally, the input datapath interfaces to a 64-bit wide Channel Engine Interface, with DMA transactions on the CEI being either 32- or 64-bits wide. The input datapath can assemble successive 32-bit words on the PCI Bus into 64-bit words on the CEI.

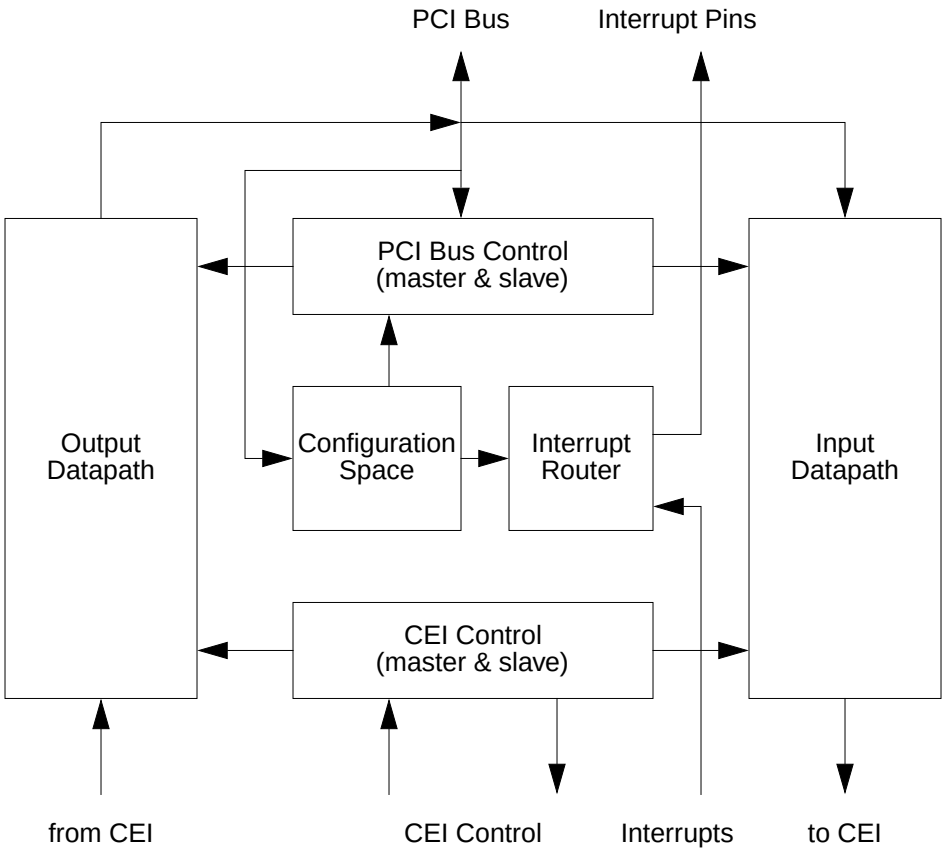

FIGURE 4-1 Bus Adapter Block Diagram

4.3.3 Output Datapath

A more detailed view of the Output Datapath is provided in FIGURE 4-3. All PCI Bus outputs are registered, to minimize timing constraints. Parity for data and command/byte enables is generated and output with one PCI clock delay of its respective data.

Data coming from the CEI is converted from big endian to little endian format, as described in the input datapath and shown in TABLE 4-1. Addresses are not affected by the conversion.

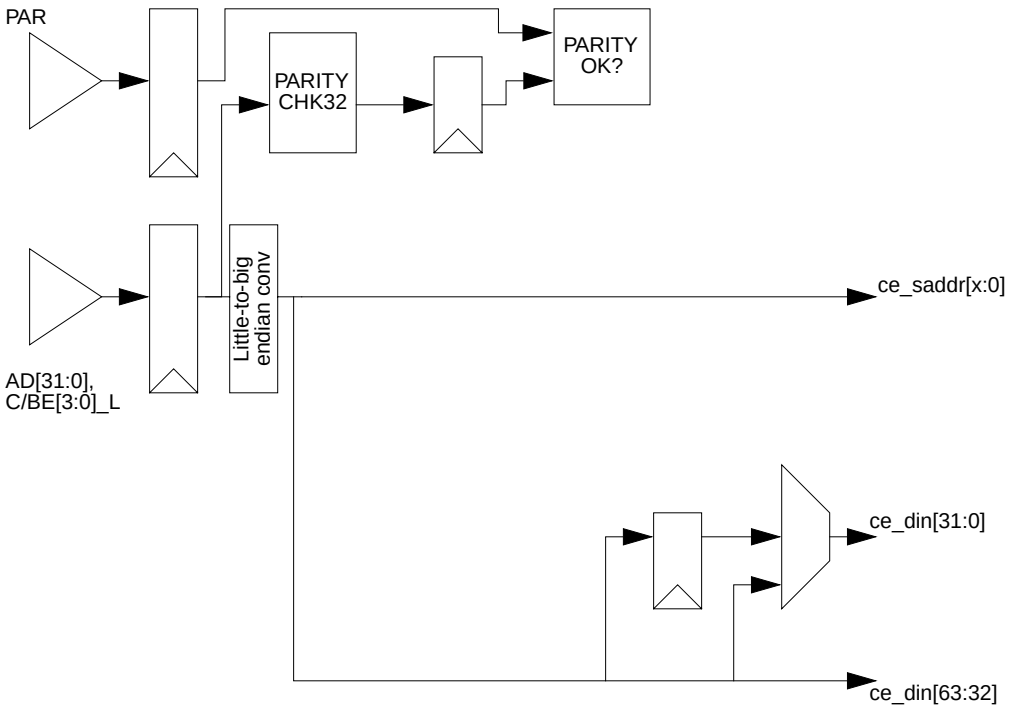

FIGURE 4-2 Bus Adapter Input Datapath

Data coming in from the CEI is first multiplexed, depending on which channel engine has ownership of the CEI. DMA address pointers can be stored in any of two write address registers, or a single read address registers. Up to three DMA transactions can be pending or in process: a DMA read, and two DMA writes.

DMA data from the CEI comes in either 32- or 64-bit formats, and is assembled into a DMA write buffers. Up to two 64-byte DMA write bursts can be pending. The 64-bit data out of the buffer is multiplexed into the 32-bit PCI Bus width. A 32-bit wide slave data path is also provided for PIO read operations.

4.3.4 Configuration Space

The PCI Bus Configuration Space is used by power-on software (i.e. OpenBoot) to probe and initialize system resources. The configuration space is a collection of read-only and read-write memory locations, all of which can be read by the power-on

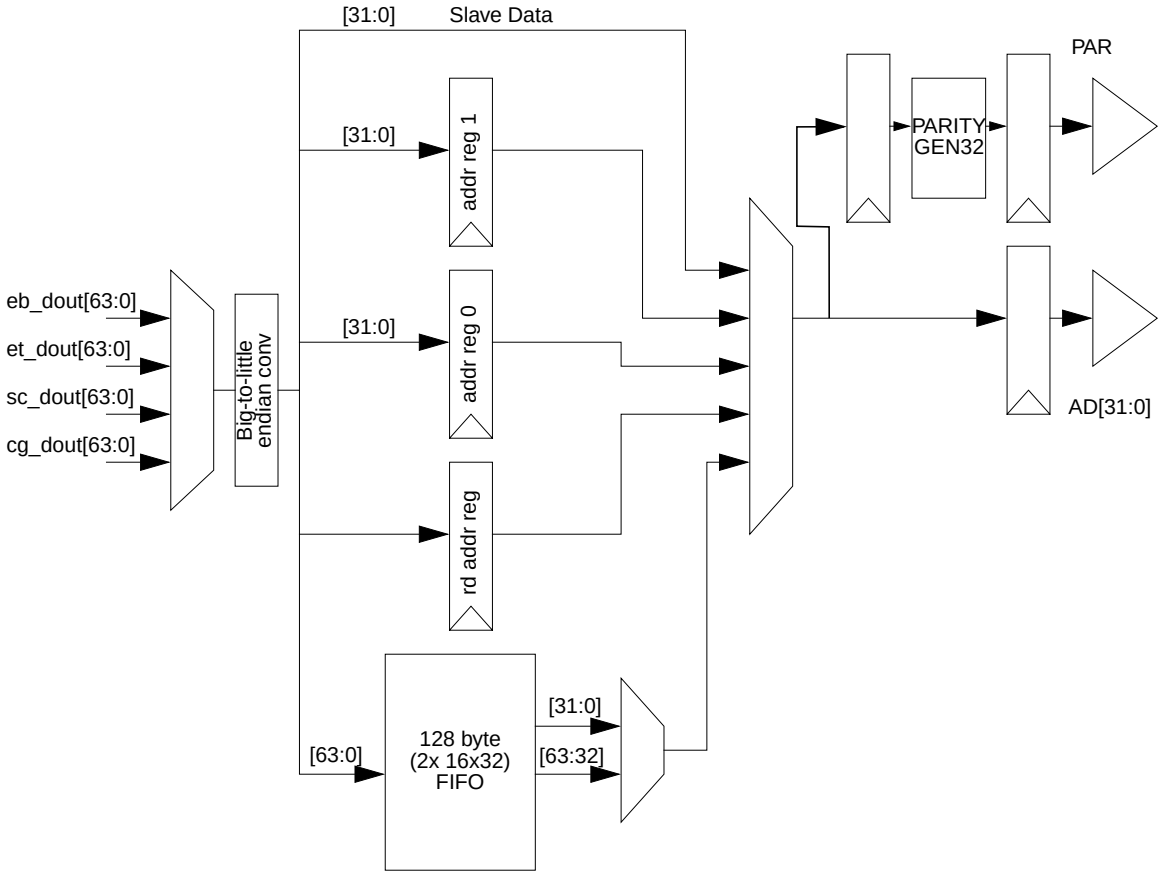

FIGURE 4-3 Bus Adapter Output Datapath

software using a configuration cycle. Some fields —Command, Latency Timer, Cache Line Size— are read-write fields with values visible to the PCI Bus Control logic at all times. The Status register has several inputs which are used to set and reset specific bits. Writing the Status register is by clearing bits only. More detailed information is available in the “PCI Bus Configuration Space” section of Chapter 3, “Programmer’s Model.” A diagram of the configuration space is shown in FIGURE 4-4.

The Address Decoders compare the address on the PCI Bus with the Base Address Registers (one or two) and Expansion ROM Base Address Register, when enabled by the “Memory Space” bit in the Command Register. The Expansion ROM Base Address Register has an additional Decode Enable bit within it.

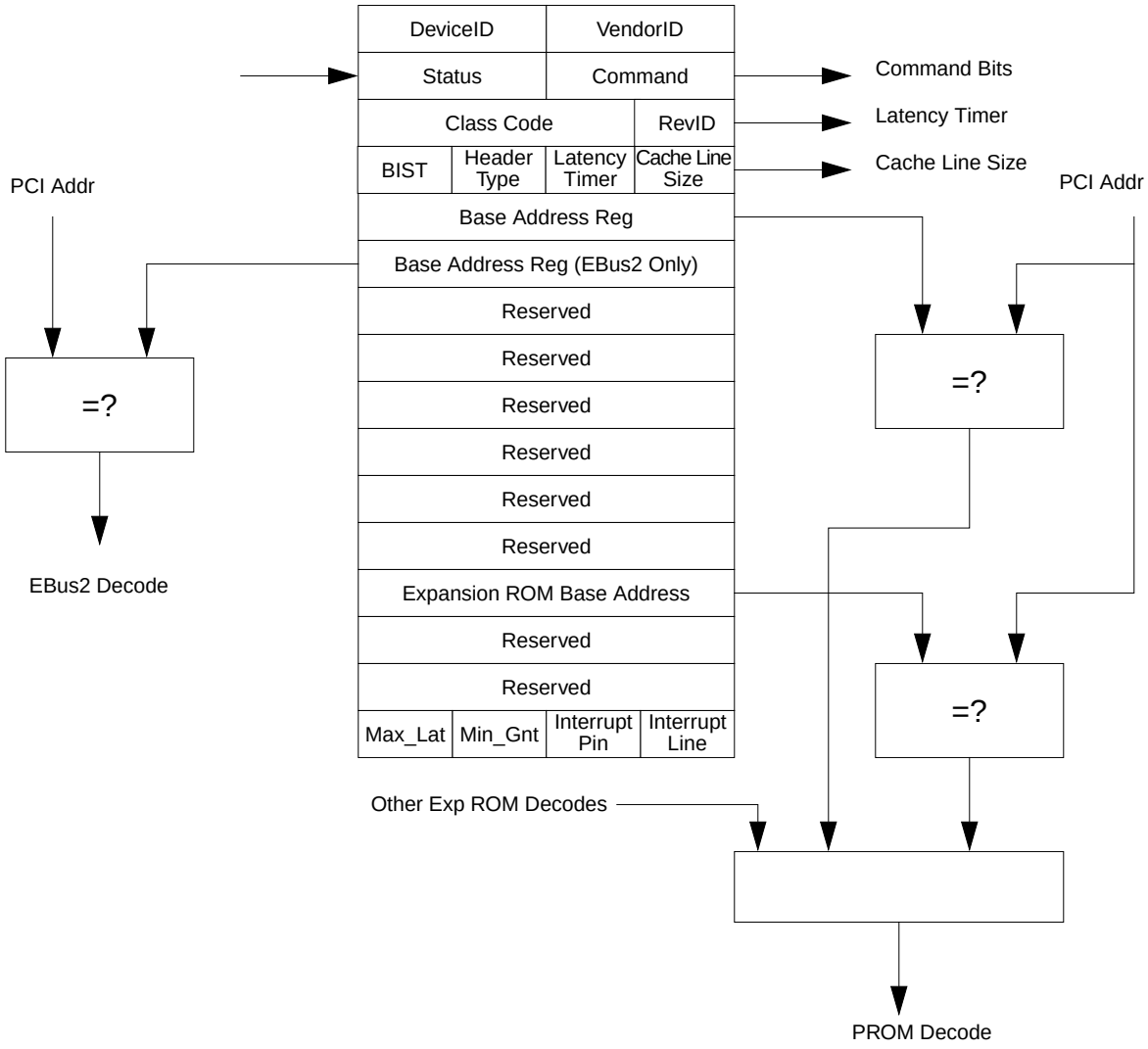

FIGURE 4-4 Configuration Space and Address Decoders

If PCIO is in ‘motherboard’ mode, the EBus2 address decoders are enabled on power-on, for access to boot PROM. The Base Address Registers for the EBus2 resets to a value depending on the BOOT pins for PROM and 0xF100_0000 for the other EBus2 devices. These address can be changed by the power-on software, to relocate either PROM or EBus2. Also, note that in ‘motherboard’ mode, the Memory Space bit in the Command Register for EBus2 has no effect.

TABLE 4-1 "Intelligent" Little-to-Big Endian Conversion

```
if PIO_access then
        case access_size of
                byte:{
                        pci_bits(31:24) goto channel_bits(7:0)
                        pci_bits(23:16) goto channel_bits(15:8)
                        pci_bits(15:8) goto channel_bits(23:16)
                        pci_bits(7:0) goto channel_bits(31:24)
                        }
                half_word: {
                        pci_bits(31:16) goto bits(15:0)
                        pci_bits(15:0) goto bits(31:16)
                        }
                word: pci_bits(31:0 goto channel_bits(31:0)
        endcase
else if DMA_access then
        if data_stream then {
                pci_bits(31:24) goto channel_bits(7:0)
                pci_bits(23:16) goto channel_bits(15:8)
                pci_bits(15:8) goto channel_bits(23:16)
                pci_bits(7:0) goto channel_bits(31:24)
                }
        else if descriptor then
                case access_size of
                        byte:{          /* not necessary if PCIO does not use */
                                pci_bits(31:24) goto channel_bits(7:0)
                                pci_bits(23:16) goto channel_bits(15:8)
                                pci_bits(15:8) goto channel_bits(23:16)
                                pci_bits(7:0) goto channel_bits(31:24)
                                }
                        half_word: {/* not necessary if PCIO does not use */
                                pci_bits(31:16) goto bits(15:0)
                                pci_bits(15:0) goto bits(31:16)
                                }
                        word: pci_bits(31:0 goto channel_bits(31:0)
                endcase
```

4.3.5 Interrupt Router

The Interrupt Router directs the channel engines' interrupts to the appropriate device pins. In 'add-in' mode, PCI devices must use the INTA, B, C and D provided in the PCI Bus Specification. In PCIO, EBus2 interrupts (only those associated with a DMA channel) are assigned to INTA# and Ethernet interrupts are assigned to INTB#. In 'motherboard' mode, PCIO has separate interrupt lines for each internal device.

INTA# becomes enet_irq_l, INTB# is unused, INTC# becomes pport_irq_l and INTD# becomes fpy_irq_l; interrupts from audio capture are routed to audio_cap_irq_l while interrupts from audio playback are routed to audio_pb_irq_l.

In ‘motherboard’ mode, interrupts from external EBus2 devices not associated with DMA channels (*e.g.* keyboard and mouse) are connected directly to the system interrupt controller (RIC chip in Ultra AX). In ‘add-in’ mode, these interrupts need to be combined with INTA#, since all EBus2-related interrupts should be on a single interrupt pin.

4.3.6 Control Logic

There are two main blocks of control logic, the PCI Bus Control, with a slave and master portion, and the CEI Control.

4.3.6.1 PCI Bus Control

The slave portion of the PCI Bus Control monitors the PCI Bus for configuration or memory commands. Based on the address decoding in the configuration space, it will claim transactions for which PCIO is the target, asserting DEVSEL#. For PIO reads, it will direct the CEI Control to arbitrate for the CEI and perform a slave read transaction. When the CEI Control indicates data is ready, the PCI Bus Control will complete the PCI transactions.

The master portion of the PCI Bus Control initiates transactions on the PCI on behalf of any of the channel engines. For DMA writes, after the CEI Control has accepted data into the DMA write buffer, it will direct the PCI Bus Control to start a PCI transaction. After the transaction is completed, it will issue a transaction acknowledge back into the CEI for the channel engine to synchronize its data flow. For DMA reads, after the CEI Controls grants the CEI to a master requesting a read, it will direct the PCI Bus Control to initiate a PCI read transaction. As data is returned from the PCI target, it is passed along into the CEI and the waiting master.

4.3.6.2 CEI Control

The CEI Control performs arbitration on the CEI and controls loading of the DMA write buffers and data flows during DMA reads and slave reads and writes.

The CEI Control treats the CEI as two independent, unidirectional busses: an adapter-to-channel engine bus carrying slave address, slave write data and DMA read data, and a channel engine-to-adapter bus, carrying DMA address, slave read data and DMA write data. Different transactions can proceed at the same time as

long as they don't compete for resources. For example, a DMA write utilizes the channel engine-to-adapter bus exclusively, while a slave write utilizes the adapter-to-channel engine bus exclusively, and thus can proceed at the same time.

4.4 PCI Compatibility

PCIO conforms fully with the *PCI Bus Specification Revision 2.0*. PCIO is a 32-bit device, with 32-bit data and 32-bit address. All the PCI Bus I/O pads are "socket-compliant", having a dedicated power rail that can be either 3.3V or 5V. PCIO can be used in either 3.3V or 5V systems. The core logic in PCIO requires 5V.

PCIO is a multi-function device comprising network (Ethernet) and a bridge function (EBus2). There are no provisions for cache coherency built into PCIO.

4.4.1 Little Endian-ness

Like most other PCI Bus devices, PCIO conforms to the little endian byte ordering, that is, the most significant byte of a 32-bit number is at the highest byte address.

The channel engines within PCIO are big endian. The Bus Adapter performs the transformation described in TABLE 4-1, taking into account whether the data object is a data stream, programmed I/O or descriptor.

PCIO supports byte stacking when accessing the PROM and other devices on the EBus2. The 32-bit word assembled from successive reads of the 8-bit PROM/EBus2 is returned in little endian format. This is still compatible with CPU code fetches, which are always big endian, since the CPU bridge (U2P) incorporates a fixed byte lane alignment, converting the data in this case back to big endian.

4.4.2 Commands

PCI Local Bus reference: §3.1.1 and §3.1.2

PCIO responds to a limited number of PCI Bus commands, as summarized in TABLE 4-2. Note that in determining the number of cache lines to transfer, the 'Cache Line Size' value in the Configuration Space is utilized.

TABLE 4-2 PCI Bus Commands Implemented and Generated by PCIO

C/BE [3:0]#	Command Type	PCIO as Target	PCIO as Initiator
1010	Configuration Read	Implement	n/a
1011	Configuration Write	Implement	n/a
0110	Memory Read	Treated identically, generates successive transactions in the CEI	Less than one full cache line
1110	Memory Read Line		Exactly one cache line
1100	Memory Read Multiple		More than one cache line, or when the prefetch mechanism in bridge needs to be activated.
0111	Memory Write	Treated identically, generates successive transactions in the CEI	Less than one cache line, or a non-integer number of cache lines
1111	Memory Write and Invalidate		One or more **full** cache lines

Specifically, PCIO does not initiate Configuration commands, or initiate or implement Interrupt Acknowledge, Special Cycles, I/O Read or Write or Dual Address Cycle.

4.4.3 Basic Transfer Control

PCI Local Bus reference: §3.2.1

As an initiator, PCIO will not use IRDY# to force wait states into the transaction (i.e. PCIO is always ready to either source or receive data). As a target, PCIO will monitor IRDY# to properly control flow.

4.4.4 Addressing

PCI Local Bus reference: §3.2.2

As a target, PCIO does not implement the I/O address space; memory space accesses are performed only if linear burst ordering is requested, *i.e.* AD[1:0]=00. For any other encoding of AD[1:0], the transaction is disconnected after the first data phase.

As an initiator, only transactions with linear burst ordering are generated.

4.4.5 Byte Alignment

PCI Local Bus reference: §3.2.2 and 3.2.3

As a target, PCIO recognizes only a limited number of byte enable combinations, as summarized in Table 4-3. All other transactions will be terminated with target-aborts, except for configuration cycles, where any byte enable combination is allowed.

Table 4-3

Transaction Type	C/BE3#	C/BE2#	C/BE1#	C/BE0#
No-op (ignore)	1	1	1	1
Byte at address 0x00	1	1	1	0
Byte at address 0x01	1	1	0	1
Byte at address 0x02	1	0	1	1
Byte at address 0x03	0	1	1	1
Half word at address 0x00	1	1	0	0
Half word at address 0x02	0	0	1	1
Word at address 0x00	0	0	0	0
Error	Any Other			

Since successive data phases as a target are treated as individual transactions, byte enables can change between data phases.

As an initiator, PCIO will generate only byte, half-word and word transactions with encoding as in Table 4-3. Byte enables will not change between data phases.

4.4.6 Transaction Termination

PCI Local Bus reference: § 3.3.3

As a target, PCIO will never terminate a transaction with a retry, disconnect or target-abort, except as noted in Section 4.4.4 (non-linear burst ordering) and Section 4.4.5 (byte alignment).

As an initiator, PCIO will generate master-initiated terminations based on expiration of the latency timer and the status of the GNT# signal. Memory Write and Invalidate commands will ignore the latency timer until a cache line boundary is reached. Note that the channel engine is not notified of this condition, and the data transfer is restarted at the next memory address.

PCIO will generate master-abort terminations when no target responds to a command initiated by PCIO. The master-abort detected bit in the corresponding function's configuration space is set, and an error is reported to the channel engine, which will then generate an interrupt.

PCIO can deal with target retry and disconnect terminations, hiding this conditions from the channel engines requesting the transactions. Target-abort terminates the transactions and the channel engine gets notified of the error. There are no timeout counters or retry counters to recover from targets that do not respond properly.

4.4.7 Fast Back-to-Back

PCI Local Bus reference: §3.4.2

PCIO has the Fast Back-to-Back Capable bit hardwired to '1' in the Status register, indicating that, as a target, it meets the requirements for performing fast back-to-back cycles.

4.4.8 Arbitration Parking

PCI Local Bus reference: §3.4.3

PCIO does not require or expects the PCI Bus to be parked so that it is granted to it when no other agent is requesting it.

At the expense of an extra arbitration latency clock for other agents when the bus is IDLE, optionally parking PCIO will reduce its arbitration latency to zero clocks.

4.4.9 Latency

PCI Local Bus reference: §3.4.4

As a target, PCIO tries to minimize target latency whenever possible. Slave writes are buffered at the EBus2 channel engine, so that latency should be minimized. Register writes to internal PCIO locations have relatively fast throughput as well.

Slave reads from internal PCIO locations, although not buffered, are relatively fast, possibly in the order of four PCI clocks. However, if a slave read starts when a DMA write transaction is taking place in the CEI, the slave read will be held until the write finishes, since they both use the same CEI resource. For a 64-byte burst on with a 32-bit word size, this can be up to 16 clocks.

Note – Version 2.1 of the PCI Specification recommends/requires slave reads expected to take more than 16 clock cycles to be disconnected (pseudo split transaction). PCIO does not implement this feature. Moreover, in the case of byte-stacked reads to slow EBus2 devices, read latencies can be very high.

Slave reads from the EBus2 devices, or slave writes when the write buffers are backed up, have their latency determined by the timing of the external devices, some of which are rather slow. In addition, the EBus2 can be busy with a DMA transfer resulting in additional latency. Yet another source of latency is byte stacking from the PROM.

As an initiator, PCIO contains internal buffering for each of its internal functions to minimize the impact of bus latency. PCIO will generate burst of up to 64 bytes directed, under normal circumstances, to the host bridge (U2P). Assuming a relatively short delay for the first target-ready, and one data phase per clock thereafter, Min_Gnt should be set to about 24 clock cycles.

4.4.10 Exclusive Access

PCI Local Bus reference: §3.5

PCIO does not implement or support exclusive access and does not have a LOCK# signal.

4.4.11 Device Selection

PCI Local Bus reference: §3.6.1

PCIO will assert DEVSEL# when selected as a target at ‘medium’ speed (*i.e.* two clocks after assertion of FRAME#). This facilitates design of a synchronous interface, and does not affect performance, since internal transactions can proceed in parallel with assertion of DEVSEL#.

CHAPTER 5

Channel Engine Interface

The Channel Engine Interface is a modular, bus-independent interconnect intended to serve as the backbone of the FEPS and PCIO designs.

5.1 Goals

The Channel Engine Interface (CEI) has the following goals:

- Modular
- Extensible (number of master and slave ports)
- Bus-independent and free of external timing constraints, but closely resemble SBus protocol
- Interface speed limited by existing ASIC technology

5.2 Terminology / Glossary

Channel Engine Interface (CEI) refers to the interconnect mechanism and protocol between the Channel Engines and Bus Adapter.

Bus Adapter refers to all the logic between the CEI and the external bus.

Channel Engine refers to each of the individual, self-contained functional blocks in PCIO: Ethernet and EBus2.

Port refers to the connection of a Channel Engine to the CEI. A Channel Engine can have master or slave ports, or both.

Double Word refers to a 64-bit word (8 bytes), moved in a single data beat on the CEI.

5.3 Signals

Signals in the CEI are either shared, going from the Bus Adapter to all of the Channel Engines, or dedicated, going from one Channel Engine to the Bus Adapter or vice versa.

The signals are summarized in Table 5-1. “Direction” is indicated as seen from the Channel Engine perspective (*i.e.* “input” means driven by Bus Adapter and received by Channel Engine). All signals are active high, with the exception of ce_sack[2:0]_l and ce_drack[2:0]_l, which follow the SBus encoding.

Table 5-1 Channel Engine Interface Signals

Signal Name	Direction	Function
General		
ce_rst	Input, shared	Reset, either hardware / power-on of from test control logic
ce_clk	Input, shared	Channel Engine interface clock. All signals are synchronous to this clock. Maximum frequency is determined by ASIC technology and is at least 33 MHz + 10%
Datapath		
ce_dout[63:0]	Output, dedicated	Multiplexes master address (32-bit) and transaction size / type, master data (32 / 64-bit) and slave data (32-bit)
ce_din[63:0]	Input, shared	Multiplexes master data (32 / 64-bit), slave address and size and slave data (32-bit)
Slave Port		
ce_sel	Input, dedicated	Slave select, indicates the start of a slave transaction, remain asserted until the transaction is completed
ce_sack[2:0]_l	Output, dedicated	Slave acknowledge, issued by the slave to indicate acceptance of data or data ready. The encoding is as follows: 111 — Idle / Wait 110 — Error acknowledge 101 — Byte acknowledge 011 — Word (32-bit) acknowledge 001 — Half-word (16-bit) acknowledge 100, 010, 000 — Reserved

Table 5-1 Channel Engine Interface Signals *(Continued)*

Signal Name	Direction	Function
Master Port		
ce_br	Output, dedicated	Bus request, issued by master port to indicate it needs to perform a read or write DMA transaction
ce_bg	Input, dedicated	Bus grant, issued by the Bus Adapter to a specific master port when it is ready to perform a DMA transaction
ce_drack[2:0]_l	Input, shared	DMA read acknowledge, issued by Bus Adapter to indicate data is ready during DMA reads. Data follows ce_drack_l on the next clock cycle. Encoding is as follows: 111 — Idle / Wait 110 — Error acknowledge 101 — Byte acknowledge 011 — Word (32-bit) acknowledge 001 — Half-word (16-bit) acknowledge 100, 010, 000 — Reserved
ce_dwack	Input, shared	DMA write acknowledge, issued by Bus Adapter to indicate it can accept new data from Channel Engine. Bus Adaptor latches data on the same clock edge it issues the acknowledge.
ce_memdone	Input, dedicated	DMA write transaction done, is issued by the Bus Adapter when a previously posted CEI transaction actually completes on the external bus
Error and Exception Reporting		
ce_lerr	Input, shared	Late Error, is issued by Bus Adapter (in SBus systems) if sb_lerr_ is detected after a DMA read operation. Masters must monitor ce_lerr exactly two clock cycles after each n0n-idle ce_drack_l
ce_dwerr	Input, shared	DMA Write Error, is issued by the Bus Adapter together with ce_memdone to indicate sb_lerr_ was detected or a bus error occurred during a DMA write transaction
ce_perror	Input, shared	Parity Error, is issued by the bus adapter during slave port write access or a DMA read transaction, to indicate a parity error on the external bus.
ce_int	Output, dedicated	Interrupt(s): number varies by Channel Engine

The data busses ce_din[63:0] and ce_dout[63:0] are used to encode address, data and transaction type, during both slave and master transactions. Table 5-2 shows the encoding of signals for slave transactions. All slave transactions are 32-bit or less.

Table 5-2 Slave Access Encoding of ce_din and ce_dout

Signal	Slave Encoding	Function
ce_dout[31:0]	slave_dout[31:0]	Slave write data
ce_din[31:0]	slave_din[31:0]	Slave read data
ce_din[63]	slave_rd	Transfer direction: 0 — slave write 1 — slave read
ce_din[62:60]	slave_size[2:0]	Transfer size: 000 — Word transfer 001 — Byte transfer 010 — Half-word transfer 011, 1x× — Reserved
ce_din[59:32]	slave_addr[27:0]	Slave address, 28-bit address space
ce_dout[63:32]	Reserved	Unused

During slave accesses, the 28-bit address space per channel engine is qualified by slave select (ce_sel). Slave ports can use as many or as few of the slave address bits as necessary to further decode addresses within their space.

Table 5-3 shows the encoding of signals for DMA transactions. DMA transactions use either 32- or 64-bit data paths.

Table 5-3 DMA Transaction Encoding of ce_din and ce_dout

Signal	DMA Encoding	Function
Address Phase		
ce_dout[31:0]	dma_addr[31:0]	DMA (virtual) address
ce_dout[63]	dma_rd	Transfer direction: 0 —DMA write (Channel Engine to Memory) 1 — DMA read (Memory to Channel Engine)

Table 5-3 DMA Transaction Encoding of ce_din and ce_dout *(Continued)*

Signal	DMA Encoding	Function
ce_dout[62:60]	dma_size[2:0]	Transfer size: 000 — Word transfer 001 — Byte transfer 010 — Half-word transfer 011 — Extended (64-bit) transfer 100 — Four word burst (16 bytes) 101 — Eight word burst (32 bytes) 110 — Sixteen word burst (64 bytes) 111 — Two word burst (8 bytes)
ce_dout[59:57]	dma_etsize[2:0]	Extended transfer size, valid when dma_size=011: 011 — One double word burst (8 bytes) 100 — Two double word burst (16 bytes) 101 — Four double word burst (32 bytes) 110 — Eight double word burst (64 bytes) 111 — Reserved, place holder for 128 bytes 000, 001, 010 — Reserved
ce_dout[56]	dma_desc	Descriptor traffic. Issued by Channel Engine to differentiate between descriptor and DMA data transfers: 0 — DMA data 1 — Descriptor
ce_dout[55:32] ce_din[63:0]	Reserved	Unused
Data Phase		
ce_din[63:0]	dma_din[63:0]	DMA data in, 32- or 64-bit wide
ce_dout[63:0]	dma_dout[63:0]	DMA data out, 32- or 64-bit wide

5.4 Transactions

5.4.1 Slave Write

Slave Write transactions are shown in FIGURE 5-1. The first transaction shows the slave acknowledge (ce_sack[2:0]_l) valid with the proper port size the clock edge following slave select (ce_sel). Data is clocked in by the slave during the same edge that port_size is asserted.

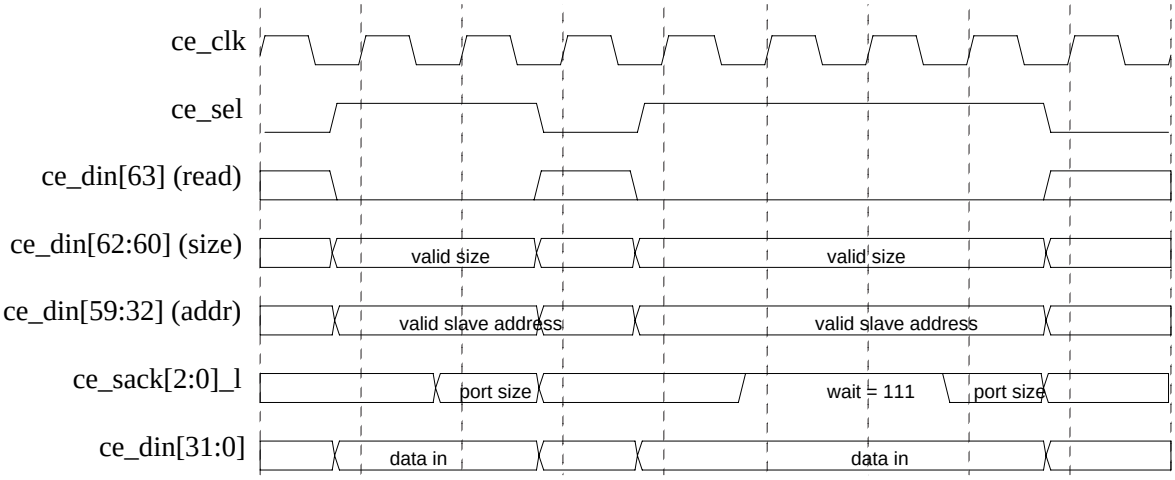

FIGURE 5-1 Slave Write Transactions

The second transaction shows the slave asserting wait (ce_sack[2:0]_l=111) for two clocks. Transfer direction, size and address remain stable until the slave acknowledges the port size and clocks in the data.

Slave write only use the ce_din[63:0] bus. Therefore, there is no need for the Bus Adapter to arbitrate for the CEI before it can assert slave select (ce_sel) and drive address and data, since the only other transaction that can be going on simultaneously is a DMA write, which only uses the ce_dout]63:0] bus. Channel Engines can be designed so that they can respond to slave writes even when a DMA write is in progress.

Note – The Ethernet Channel Engine currently **requires** that “data in” remain valid during PIO writes to the ERX fifo for one clock after deassertion of ce_sel.

5.4.2 Slave Read

Slave Read transactions are shown in FIGURE 5-2. The first transaction shows the slave acknowledge (ce_sack[2:0]_l) valid with the proper port size the clock edge following slave select (ce_sel). Data is valid for the bus adapter during the following clock edge.

On the second transaction, the slave asserts wait acknowledge for one clock cycle, then acknowledges the port size and presents the data a clock cycle after that.

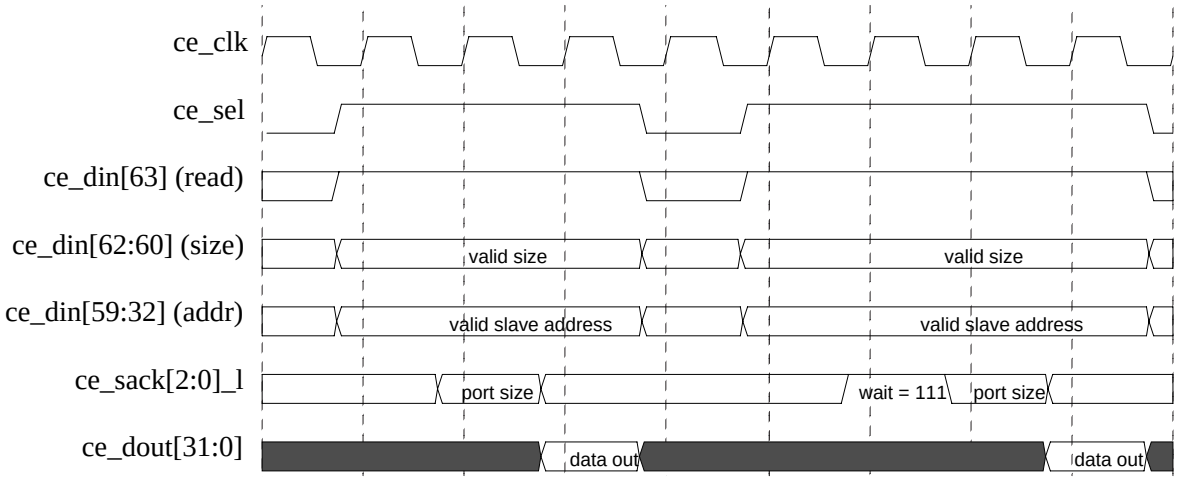

FIGURE 5-2 Slave Read Transactions

Note – The Ethernet Channel Engine currently **requires** that ce_din[63] (read) be valid one clock cycle before ce_sel is asserted. Otherwise, slave read data is latched at the wrong time.

5.4.3 DMA Write (Channel Engine to Memory)

FIGURE 5-3 shows a DMA write transaction using 32-bit wide data. The master port present the DMA address, direction and size together with its internal bus request. Once the channel engine interface is granted, the master deasserts its bus request, and bursts the data to the bus adapter. The transaction terminates with the last data beat, and the arbiter deasserting bus grant.

The Bus Adapter can optionally extend the cycle by withholding ce_dwack for each of the data phases.

FIGURE 5-2 shows a DMA write transaction using 64-bit extended mode. The size field (ce_dout[62:60]) is set to 011, and the extended size information is presented on etsize (ce_dout[59:57]). During the data phase, ce_dout[63:0] carries the transfer data, DMA Read (Memory to Channel Engine).

A DMA Read transaction is shown in FIGURE 5-5. After the bus grant, the Bus Adapter issues wait acknowledge (ce_drack[2:0]_l=111) until data is received from the external bus. Data for the channel engine follows the port acknowledge on the next clock cycle.

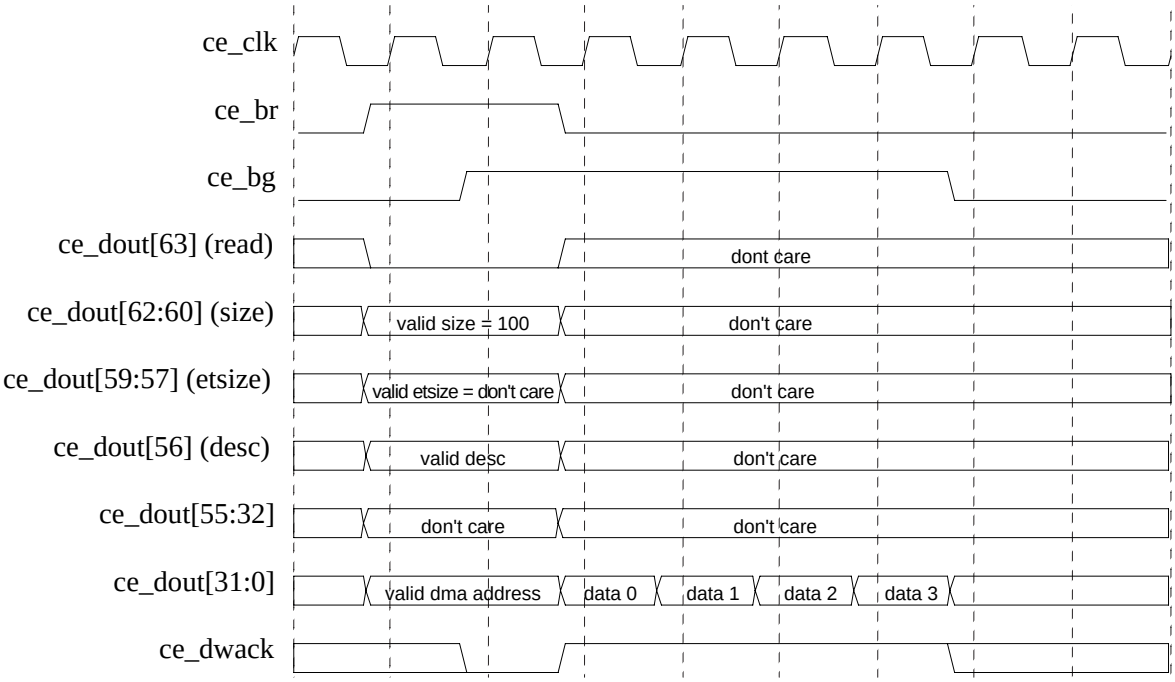

FIGURE 5-3 DMA (Master) Write Transaction: 32-bit - 16-byte burst

For 64-bit Extended Mode, size and etsize are set appropriately, and timing is the same as 32-bit mode.

DMA reads are interlocked with the external bus. The CEI will not be granted until the external bus is granted. Slave transactions may occur while request (ce_br) is asserted but the bus as not been granted (*i.e.* ce_bg deasserted). During this time, Channel Engine must be able to respond to slave accesses to avoid deadlocks.

Furthermore, a rerun on the external bus, although not exposed to the CEI, would result in long delays between the bus grant (ce_bg), and the first data acknowledge (ce_drack_l=port size). During this time, the external bus is relinquished by the Bus Adapter, and one or more slave transactions may occur, even though bus grant (ce_bg) continues to be asserted.

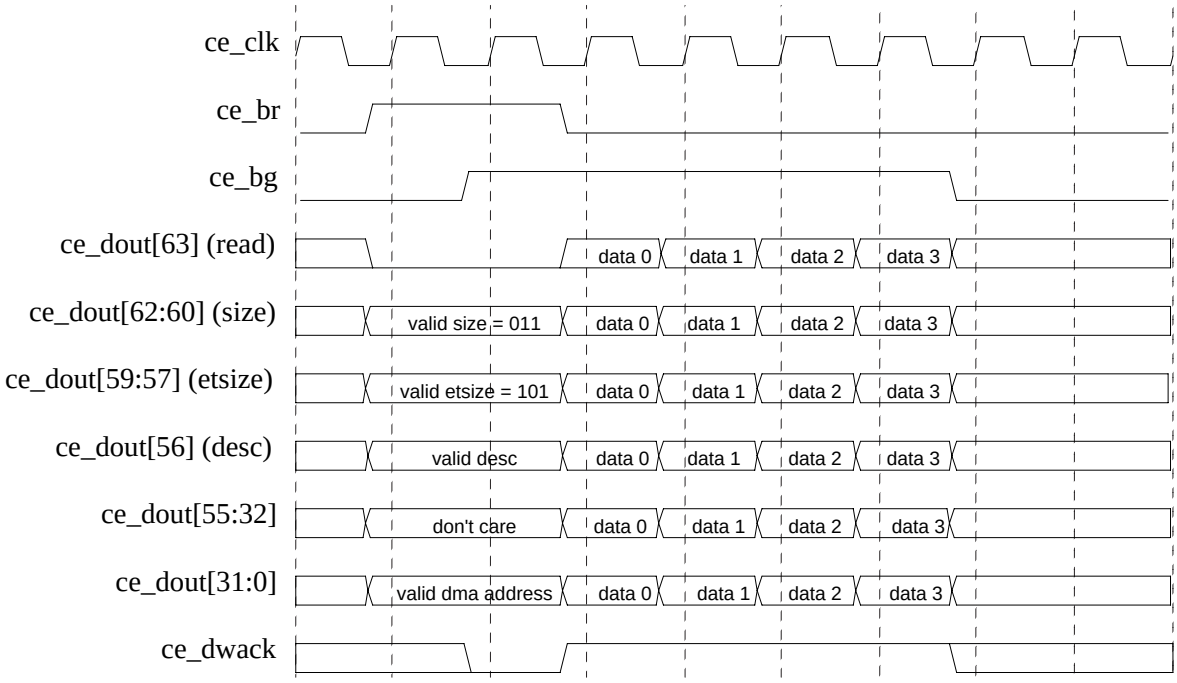

FIGURE 5-4 DMA (Master) Write Transaction: Extended Mode - 32-byte burst

5.5 Data Ports

For all master and slave transactions, data is replicated in all logical data ports. For slave byte accesses, data is replicated in bits [31:24], [23:16], [15:8] and [7:0]; for slave half word accesses, data is replicated in bits [31:16] and [15:0].

For master byte accesses, data is replicated in bits [31:24], [23:16], [15:8] and [7:0]; for master half word accesses, data is replicated in bits [31:16] and [15:0].

When driving data, both Bus Adapter and Channel Engine must replicate it into all data ports as outlined above. When receiving data, both Bus Adapter and Channel Engine can select any of the available data ports.

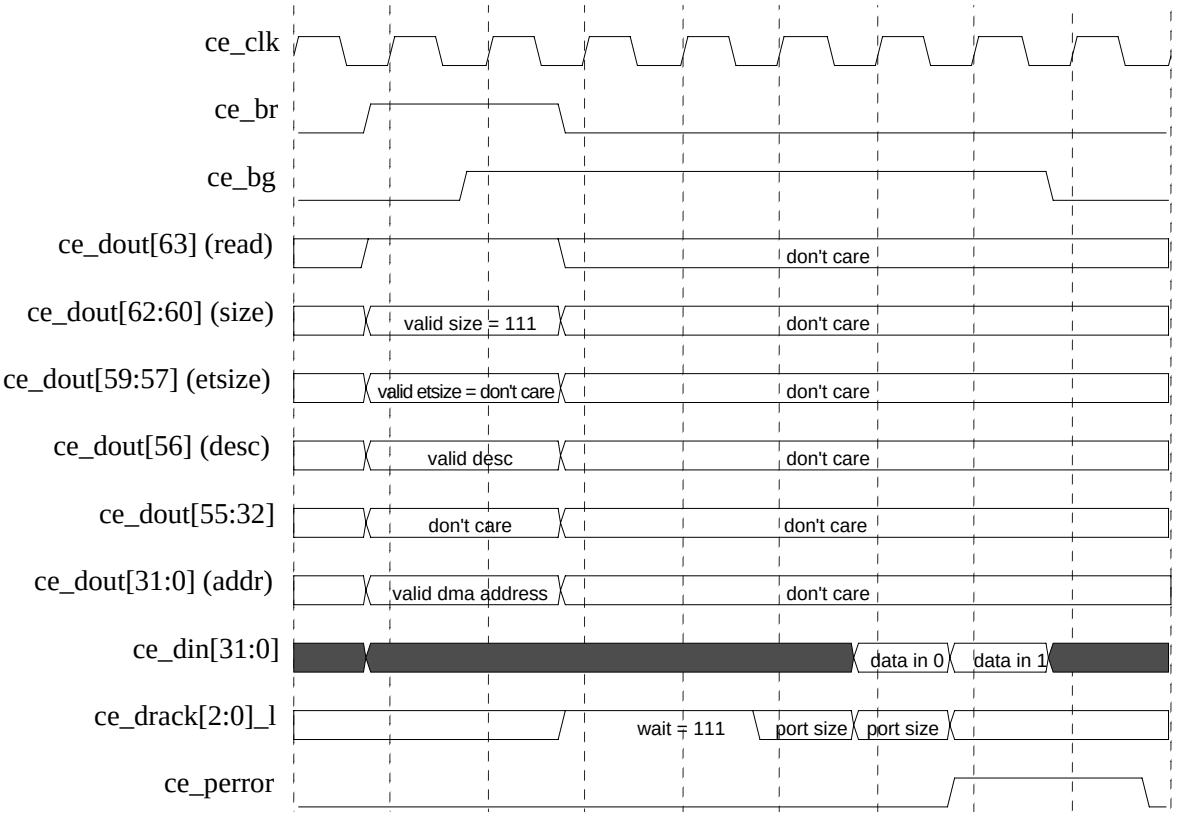

FIGURE 5-5 DMA (Master) Read Transaction: 32-bit - 8-byte burst

5.6 Error Handling & Reporting

5.6.1 Slave Transactions

5.6.1.1 Parity

The Bus Adaptor checks for data (and address, in PCI) parity during slave writes. Parity errors are optionally reported on the external bus.

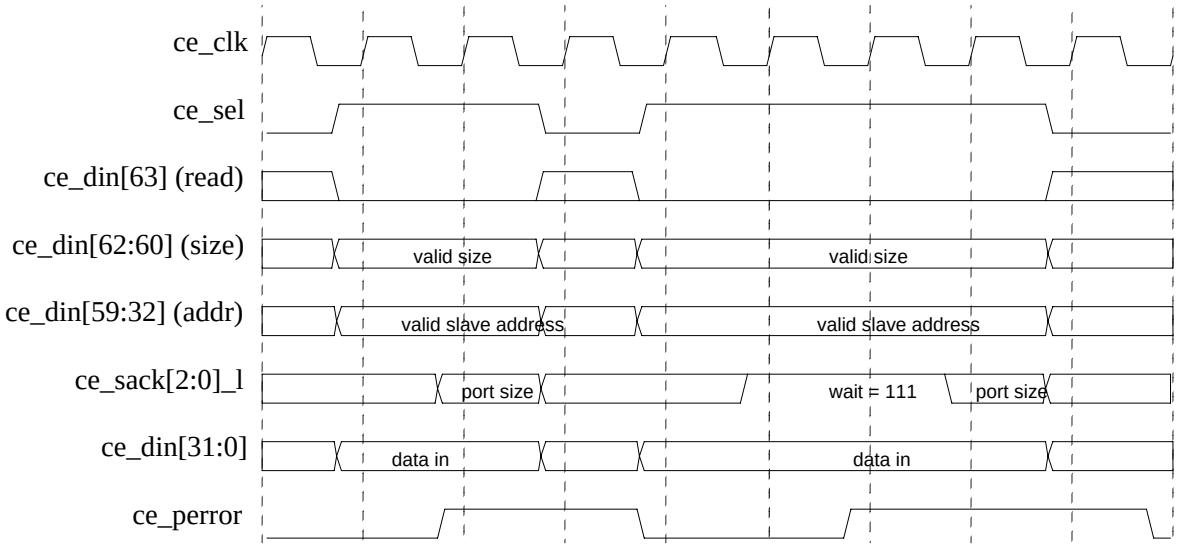

FIGURE 5-6 Reporting a parity error during a slave write

FIGURE 5-5 shows how the parity error is reported to the channel engine. ce_perror must be asserted in the cycle following the error data, and remain asserted until one clock after the port acknowledge. In the first transaction shown, the parity error is detected during the first clock cycle with valid data. During the second transaction, the parity error does not get detected until the second clock cycle with valid data. In both cases, ce_perror remains asserted until one clock cycle after the port size acknowledge.

The channel engine is responsible for reporting the parity error back to the system via an interrupt, and logging the error in a status register. The transaction itself can be ignored or completed, but it must be terminated correctly.

PCI address parity is reported using the PCI SERR# signal. No slave cycle is performed.

5.6.1.2 Late Error

There is no concept of Late Error during slave cycles on the Channel Engine Interface. The Bus Adapter should not generate Late Error.

5.6.1.3 Access Error and Bus Sizing

Error acknowledge (ce_sack[2:0]_l=110) can be used by the Channel Engine to indicate errors back to the Bus Adapter. The Bus Adapter will then pass along the error to the external bus, via an error acknowledge in SBus or System Error in PCI. The Channel engine must also report access errors to the systems via an interrupt and log the error in a status register.

Access size errors, for example when a word-only register is accessed as a byte, can be acknowledged to the Channel Engine Interface, or error-acknowledged. Bus sizing is not supported in the Channel Engine Interface.

Access to invalid memory locations can be ignored and acknowledged, or error-acknowledged.

No time-out mechanism exists in the Channel Engine Interface. A selected slave must respond promptly with a port size acknowledge or an error acknowledge.

5.6.2 DMA Transactions

5.6.2.1 Parity

The Bus Adapter checks data parity during DMA reads only. Parity errors detected are not reported on the external bus, but are sent along to the Channel Engine one clock after the port acknowledge. Timing is like in FIGURE 5-5.

Errors are reported back to the system by the Channel Engine via an interrupt, and logged in a status register.

5.6.2.2 Late Error

On CEI, Late Errors encountered during DMA reads are passed along to the Channel Engine, two clocks after the port size acknowledge. During DMA writes, the Bus Adaptor does not assert ce_memdone until two clocks after the last port size acknowledge. If a Late Error occurs, it will asset ce_dwerr together with ce_memdone.

There is no Late Error in PCI.

5.6.2.3 Bus Errors

During DMA reads, bus errors are sent to the Channel Engine as error-acknowledges. During DMA writes, any bus error encountered is reported via the ce_dwerr mechanism described above.

5.6.2.4 Bus Sizing

Bus Adapter are not required to support bus sizing on the external bus.

5.7 Arbitration

The CEI assumes all Channel Engines and Bus Adapter cooperate with each other and function correctly. Since there are no signals that can be driven by more than one master, there is no possibility of collisions. However, if more than one Channel Engine believes it owns the CEI at any one time, things will become utterly confused.

5.8 Design Guidelines

5.8.1 Timing

All signals must be synchronous to the Channel Engine Interface clock (ce_clk). The nominal clock cycle is 30 nS.

5.8.1.1 Cycle Time Budget Allocation

ce_clk to valid signal: maximum 15 nS, measured at the driving end, with a 4× standard load

valid signal setup time to ce_clk: minimum 10 nS, measured at the receiving end, assuming a standard driver (1×)

5.8.2 Signal Loading

One standard load for ce_din and shared signals, up to two standard loads for dedicated signals.

Dedicated signals are driven by standard drivers. ce_din and shared signals are driven by “strong” drivers.

CHAPTER 6

Ethernet Channel Engine

6.1 Introduction

6.1.1 Overview

The Ethernet Channel provides the network interface functionality for two highly integrated combo chips:

- FEPS — SBus version
- PCIO — PCI version

FIGURE 6-1 shows the positioning of the Ethernet Channel in the overall FEPS architecture. The following brief description is for the Ethernet channel of FEPS. It is applicable to the Ethernet channel of PCIO with small changes.

The Ethernet Channel implements two major functions:

- Provides the Media Access Control (MAC) function for a 10/100Mbps CSMA/CD protocol based network
- Provides a high performance two-channel DVMA host interface between the MAC and the PCI Bus

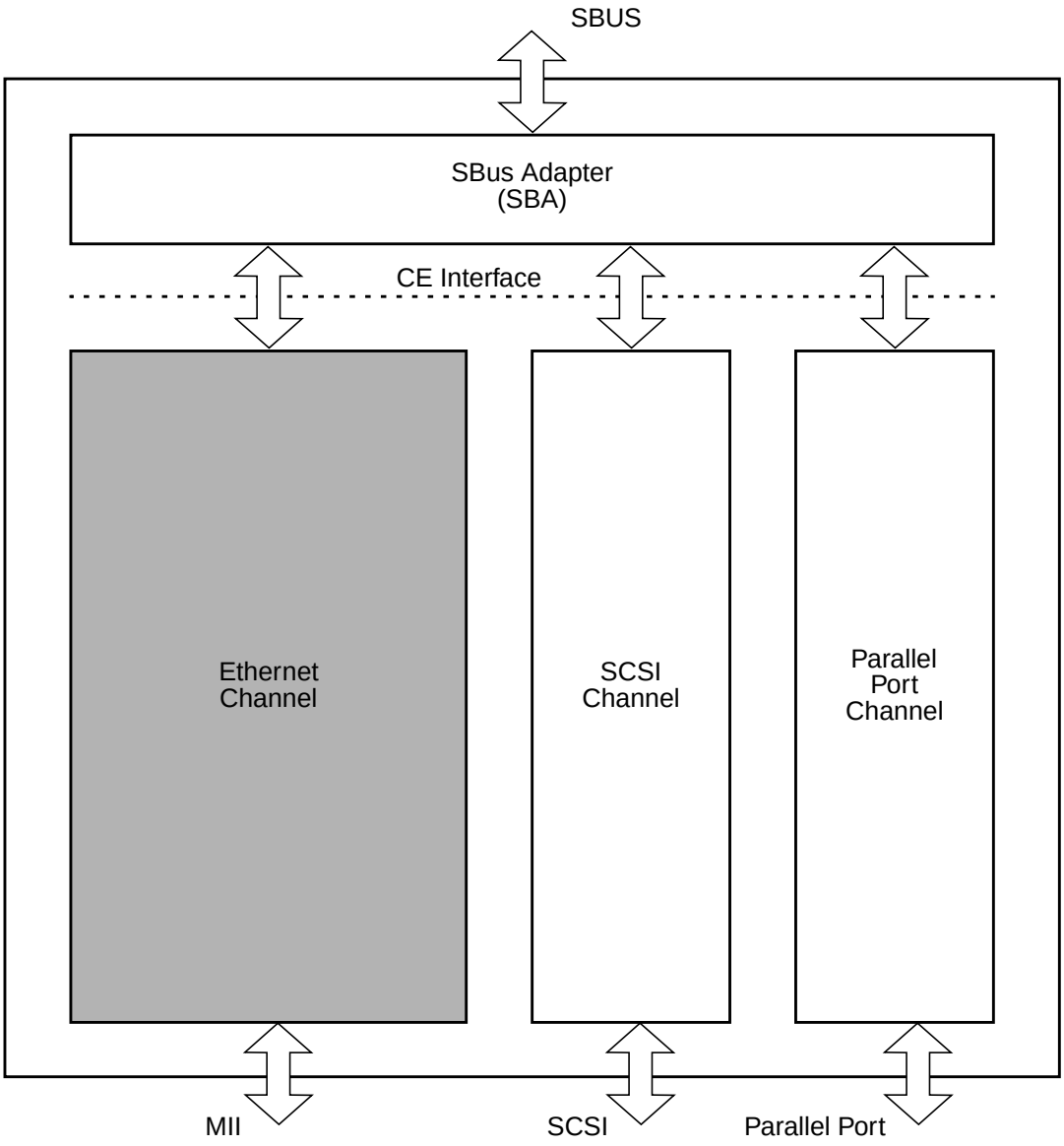

FIGURE 6-1 FEPS Block Diagram

6.1.2 Major Components

6.1.2.1 Functional Blocks

The Ethernet Channel is comprised of five major blocks:

MAC Core

The MAC Core implements the IEEE 802.3 MAC protocol for 10/100Mbps CSMA/CD networks.

MIF

The Management Interface block implements the management portion of the MII interface to an external transceiver, as defined in the IEEE 802.3 MII specification.

ETX

The Ethernet Transmit block provides the DMA Engine for transferring frames from the host memory to the MAC. It contains a local buffer of 2K bytes for rate adaptation between the available bandwidth on the PCI Bus and on the network.

ERX

The Ethernet Receive block provides the DMA Engine for transferring frames from the MAC to the host memory. It contains a local buffer of 2K bytes for rate adaptation between the available bandwidth on the network and on the PCI Bus.

SEB

The Shared Ethernet Block contains common functions that are shared between the ETX and ERX blocks. It also separates the DMA data path from the Programmed IO data path.

6.1.2.2 Interfaces

The Ethernet Channel interfaces to the rest of FEPS via two exposed interfaces:

Channel Engine Interface (CEI)

The CEI connects the Ethernet Channel to the SBA. This interface is defined to be identical for all three channels in FEPS.

Media Independent Interface (MII)

The MII connects the Ethernet Channel to an external Ethernet transceiver. It conforms with the IEEE 802.3u defined MII.

6.1.3 Features List

- Conforms to ISO/IEC 8802-3 and IEEE 802.3u standards
- Programmable network parameters for standard's extension and/or private applications
- Supports full duplex operation
- Flexible transceiver choice via the MII
- Extensive support for network management
- Local on-chip buffers (FIFOs) of 2K bytes in each direction
- Host packet management via descriptor rings
- TCP checksum support in hardware
- Transmit "gather" function
- Programmable first byte alignment on receive
- Support for 32-bit or 64-bit PCI Bus, maximum of 64-byte bursts

6.2 Functional Description

The following detailed description is for the Ethernet channel of the PCIO chip.

6.2.1 Overview

The Ethernet Channel is a dual-channel intelligent DMA controller on the system side, and an IEEE 802.3 MAC on the network side. It was designed as a high performance full duplex device, allowing for simultaneous transfers of data from/to host memory to/from the “wire”. The MAC portion of the Ethernet Channel is compliant with the IEEE 802.3u (100BASE-T) standard.

Packets scheduled for transmission are transferred over the PCI Bus into a local transmit FIFO, and are later transferred to the TX_MAC core for protocol processing and transmission over the medium. A programmable transmit threshold is provided to enable the transmission of the frame. The reverse process takes place in the receive path. Packets received from the medium are processed by the RX_MAC, loaded into the receive FIFO, and are later transferred to the host memory over the PCI Bus. The receive threshold for data transfers is 128 bytes.

At the device driver level, the user deals with transmit and receive descriptor ring data structures for posting packets and checking status. In the transmit case, packets may be posted to the hardware in multiple buffers (descriptors), and the transmit DMA engine will perform “data gather”. In the receive case, the receive DMA engine will store an entire packet in each buffer that was allocated by the host. “Data scatter” is not supported, but instead a programmable first byte alignment offset within a burst is implemented.

For TCP packets, hardware support is provided for TCP checksum computation. On transmit, it is assumed that the entire packet is loaded into the local FIFO before its transmission begins. The checksum is computed “on-the-fly” while the packet is being transferred from the host memory into the local FIFO. The checksum result is then “stuffed” into the appropriate field in the packet, and the transmission of the frame begins. On receive, checksum is computed on the incoming data stream from the MAC core, and the result is posted to the device driver as part of the packet status in the descriptor.

6.2.2 Hardware Architecture

FIGURE 6-2 shows the top level architecture of the Ethernet Channel.

The architecture was defined to achieve the following goals:

- High Throughput:

Full duplex operation with no hardware bottlenecks in the data path. The throughput is limited either by the available bandwidth on the PCI Bus or by the maximum throughput on the network. The data path is optimized to achieve maximum utilization of the PCI Bus bandwidth

- Minimal software Overhead:

- One or less interrupts per packet
- Non-restricted transmit “data gather” in hardware
- Programmable first byte alignment within a burst on receive
- Hardware support for TCP checksum

- Modularity:
  - Leverages from existing designs (MAC core)
  - Allows future design leveraging — the functional blocks are partitioned by well defined functions, that interact via well defined interfaces

6.2.2.1 Functional Blocks

The Ethernet Channel is comprised of five major blocks:

MAC Core

The MAC Core implements the IEEE 802.3 MAC protocol for 10/100Mbps CSMA/CD networks. It consists of four major functional modules:

Host Interface Buffer (HIB)

- Implements the Programmed IO interface between the SEB and the MAC core

Transmit MAC (TX_MAC)

- Implements the IEEE 802.3 transmit portion of the protocol
- Implements the slave interface handshake between the ETX and TX_MAC for frame data transfers
- Performs the synchronization between the system clock domain and the transmit media clock domain in the transmit data path

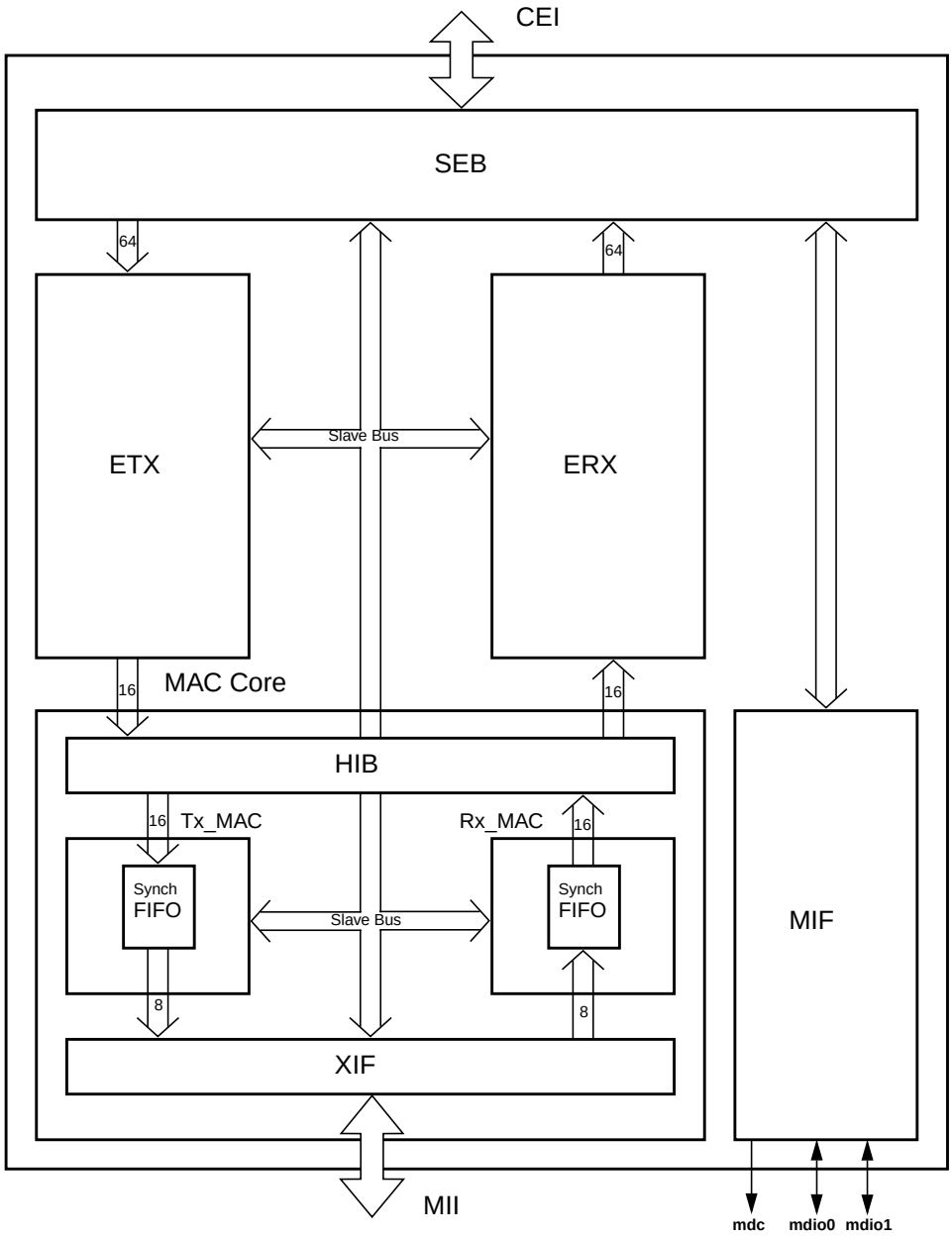

FIGURE 6-2 Ethernet Channel Engine

Receive MAC (RX_MAC)

- Implements the IEEE 802.3 receive portion of the protocol
- Implements the slave interface handshake between the ERX and RX_MAC for frame data transfers
- Performs the synchronization between the system clock domain and the receive media clock domain in the receive data path

Transceiver Interface (XIF)

- Implements the MII interface protocol (excluding the Management Interface)
- Performs the nibble-to-byte and byte-to-nibble conversion between the protocol engine and the MII

MIF

The Management Interface (MIF) implements the management portion of the MII protocol. It allows the host to program and collect status information from two external transceivers, connected to the MII. The MIF supports three modes of operation:

"Bit-Bang" Mode

This mode of operation provides maximum flexibility with minimum hardware support for the serial communication protocol between the host and the transceivers. The actual protocol is implemented in software, and the interaction with the hardware is done via three one-bit registers: data, clock and output_enable. Each read / write operation on a transceiver register would require approximately 150 software instructions by the host.

"Frame" Mode

This mode of operation provides a much more efficient way of communication between the host and the transceivers. The serial communication protocol between the host and the transceivers is implemented in hardware, and the interaction with the software is done via one 32-bit register (Frame Register). When the software wants to execute a read / write operation on a transceiver register, all it has to do is load the Frame Register with a valid instruction ("frame"), and poll the Valid Bit for completion. The hardware will detect the instruction, serialize the data, execute the serial protocol on the MII Management Interface and set the Valid Bit to the software.

Polling Mode

As defined in the IEEE 802.3u MII Standard, a transceiver shall implement at least one status register that will contain a defined set of essential information needed for basic network management. Since the MII does not include an interrupt line, a polling mechanism is required for detecting a status change in the transceiver. In order to reduce the software overhead, the above mentioned polling mechanism has been implemented in hardware. When this mode of operation is enabled, the MIF will continuously poll a specified transceiver register, and generate a maskable interrupt when a status change is detected. Upon detection of an interrupt, the software can read a Local Status Register that will provide the latest contents of the transceiver register, and an indication which bits have changed since it was last read. This mode of operation can only be used when the MIF is in the “Frame Mode”.

ETX

The Ethernet Transmit (ETX) block provides the DMA Engine for transferring frames from the host memory to the TX_MAC. FIGURE 6-3 shows the block diagram of the Transmit DMA Channel.

Following are the major functional modules:

SA I/F

This module implements the relevant subset of the Channel Engine Interface (CEI) handshake between the ETX and the PCI Bus Adaptor. It contains two sub modules: one for the master (DMA) and the other for the slave (PIO) protocols.

CSMC

Contains the Command, Status, Mask and Configuration registers for the ETX channel.

SMM

The ETX System Memory Manager manages the transmit host memory data structures (one descriptor ring of up to 256 data buffers).

Chaining

This module implements the “gather” function of transmit buffers. Proper byte alignment is performed to ensure that data bytes at multiple buffers boundary, belonging to the same packet, are packed sequentially as one segment in the transmit FIFO. The “chaining” of transmit buffers is done on-the-fly during packet data transfer between the PCI Bus and the transmit FIFO.

Checksum

This module provides the hardware support for TCP checksum computation in the transmit data path. This optional function can be enabled or disabled on a per packet basis through a descriptor control bit. If the function is enabled, the software has to provide in the descriptor a “start offset” and a “stuff offset.” For proper operation of the hardware, it is assumed that the entire packet is loaded into the transmit FIFO before its transmission is enabled. There is no restriction on the number of buffers (descriptors) per a packet, as long as the entire frame can fit in the transmit FIFO (2K bytes). Also, it is assumed that the software will initialize the TCP checksum field in the TCP packet to compensate for the fact that the hardware calculates the checksum over the actual IP header, rather than the pseudo-IP header. The 16-bit TCP checksum is computed on-the-fly during packet data transfer between the PCI Bus and the transmit FIFO, starting from the “start offset” until the end of the packet. The result is then “stuffed” at the “stuff offset.”

TxFIFO

The TxFIFO acts as a local buffer for rate adaptation between the available bandwidth on the PCI Bus and on the network. It is large enough (2K bytes) to be able to buffer an entire maximum size standard IEEE 802.3 frame, but it can also hold an unrestricted number of smaller frames. This guarantees minimal performance (no underruns) if the available bandwidth on the PCI Bus is less than the available bandwidth on the network. On the other hand, it guarantees maximum throughput (keeps up with the “wire speed”) if the bandwidth situation is reversed. The logical configuration of the transmit buffer can be either 512x4Bytes or 256x8Bytes, depending on the PCI Bus width. The physical configuration of the transmit buffer is four banks of 128x33bits.

FIFOIn

- Performs the multiplexing of all the data sources to the TxFIFO
- Generates the control signals for loading the TxFIFO
- Generates the frame delimiters (tags) to the TxFIFO

FIFOOut

- Performs “unpacking” of 64-bit data from the TxFIFO to 16-bit data to the TX_MAC
- Generates the tag lines to the TX_MAC

FMM

The FIFO Memory Manager (FMM) module manages the transmit FIFO data structures and provides the following functionality to the transmit DMA engine:

- Makes the dual-port memory core look like a “virtual FIFO”

- Generates “Write” and “Read” pointers to the memory core
- Allows for simultaneous loading and unloading of the TxFIFO
- Generates the TxFIFO status flags
- Maintains a “Shadow Write Pointer” for checksum “stuffing”
- Maintains a “Shadow Read Pointer” for frame re-transmission due to a collision on the network
- Maintains a Packet Counter, a Threshold Register and a Packet Byte Counter for enabling transmission of a frame

Load Control State Machine

This state machine controls the transfer of packet data buffers from the host memory to the TxFIFO. It monitors the PCI Bus DVMA process and controls the execution of the following functions:

- Processing of transmit descriptors
- Execution of the PCI Bus burst transfers
- Loading of the packet data and checksum (if enabled) into the FIFO
- Loading of the control/status word into the FIFO at the end of a frame
- Incrementing the FIFO Write Pointer
- Manipulation of the Shadow Write Pointer (loading of the checksum and pointer restoration)

Unload Control State Machine

This state machine controls the transfer of packet data from the TxFIFO to the TX_MAC.

- Generates the read control signal to the FIFO
- Executes the master-slave handshake between the DMA engine and the TX_MAC
- Increments the FIFO Read Pointer
- Manipulates the Shadow Read Pointer (”tx_retry” due to a collision on the network)

ERX

The Ethernet Receive (ERX) block provides the DMA Engine for transferring frames from the RX_MAC to the host memory. FIGURE 6-4 shows the block diagram of the Receive DMA Channel.

Following are the major functional modules:

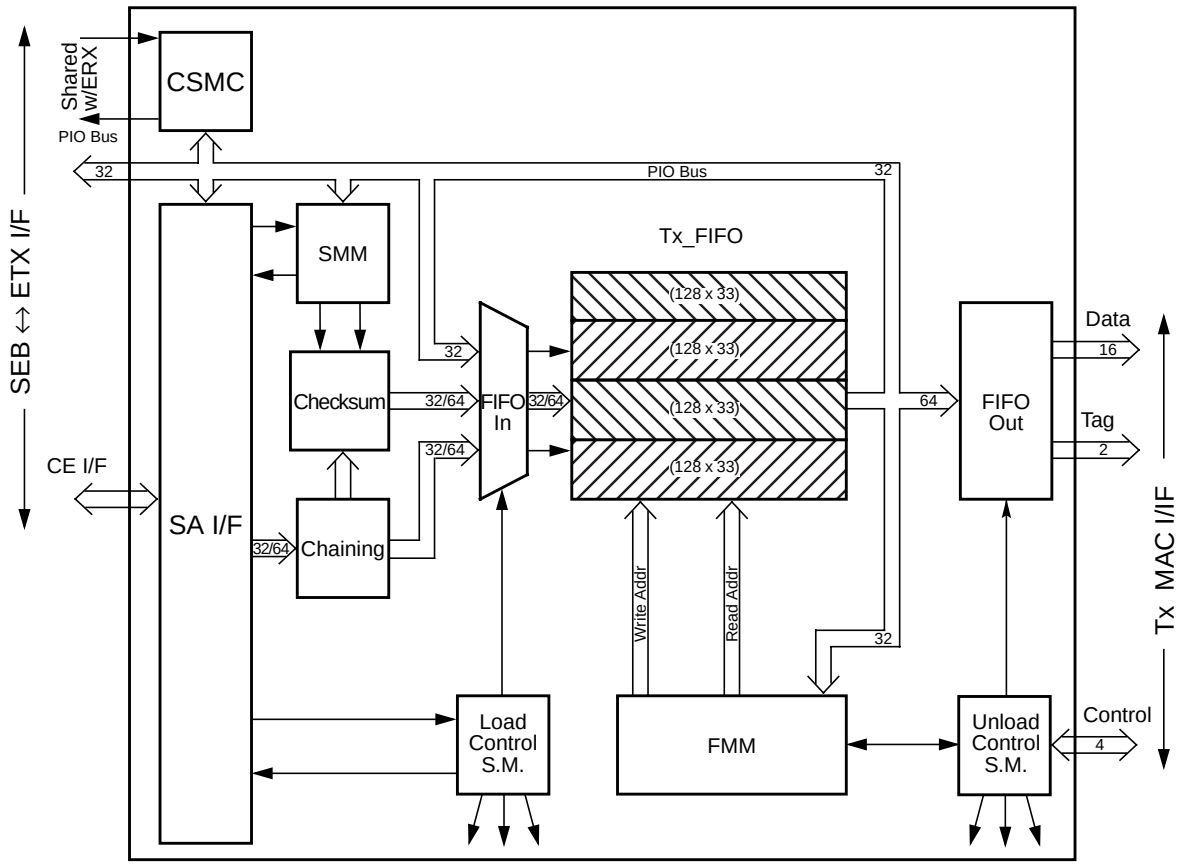

FIGURE 6-3 Transmit DMA Channel

Path

- Implements the relevant subset of the Channel Engine Interface (CEI) between the ERX and the SEB
- Performs alignment of the first word of a frame within an PCI Bus burst
- Performs the 32/64-bit data steering

Slave

Contains the Command, Status, Mask and Configuration registers for the ERX channel, as well as the Slave State Machine.

Checksum

This module provides the hardware support for TCP checksum computation on the receive data path. This optional function is always enabled. The 16-bit checksum value is computed on the entire frame, starting from a programmable 16-bit aligned “start offset”. The computation is performed on-the-fly during packet data transfer between the RX_MAC and the receive FIFO. The result is appended to the end of the frame in the receive FIFO, and is later posted to the device driver as part of the frame status in the receive descriptor. If the software decides to make use of this information, it must compensate/adjust for the additional bytes that have been included in the TCP checksum by subtracting them, and/or for the IP fragmentation by adding up the partial checksums.

RxFIFO

The RxFIFO acts as a local buffer for rate adaptation between the available bandwidth on the network and on the PCI Bus. It is large enough (2K bytes) to be able to receive an entire maximum size standard IEEE 802.3 frame, but it can also hold an unrestricted number of smaller frames. This guarantees minimal performance (a frame is guaranteed to be received “once in a while”) if the available bandwidth on the PCI Bus is less than the available bandwidth on the network. On the other hand, it guarantees maximum throughput (keeps up with the “wire speed”) if the bandwidth situation is reversed. The logical configuration of the receive buffer can be either 512x4Bytes or 256x8Bytes, depending on the PCI Bus width. The physical configuration of the receive buffer is four banks of 128x33bits.

FIFOIn

- Performs alignment of the first byte of a frame within a 32/64-bit word that is loaded into the RxFIFO
- Performs “packing” of 16-bit data from the RX_MAC to 32-bit data that is loaded into the RxFIFO
- Performs the multiplexing of all the data sources to the RxFIFO
- Generates the control signals for loading the RxFIFO
- Generates the frame delimiters (tags) in the RxFIFO

FMM

The FIFO Memory Manager (FMM) module manages the receive FIFO data structures and provides the following functionality to the receive DMA engine:

- Makes the dual-port memory core look like a “virtual FIFO”

- Generates “Write” and “Read” pointers to the memory core
- Allows for simultaneous loading and unloading of the RxFIFO
- Generates the RxFIFO status flags
- Maintains a “Shadow Write Pointer” for recovering from certain receive errors (frame fragments)
- Maintains a Packet Counter for enabling frame transfers from the RxFIFO to the host memory

Load Control State Machine

This state machine controls the transfer of packet data from the RX_MAC to the RxFIFO.

- Generates the load control signal to the FIFO
- Executes the master-slave handshake between the DMA engine and the RX_MAC
- Increments the FIFO Write Pointer
- Manipulates the Shadow Write Pointer (”early rx_abort”)

ERX Unload

This module controls the transfer of packet data from the RxFIFO to the host memory.

- Initiates CEI transactions
- Increments the FIFO Read Pointer
- Recovers from certain error conditions detected by the RX_MAC (”late rx_abort”)
- Monitors the PCI Bus DVMA process
- Processing of receive descriptors
- Execution of the CEI burst cycles
- Manages the receive host memory data structures (one descriptor ring of up to 256 data buffers)

SEB

This block contains common functions that are shared between the ETX and ERX blocks. It performs the first level arbitration between the receive and transmit DMA channels for access to the PCI Bus, and provides one common interface between the Ethernet Channel and the PCI Bus Adapter. The arbitration mechanism is “Round-Robin”.

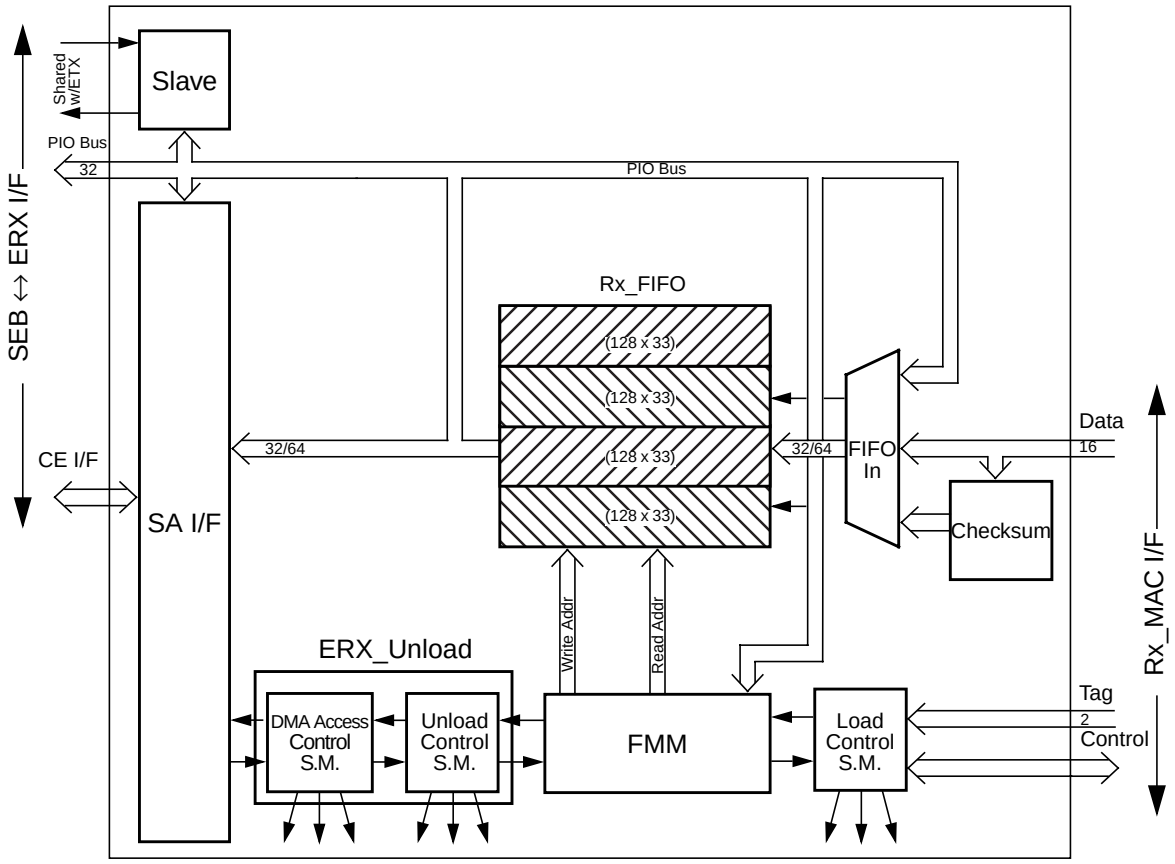

FIGURE 6-4 Receive DMA Channel

6.2.2.2 Interfaces and Data Paths

The Ethernet Channel contains seven defined interfaces:

Channel Engine Interface (CEI)

This exposed interface connects the Ethernet Channel to the PCI Bus Adaptor. The CEI is defined to be identical for both channels in PCIO, and it follows the DVMA and the Programmed IO protocol on the PCI Bus. The CEI data path is defined to be 32 or 64 bits wide in each direction, and it scales with the width of the PCI Bus.

SEB ↔ ETX Interface

This non-exposed interface connects the transmit DMA channel (ETX) to the Shared Ethernet Block (SEB). This interface implements the “DVMA Read” subset of the CEI. The data path is defined to be 32 or 64 bits wide, and it scales with the width of the PCI Bus.

SEB ↔ ERX Interface

This non-exposed interface connects the receive DMA channel (ERX) to the Shared Ethernet Block (SEB). This interface implements the “DVMA Write” subset of the CEI. The data path is defined to be 32 or 64 bits wide, and it scales with the width of the PCI Bus.

PIO Bus Interface

This non-exposed interface connects the ERX, the ETX and the MAC core to the SEB for Programmed IO cycle execution. This interface implements the Programmed I/O (”slave”) subset of the CEI. The data path is defined to be always 32 bits wide in each direction.

ETX ↔ TX_MAC Interface

This non-exposed interface connects the ETX to the TX_MAC for packet data transfer from the transmit FIFO to the network. The data path is defined to be 16 bits wide.

ERX ↔ RX_MAC Interface

This non-exposed interface connects the ERX to the RX_MAC for packet data transfer from the network to the receive FIFO. The data path is defined to be 16 bits wide.

Media Independent Interface (MII)

This exposed interface connects the Ethernet Channel to an external Ethernet transceiver. It conforms with the IEEE 802.3u defined MII. The MII data path is defined to be 4 bits wide in each direction. The MII management interface is a I-directional serial interface. The Ethernet Channel implements two management interfaces.

6.2.2.3 Clock Domains

The Ethernet Channel contains three completely asynchronous clock domains. FIGURE 6-5 shows the clock domain boundaries.

System Clock Domain

The bulk of the logic in the Ethernet Channel is driven off this clock. It is sourced by the System Bus and is defined to be in the range of 16.67 MHz through 33.33 MHz.

Transmit Clock Domain

This clock is used to drive the Transmit Protocol Engine in the MAC core. It is sourced by the MII and has the operating frequency of 2.5/25 MHz 100ppm. The 2.5/25 MHz version of this clock (tx_nclk) is used for byte-to-nibble conversion of the data stream to the MII and for synchronization of the asynchronous signals from the MII (CRS and COLL). The 1.25/12.5 MHz “divide-by-two” version of this clock (tx_bclk) is used for transmit protocol processing and state machine operation.

Receive Clock Domain

This clock is used to drive the Receive Protocol Engine in the MAC core. It is sourced by the MII and has the operating frequency of 2.5/25 MHz 100ppm. The 2.5/25 MHz version of this clock (rx_nclk) is used for strobing in the packet data from the MII and for nibble-to-byte conversion of the incoming data stream. The 1.25/12.5 MHz “divide-by-two” version of this clock (rx_bclk) is used for receive protocol processing and state machine operation.

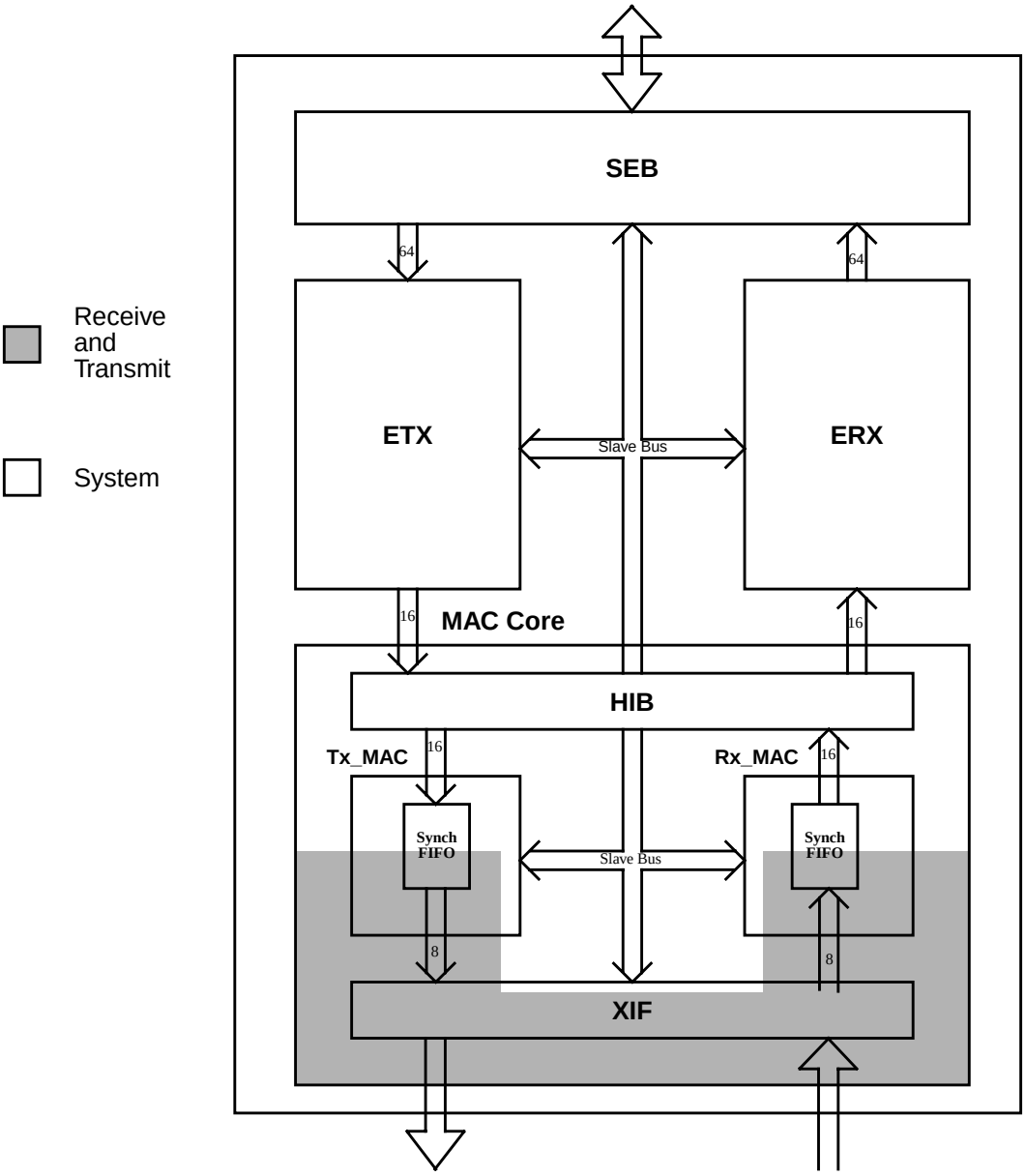

FIGURE 6-5 Ethernet Channel Clock Domains

6.2.3 Host Memory Data Management

The device driver maintains two data structures in the host memory: one for transmit and the other for receive packets. Both data structures are organized as "wrap-around descriptor rings." Each descriptor ring has a programmable number of descriptors (in the range of 16 through 256). Each descriptor has two entries (words): a control/status word and a pointer to a data buffer.

The interaction between the hardware and the software is managed via a semaphore (OWN) bit, that resides in the control/status portion of the descriptor. When the OWN bit is set to '1', the descriptor is "owned" by the hardware. If the OWN bit is cleared to '0', the descriptor is "owned" by the software. The owner of the descriptor is responsible for releasing the ownership when it can no longer use it. Once the ownership is released, the previous owner may no longer treat the descriptor contents as valid, since the new owner may overwrite it at any time.

6.2.3.1 Transmit Data Descriptor Ring

A transmit packet that is posted by an upper layer protocol to the device driver may reside in several data buffers (headers and data) which are scattered in the host memory. When the device driver posts the packet to the hardware, it allocates a descriptor for each buffer. The descriptor contains the necessary information about the buffer that the hardware needs for the packet transfer.

When the packet is ready for transmission, the descriptor(s) ownership is turned over to the hardware, and a Programmed IO command is issued to the transmit DMA channel to start the packet transfer from the host memory to the TxFIFO.

When the packet transfer has been completed, the transmit DMA channel turns over the descriptor ownership back to the driver and polls the next descriptor in the ring. If the descriptor is owned by the hardware, the next packet transfer begins. If not — the DMA channel "goes to sleep" until a new command is issued.

The size of the descriptor ring is programmable, and it can be varied in the range of 16–256 in increments of 16 descriptors: 16, 32, 48, ..., 240, 256.

6.2.3.2 Receive Free Buffer Descriptor Ring

For receive operation, the device driver requests a pool of free buffers from the OS. The buffers are posted to the hardware by allocating a descriptor for each buffer. The descriptor contains the necessary information about the buffer that the hardware needs for the packet transfer.

When a packet is ready to be transferred from the RxFIFO to the host memory, the receive DMA channel polls the next descriptor in the ring. If the hardware owns the descriptor (free buffer available), the packet transfer begins. During the first burst, the receive DMA engine will perform “header padding” of the packet by inserting a programmable number of “junk” words at the beginning of the packet.

When the packet transfer has been completed, the receive DMA channel updates the descriptor with status information about the received packet, and turns over the descriptor ownership back to the driver.

If a packet is ready to be transferred from the RxFIFO to the host memory, but the driver does not have any free buffers allocated to the hardware, the packet will be dropped into the “bit bucket”, and the DMA channel will try again when the next packet is ready to go.

The size of the descriptor ring is programmable and can assume the following values: 32, 64, 128, 256.

6.2.4 Local Memory Data Management

Each DMA channel contains its own dedicated on-chip local buffer of 2K bytes (fixed) in size. The local buffers are used for temporary storage of packets en route to or from the network, and are organized as wrap-around FIFOs.

In general, the local buffer organization and data structures are invisible to the software, except for diagnostic purposes.

Since the local buffers reside in the data path, their logical organization changes depending on the PCI Bus width. For a 32-bit PCI Bus, the FIFO organization is 512words × 33bits. For a 64-bit PCI Bus, the FIFOs are organized as 256words × 65bits. The “extra” bits (bit 33 or bit 65) along the word are used as end-of-packet delimiters (or “tags”). When a packet is stored in the local buffer, the tag will be cleared to ‘0’ for the entire data portion of the packet, except for the last word. The tag will be set to ‘1’ for the last data word of the packet and for the control/status word.

6.2.4.1 Transmit FIFO Data Structures

When a transmit packet is transferred from the host into the local memory, the first byte of the packet in the FIFO is always loaded to be word (or double-word) aligned. If the packet is composed of several data buffers, the data buffers are concatenated as a contiguous byte stream in the FIFO (”gather” function). The last byte of a packet can reside at any byte boundary, therefore the last data word of the packet is marked

by a tag. At the end of the packet a control word is appended, which is again marked by a tag bit. The control word indicates the last byte boundary for the packet.

6.2.4.2 Receive FIFO Data Structures

When a receive packet is transferred from the RX_MAC into the local memory, the half-word (16-bit) data stream is packed into words (or double-words), with the first byte of the packet starting at a programmable offset within the first word.

Even though the receive data structures' functionality does not require to tag the last data word of a packet, the hardware will do that to provide a more robust implementation.

At the end of the packet a status word is appended, which is again marked by a tag bit. This word provides status information about the received frame, which is either passed to the device driver or used for unloading the frame from the RxFIFO.

6.2.5 Theory of Operation and Data Flow

Following are detailed descriptions of the sequences of events that take place in each DMA channel during normal transmission and reception of packets to/from the network. Each DMA channel has two simultaneously active processes: one for loading the local FIFO and the other for unloading it. The loading process is trying to keep the corresponding FIFO "as full as possible", and the unloading process will try to keep it "as empty as possible".

6.2.5.1 Transmit Operation

▼ TxFIFO Load Process

1. An upper layer protocol posts a packet for transmission to the device driver. The packet may reside in one or more data buffers.
2. The device driver posts the packet to the hardware, by allocating a descriptor for each data buffer. The descriptor ownership is turned over to the hardware.
3. The device driver issues a PIO command to the Tx DMA engine: "transmit pending." At this time the Write Pointer and the Shadow Write Pointer are equal, and point to next available location in TxFIFO.
4. The transmit DMA engine fetches the next descriptor from the ring using an 8-byte DVMA burst read.

5. The Load Control state machine checks for space availability in the TxFIFO.

6. If there is not enough space in the FIFO to fit a DVMA burst (programmable size), the state machine waits for some space to free up.

7. Once there is enough space in the FIFO, the state machine executes a DVMA burst read. As data is being loaded into the FIFO, the Write Pointer increments while the Shadow Write Pointer remains fixed pointing to the beginning of the packet in the FIFO.

8. On its way to the FIFO the data passes through the Chaining module where byte alignment is performed. For the first buffer, the data is aligned to a word or double-word boundary in the FIFO. For subsequent buffers, “byte rotation” is performed, to form a contiguous data stream in the FIFO for each packet.

9. As the “chained” data stream “flies-by” on its way to the FIFO, the Checksum module monitors it, and the TCP checksum is computed starting from the “start_checksum_offset”. Also, the word indicated by “stuff_checksum_offset” is saved in a temporary holding register.

10. Steps 5 through 9 are repeated until the entire buffer is transferred from the host memory into the FIFO.

11. The transmit DMA engine turns over the descriptor ownership back to the device driver using a single-word DMA cycle to update the descriptor.

12. If the packet contains more than one data buffer, steps 4 through 11 are repeated until the entire packet has been transferred from the host memory into the FIFO.

13. If TCP checksum generation is disabled for the packet — skip to step 18.

14. The Write Pointer and the Shadow Write Pointer exchange their values.

15. The checksum result is loaded to the appropriate field in the holding register in the Checksum module.

16. The holding register contents are loaded into the FIFO at the “stuff_checksum_offset.”

17. The Write Pointer is loaded by the contents of the Shadow Write Pointer.

18. The last data word of the packet and the control word are loaded into the FIFO with the tag bits set.

19. The Shadow Write Pointer is loaded by the contents of the Write Pointer, with both of them pointing to the next available location in the FIFO.

20. An interrupt is generated to indicate a successful completion of the loading process.

21. The transmit DMA engine fetches the next descriptor from the ring. If the descriptor is owned by the hardware, steps 5 through 19 are repeated. If not — the DMA engine generates an interrupt (tx_all) and “goes to sleep” until it is “awakened” by a “transmit pending” command from the device driver.

▼ TxFIFO Unload Process

1. The TxFIFO unload process is kicked off when the amount of packet data in the FIFO has exceeded the programmable threshold, or the number of frames in the FIFO is greater or equal to one. Until then the requests for data from the TX_MAC are ignored. At this time the Read Pointer and the Shadow Read Pointer are equal, and point to the beginning of the next packet in the FIFO.
2. Once transmission is enabled, the Unload Control state machine reads one word of data out of the FIFO, unpacks the data into a 16-bit data stream, and transfers it to the TX_MAC in bursts of 8 bytes at a time over the ETX↔TX_MAC interface. As data is being unloaded from the FIFO, the Read Pointer increments while the Shadow Read Pointer remains fixed pointing to the beginning of the packet in the FIFO.
3. If the TxFIFO runs out of data during a packet transfer, the Unload Control state machine will wait until more data is accumulated in the FIFO.
4. If a normal collision occurs on the network, the frame will be re-transmitted using the retry mechanism on the ETX↔TX_MAC interface. The Shadow Read Pointer is loaded into the Read Pointer, and the frame transfer starts from the beginning.
5. When the last word of the frame is encountered (tag bit set to ‘1’), the state machine waits for the control word to be unloaded, and then transfers the last burst of data to the TX_MAC.
6. The Shadow Read Pointer is loaded with the contents of the Read Pointer.
7. When the TX_MAC is ready to transmit the next frame, it requests more data from the Unload Control state machine, and steps 1 through 6 are repeated.

6.2.5.2 Receive Operation

▼ RxFIFO Load Process

1. When a receive frame arrives from the network, the RX_MAC performs protocol processing on it and evaluates the initial receive criteria (address detection, etc.). If the frame does not pass this criteria, it is dropped into the “bit bucket.” At this time the Write Pointer and the Shadow Write Pointer are equal, and point to the next available location in the RxFIFO.

2. The RX_MAC waits until 8 bytes of data are accumulated in its internal FIFO, and issues a request for data transfer to the receive DMA engine.
3. The Load Control state machine checks for space availability in the RxFIFO.
4. If there is not enough space in the FIFO to fit an 8-byte burst from the RX_MAC, the state machine waits for some space to free up.
5. Once there is enough space in the FIFO, the state machine performs a burst transfer from the RX_MAC to the RxFIFO. As data is being loaded into the RxFIFO, the Write Pointer increments while the Shadow Write Pointer remains fixed pointing to the beginning of the packet in the FIFO.
6. As the data stream “flies-by” on its way to the FIFO, the Checksum module monitors it, and the frame checksum is computed on the entire MAC frame, starting from a 16-bit aligned programmable “start_checksum_offset”.
7. On its way to the FIFO the data is packed into words. For the first word, the first byte of the frame is aligned to a programmable offset. For subsequent words, “byte rotation” is performed, to form a contiguous data stream in the FIFO for each packet.
8. Steps 2 through 7 are repeated until the entire frame is transferred from the RX_MAC into the FIFO. The end of the frame is detected by the Load Control state machine when the status word is received from the RX_MAC.
9. The receive Load Control constructs a new status word, that contains the status word received from the RX_MAC and the computed TCP checksum.
10. The last word of the frame is loaded into the FIFO with the tag bit set to ‘1’.
11. The status word of the frame is appended to the end of the frame in the FIFO with the tag bit set to ‘1’.
12. The Shadow Write Pointer is loaded by the contents of the Write Pointer, with both of them pointing to the next available location in the FIFO.
13. Go to step 1.

▼ RxFIFO Unload Process

1. The RxFIFO unload process is kicked off when the amount of packet data in the FIFO has exceeded 128 bytes (fixed), or the number of frames in the FIFO is greater or equal to one.
2. Once the receive frame is ready to be transferred to the host memory, the receive DMA engine fetches the next descriptor from the ring using an 8-byte DVMA burst read.
3. The Unload Control state machine checks for data availability in the RxFIFO.

4. If the amount of packet data is less than one DVMA burst size (programmable), the state machine waits for more data to accumulate in the FIFO.
5. Once there is more than a burst-size amount of data in the FIFO, or an entire packet has been stored in the FIFO, the DMA Control state machine executes a DVMA burst write. For the first burst of a frame, the Unload Control state machine performs “header padding” by inserting a programmable number of “junk” words at the beginning of the burst.
6. Steps 3 through 5 are repeated until the entire frame is transferred into the host buffer.
7. When the last word of the frame is encountered during a DVMA burst, the Unload Control state machine constructs a descriptor status word.
8. The receive DMA engine turns over the descriptor ownership back to the device driver using a single-word DMA cycle to update the descriptor status field.
9. An interrupt is generated to indicate a successful completion of the unloading process.
10. Go to step 1.

6.2.6 Error Conditions and Recovery

There are two types of error conditions that can be encountered during the normal operation of the Ethernet Channel: fatal errors and non-fatal errors.

Fatal errors are errors that “should never occur.” They usually indicate a serious failure of the hardware or a serious programming error. When this type of error occurs, the recovery process is “non-graceful.” The corresponding DMA channel will “freeze”, and the software is expected to reset the channel after the appropriate actions were taken to correct the failure. Fatal error events are always reported to the software via an interrupt.

Non-fatal errors are errors that are “expected to occur” when certain conditions occur on the network or in the system. When this type of error occurs, a “graceful” recovery mechanism is provided via a combination of hardware and software, as described below. Non-fatal errors may or may not be reported to the software.

6.2.6.1 Fatal Errors

The error conditions described below can occur both in the transmit and in the receive DMA channels.

Master_Error_Ack

This error condition indicates that an error acknowledgment was detected by the DMA channel during a DVMA cycle.

Slave_Error_Ack

This error condition indicates that an error acknowledgment was generated by the DMA channel during a Programmed IO cycle. The hardware will generate an error acknowledgment if a Programmed IO cycle is executed with transfer size other than a “word transfer.”

Late_Error

This error condition indicates that a bus late data error was detected by the DMA channel during a DVMA cycle.

DMA_Read_Parity_Error

This error condition indicates that a parity error was detected by the DMA channel during a DVMA Read cycle.

Slave_Write_Parity_Error

This error condition indicates that a parity error was detected by the DMA channel during a Programmed IO Write cycle.

FIFO_Tag_Error

The data structures in the local FIFOs make use of tag bits for delimiting packet boundaries. The last data word and the control/status word of a frame are expected to have their tag bits set to ‘1’. If the Unload Control state machine does not see two consecutive tag bits set to ‘1’, a local memory failure is recognized, and the unloading process is aborted.

6.2.6.2 Non-fatal Errors

The error conditions described below can occur in the specified DMA channel only.

Tx_FIFO_Underrun

This error condition can occur only when the programmable threshold is used to enable transmission of the frame by the TX_MAC (the threshold value is less than the maximum frame size). If the available bandwidth on the PCI Bus dedicated to transmit DMA is less than the available throughput on the network, the TxFIFO may run out of data before the frame transmission has completed. The TX_MAC may become “starved” for data, and the frame transmission is aborted. The unloading of

the frame from the FIFO will continue until the entire frame is transferred to the TX_MAC, but the TX_MAC will drop the remainder of the frame into the “bit bucket.” The TX_MAC will generate an interrupt to the device driver to indicate the occurrence of this event.

Rx_Abort (early and late)

A receive frame can be aborted for various reasons at any time during the frame transfer from the network to the host memory. The intent of the provided abort mechanism is to utilize the available hardware resources efficiently, without incurring unnecessary performance penalties.

If an abort condition is detected before the frame transfer has begun from the RX_MAC into the Rx_FIFO (address detection criteria, short fragment, etc.) the RX_MAC drops the frame and the receive DMA channel never sees it.

If an abort condition occurred after the frame transfer from the RX_MAC into the Rx_FIFO has begun, but before at least 128 bytes of data were transferred from the RX_MAC to the RX_FIFO (long fragment, etc.), the Load Control state machine re-winds the Write Pointer to the Shadow Write Pointer and gets ready to receive the next frame. This way the FIFO locations that were occupied by the long fragment are re-used by the next frame.

If an abort condition is detected after at least 128 bytes of data were transferred from the RX_MAC to the RX_FIFO (very long fragment, crc error, code error on the media, etc.), the Load Control state machine sets the “abort” bit in the status word that is appended to the frame and gets ready to receive the next frame. When the aborted frame is unloaded from the RxFIFO, the Unload Control state machine detects the “abort” bit in the status word and reuses the current descriptor (host data buffer) for the next frame.

This error condition is not reported to the software, but the events causing it have their individual reporting mechanisms.

Rx_FIFO_Overflow

If the available bandwidth on the PCI Bus dedicated to receive DMA is less than the available throughput on the network, the RxFIFO may run out of space and not be able to receive any more data from the RX_MAC. This condition propagates to the RX_MAC, and when it runs out of space in its synchronization FIFO the frame is aborted using the rx_abort mechanism that was described above. The RX_MAC will continue to receive the frame from the network, but the remainder of the frame is dropped “on the floor.” The RX_MAC will generate an interrupt to the device driver to indicate the occurrence of this event.

Rx_Buffer_Not_Available

When a receive frame is ready to be transferred to the host memory, the DMA Control state machine fetches the next descriptor from the ring. If the descriptor is not owned by the hardware, the error condition is encountered. The unloading process unloads the frame from the RxFIFO and drops it “on the floor.” When the next frame in the FIFO is to be unloaded, the DMA Control state machine polls the descriptor again. An interrupt is generated to the device driver to indicate the occurrence of this event.

Rx_Buffer_Overflow

The unloading process transfers frames from the RxFIFO to data buffers in the host memory. If the size of a buffer in the host memory is smaller than the frame size, the buffer is filled up and the remainder of the frame is dropped “on the floor.” This error condition is not reported to the software via an interrupt. Instead, when the descriptor is returned to the device driver, an “overflow” status bit is set in the descriptor. Also, the “length” field in the descriptor specifies the actual size of the frame received.

6.3 Programmer’s Reference Guide

6.3.1 Overview

During normal operation, the software-to-hardware interaction is primarily performed via the host memory data structures, with a minimal command/status handshake (”less than one” interrupt per packet). Software intervention is required for initialization of the hardware after resetting the Channel, for network management, for error recovery and for diagnostic purposes. Local FIFOs’ data structures and most of the registers are invisible to the software, except for diagnostic purposes.

6.3.2 Host Memory Data Structures

The host memory data structures are organized as “wrap-around descriptor rings” of programmable size. The transmit and receive data structures are very similar, except for three major differences:

1. Descriptor layout

2. Number of descriptors per packet: one for receive, unlimited for transmit

3. Data buffer alignment restrictions: none for transmit, one for receive

Programming Restrictions: The pointers to Descriptor Ring Base Addresses must be 2K-byte aligned.

6.3.2.1 Transmit Data Structures

FIGURE 6-6 shows the transmit descriptor ring organization.

Table 6-1 Transmit Data Structures: Descriptor Layout – Control Word

Bits	Field Name	Description
[13:0]	Data Buffer Size	Indicates the number of data bytes in the buffer. All values are legal in a 16 KB range, including 0
[19:14]	Checksum Start Offset	Indicates the number of bytes from the first byte of the packet that should be skipped before the TCP checksum calculation begins. This field is only meaningful if the Checksum Enable bit is set to ‘1’
[27:20]	Checksum Stuff Offset	Indicates the byte number from the first byte of the packet that will contain the first byte of the computed TCP checksum. This field is only meaningful if the Checksum Enable bit is set to ‘1’
[28]	Checksum Enable	If set to ‘1’, the computed TCP checksum will be “stuffed” into the packet
[29]	End Of Packet	When set to ‘1’, indicates the last descriptor of a transmit packet
[30]	Start Of Packet	When set to ‘1’, indicates the first descriptor of a transmit packet
[31]	OWNership semaphore	To turn over ownership, the hardware clears this bit, and the software sets it.

Table 6-2 Transmit Data Structures: Descriptor Layout – Data Buffer Pointer

Bits	Field Name	Description
[31:0]	Data Buffer Pointer	This 32-bit pointer indicates the first data byte of the transmit buffer

Programming Restrictions: 1. If a packet occupies more than one descriptor, the software must turn over the ownership of the descriptors to the hardware “last-to-first”, in order to avoid race conditions.

2. If a packet resides in more than one buffer, the Checksum Enable, Checksum Stuff_Offset and Checksum_Start_Offset fields must have the same values in all the descriptors that were allocated to the packet.

3. The hardware implementation relies on the fact, that if a buffer starts at an “odd” byte boundary, the DMA state machine can “rewind” to the nearest burst boundary and execute a full DVMA burst Read.

6.3.2.2 Receive Data Structures

FIGURE 6-7 shows the receive descriptor ring organization.

Table 6-3 Receive Data Structures: Descriptor Layout – Status Word

Bits	Field Name	Description
[15:0]	TCP Checksum	This field contains the 16-bit TCP checksum that was calculated on the entire frame. It will be updated for every frame that was received from the network. The software has the choice of either making use of it, or ignoring it
[29:16]	Free_Buffer / Packet_Data Size	When the descriptor ownership is passed from the software to the hardware, this field contains the size of the free buffer that was allocated for the packet. When the descriptor ownership is passed from the hardware to the software, this field indicates the actual number of packet data bytes that were “dumped” into the buffer
[30]	Overflow	When a Rx_Buffer_Overflow condition occurs, this bit will be set to ‘1’ for the frame that could not fit into the allocated buffer
[31]	OWNership semaphore	To turn over ownership, the hardware clears this bit, and the software sets it

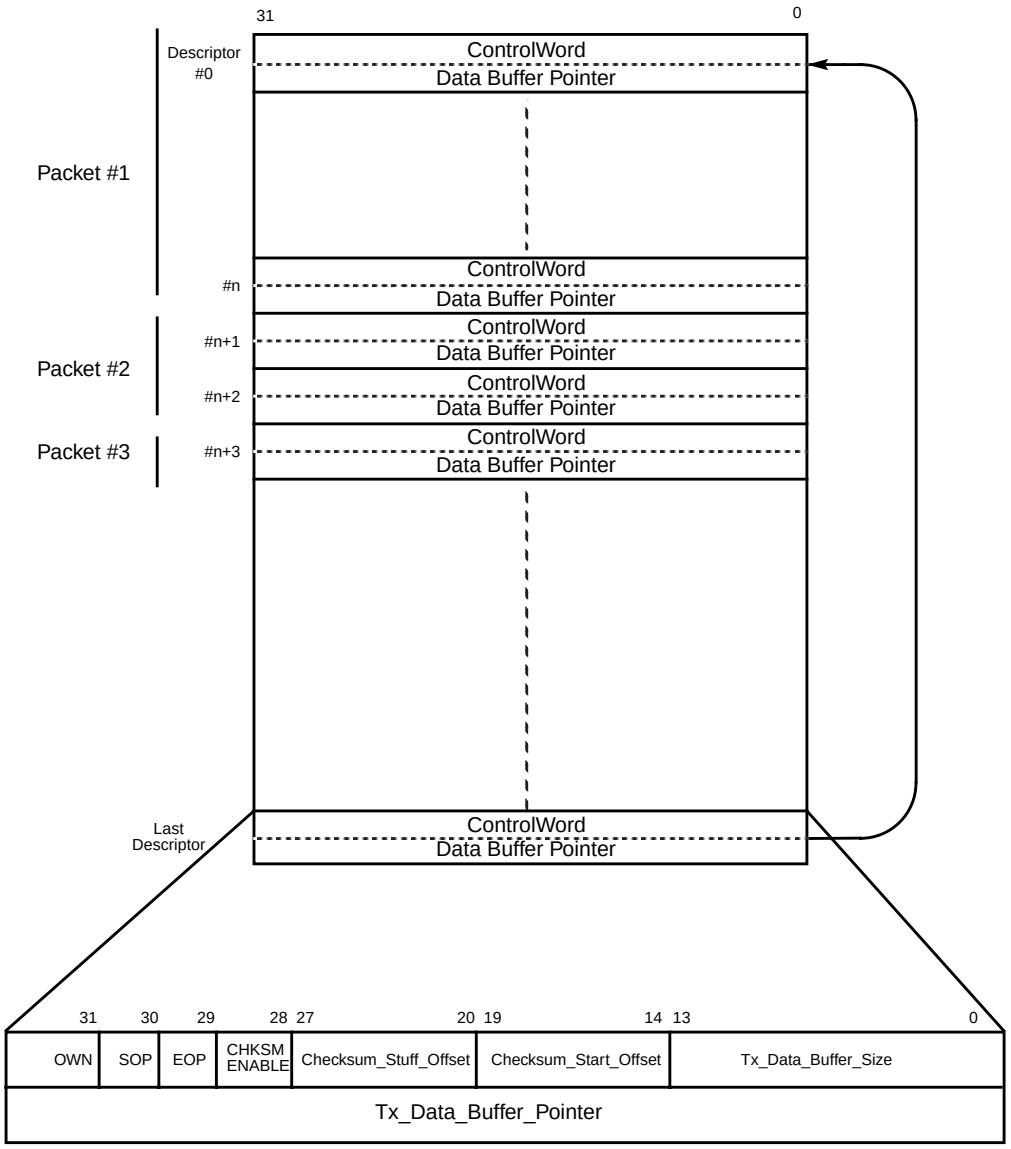

FIGURE 6-6 Transmit Host Data Structure

Table 6-4 Receive Data Structures: Descriptor Layout – Free Buffer Pointers

Bits	Field Name	Description
[31:2]	Free Buffer Pointer	This 29-bit pointer points to the beginning of the free buffer. The first byte of the actual packet data inside the buffer will always reside at a programmable offset from this location, but within a double-word range

Programming Restrictions: Free receive data buffers must be 64-byte aligned.

6.3.3 Local Memory Data Structures

The local memory data structures are organized as “wrap-around FIFOs” that can store an unlimited number of packets. The transmit and receive data structures are very similar, except for the format of the control/status word that is appended to the end of a packet and the alignment of the first byte of a packet when it is loaded into the FIFO. Also, the RxFIFO does not have a Shadow Read Pointer. The logical organization of the FIFOs changes depending on the SBus configuration. For a 32-bit SBus, the FIFO organization is 512words × 33bits. For a 64-bit SBus, the FIFOs are organized as 256words × 65bits. The “512words × 33bits” configuration makes use of both the Tag_0 and the Tag_1 bits in the FIFO, while the “256words × 65bits” configuration uses only the Tag_0 bit.

On the diagrams shown below, FIGURE 6-8 and FIGURE 6-9, frames #1 and #2 represent a “512words × 33bits” configuration, and frame #n represents a “256words × 65bits” configuration. In reality, of course, only one configuration is used at a given time. The configuration is selected by programming the Extended Transfer Mode bit in Global Config. Register. The amount of “junk” at the beginning of a frame in the RxFIFO is determined by the “first_byte_offset” field in the ERX Configuration Register.

The software has the capability to read and write the FIFOs (including Tags) at any time, using programmed IO instructions. This feature should be used for diagnostic purposes only. During normal operation, the FIFOs are “invisible” to the software.

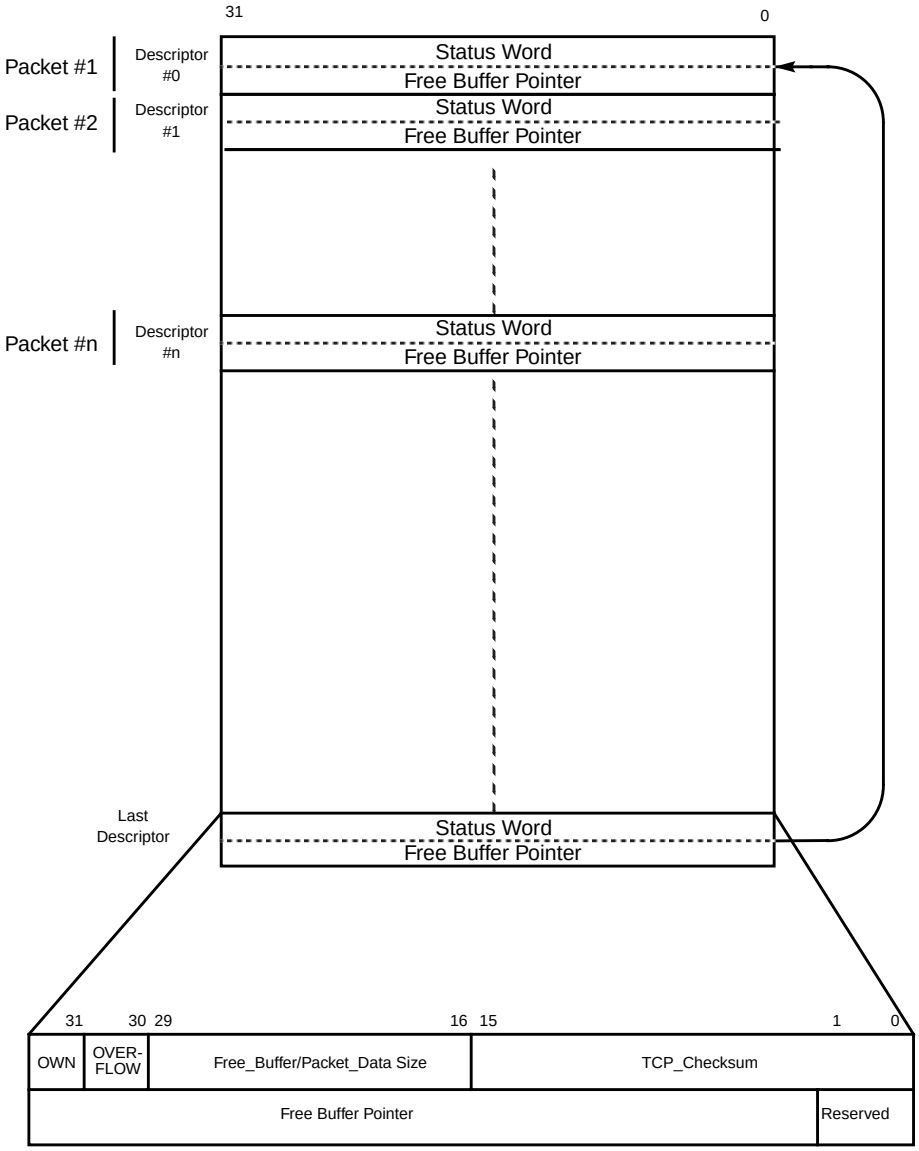

FIGURE 6-7 Receive Host Data Structure

6.3.3.1 TxFIFO Data Structures

FIGURE 6-8 shows the organization of the TxFIFO. The first byte of the frame is always loaded to be word or double-word aligned.

Table 6-5 TxFIFO Data Structures: Control Word Layout

Bits	Field Name	Description
[2:0]	Last Byte Boundary	This field indicates the offset of the last byte of the packet within the last data word (or double-word, depending on the configuration) in the FIFO.

6.3.3.2 RxFIFO Data Structures

FIGURE 6-9 shows the organization of the RxFIFO. The first byte of the frame is always loaded at a programmable offset within the first word or double-word.

Table 6-6 RxFIFO Data Structures: Status Word Layout

Bits	Field Name	Description
[15:0]	Frame Checksum	This field contains the 16-bit TCP checksum for the frame, as computed during the frame transfer from the RX_MAC to the RxFIFO
[26:16]	Frame Size	This field indicates the size of the frame in bytes as calculated by the RX_MAC
[30]	Reserved	
[31]	Receive Abort	This bit communicates the occurrence of a “late abort” event to the Unload Control state machine. The frame should be dropped and the descriptor re-used for the next frame

6.3.4 Other User Accessible Resources

Besides the host and local memory data structures, the hardware provides a Programmed IO path to a variety of hardware resources for initialization, error recovery, diagnostics and network management. From the software perspective all the programmable resources should be treated as 32-bit entities. If not all 32 bits are

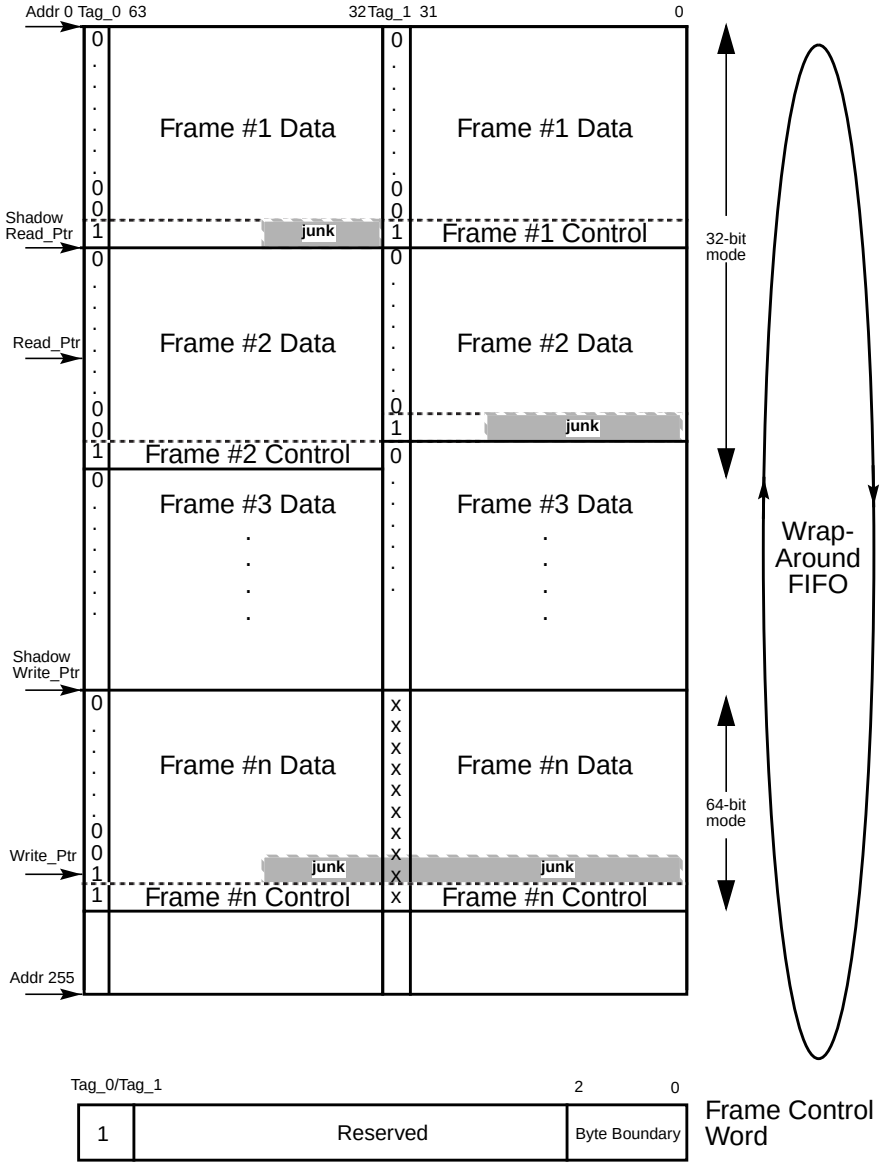

FIGURE 6-8 TxFIFO Organization

used in a register, the “unused” bits are grouped as the most significant bits of the word. Register fields that are “not used” are ignored during a PIO write, and return ‘0’s during a PIO read.

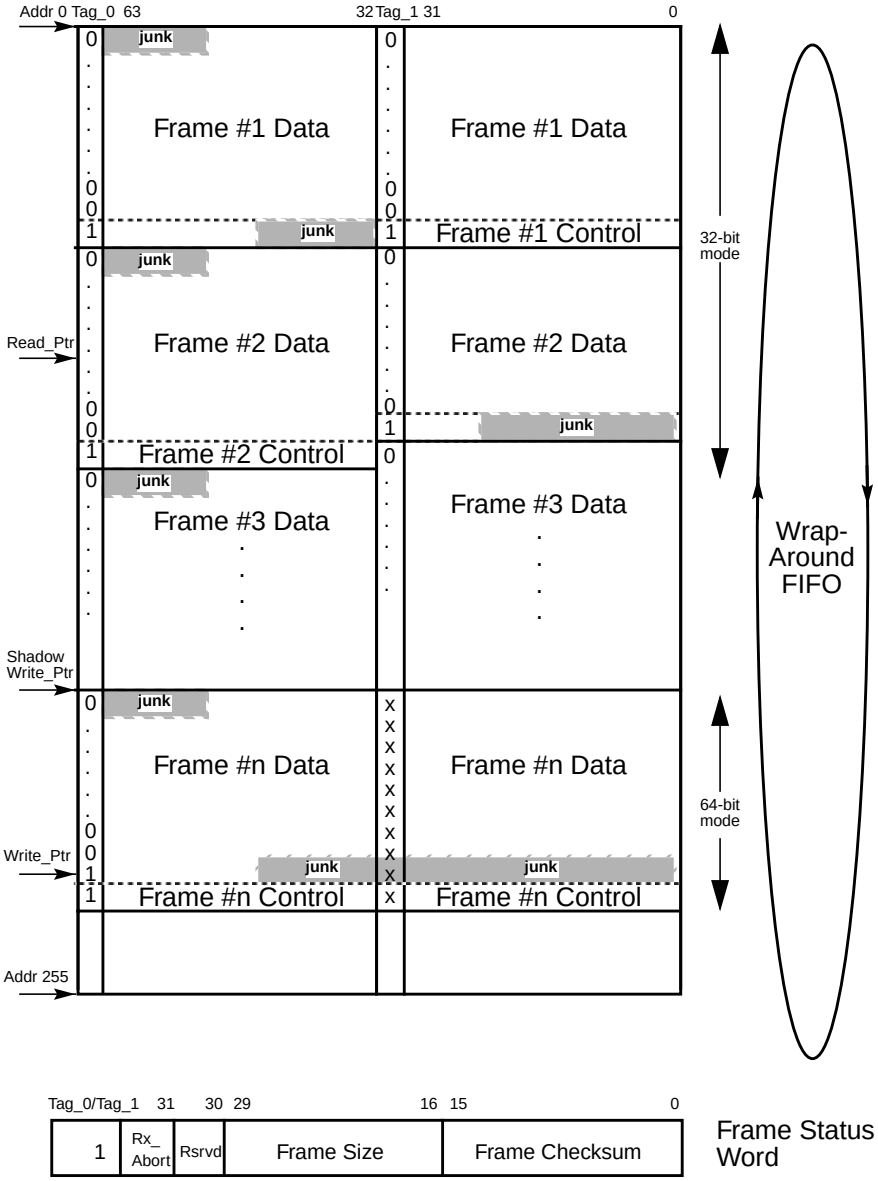

FIGURE 6-9 RxFIFO Organization

The description of these resources is grouped by functionality and not necessarily by their physical location. The default value for all the registers/counters is 0x00000000, unless specified otherwise.

6.3.4.1 SEB Programmable Resources

Software Reset Register (RW-AC)

This 2-bit register is used to perform an Individual Software Reset to the ETX or ERX modules (when the corresponding bit is set), or a Global Software Reset to the entire Ethernet Channel (when both bits are set). These bits can be set to '1' using a Programmed IO write to the defined address. They become "self-cleared" after the corresponding reset command has been executed.

TABLE 6-7 Software ResetRegister

Bits	Field Name	Description
[0]	ETX Software Reset	
[1]	ERX Software Reset	
[31:2]	Reserved	

Programming Restrictions: To ensure proper operation of the hardware after a Software Reset (Individual or Global), this register must be polled by the software. When both bits read back as 0's, the software is allowed to continue to program the hardware.

Global Configuration Register (RW)

This 5-bit register is used to determine the system-related parameters that control the operation of the DMA channels.

TABLE 6-8 Global Configuration Register

Bits	Field Name	Description
[1:0]	Burst_Size	This field determines the size of the host bus bursts that the DMA channels will execute: 00 — 16-byte burst 01 — 32-byte burst 10 — 64-byte burst 11 — Reserved
[2]	Extended_Transfer_Mode	When set to '1', 64-bit CEI and PCI Bus DVMA transactions will be performed. If cleared to '0', a 32-bit CEI/PCI Bus is assumed
[3]	Parity_Enable	When set to '1', parity checking is performed for DVMA read and PIO write cycles
[27:4]	Reserved	
[31:28]	Ethernet Channel ID	This field identifies the version number of the Ethernet Channel. Current version # is 0000.

Global Interrupt Mask Register (RW)

This 32-bit register is used to determine which status events will cause an interrupt. If a mask bit is cleared to '0', the corresponding event causes an interrupt signal to be generated on the PCI Bus. The layout of this register corresponds bit-by-bit to the layout of the Status Register, with the exception of bit [23]. The MIF Interrupt is not maskable here, and should be masked at the source of the interrupt in the MIF (see Section 6.3.4.5).

Default: 0xFF7FFFFF.

Global Status Register (R-AC)

This 32-bit register is used to communicate to the software events that were detected by the hardware. If a status bit is set to ‘1’, it indicates that the corresponding event has occurred. All the bits are automatically cleared to ‘0’ when the Status Register is read by the software, with the exception of bit [23]. The MIF Status Bit will be cleared after the MIF Status Register is read.

TABLE 6-9 Global Status Register

Bits	Field Name	Description
[0]	Frame_Received	A frame transfer from the RX_MAC to the RxFIFO has been completed
[1]	Rx_Frame_Counter _Expired	The Rx_Frame_Counter rolled over from FFFF to 0000
[2]	Alignment_Error _Counter_Expired	The Alignment_Error_Counter rolled over from FF to 00
[3]	CRC_Error_Counter _Expired	The CRC_Error_Counter rolled over from FF to 00
[4]	Length_Error_Counter_ Expired	The Length_Error_Counter rolled over from FF to 00
[5]	RxFIFO_Overflow	The synch. FIFO in the RX_MAC has experienced an overflow. A receive frame was dropped by the RX_MAC
[6]	Code_Violation _Counter_Expired	The Code_Violation_Counter rolled over from FF to 00
[7]	SQE_Test_Error	A Signal Quality Error was detected in the XIF.
[8]	Frame_Transmitted	The TX_MAC has successfully transmitted a frame on the medium
[9]	TxFIFO_Underrun	The TX_MAC has experienced an underrun in the synch. FIFO due to “data starvation” caused by transmit DMA
[10]	Max_Packet_Size _Error	The TX_MAC attempted to transmit a frame that exceeds the maximum size allowed
[11]	Normal_Collision _Counter_Expired	The Normal_Collision_Counter rolled over from FFFF to 0000
[12]	Excessive_Collision _Counter_Expired	The Excessive_Collision_Counter rolled over from FF to 00
[13]	Late_Collision_Counter _Expired	The Late_Collision_Counter rolled over from FF to 00

TABLE 6-9 Global Status Register *(Continued)*

Bits	Field Name	Description
[14]	First_Collision _Counter_Expired	The First_Collision_Counter rolled over from FFFF to 0000
[15]	Defer_Timer_Expired	The Defer_Timer rolled over from FFFF to 0000
[16]	Rx_Done	A frame transfer from RX_FIFO to the host memory has been completed
[17]	Rx_Buffer_Not _Available	The receive DMA engine tried to transfer a receive frame from the RxFIFO to the host memory, but did not find any descriptors that were available. The frame was dropped by the DMA engine
[18]	Rx_Master_Err_Ack	An Error Ack occurred during a receive DMA cycle
[19]	Rx_Late_Err	A Late Error occurred during a receive DMA cycle
[20]	Rx_DMA_Par_Err	A Parity Error was detected during a receive DMA read cycle (descriptor access)
[21]	Rx_Tag_Err	The Receive Unload Control state machine did not see two consecutive tag bits set
[22]	EOP_Error	The Transmit Load Control detected a descriptor with the OWN bit cleared, before the last descriptor of the current frame (EOP = 1) has been processed.
[23]	MIF_Interrupt	The Status Register in the MIF has at least one unmasked interrupt set.
[24]	Tx_Done	A frame transfer from the host memory to the TxFIFO has completed
[25]	Tx_All	The transmit DMA has transferred to the TxFIFO all the frames that have been posted to it by the software. There are no transmit descriptors that are currently owned by the hardware
[26]	Tx_Master_Err_Ack	An Error Ack occurred during a transmit DMA cycle
[27]	Tx_Late_Err	A Late Error occurred during a transmit DMA cycle
[28]	Tx_DMA_Par_Err	A Parity Error was detected during a transmit DMA read cycle

TABLE 6-9 Global Status Register *(Continued)*

Bits	Field Name	Description
[29]	Tx_Tag_Err	The Transmit Unload Control state machine did not see two consecutive tag bits set
[30]	Slave_Err_Ack	An Error Ack was generated by the hardware during a PIO cycle to the Ethernet Channel area. This is an indication that the PIO cycle was executed with sb_size other than a “word transfer”
[31]	Slave_Par_Err	A Parity Error was detected during a PIO write cycle to the Ethernet Channel area

6.3.4.2 ETX Programmable Resources

Transmit Pending Command (RW)

This 1-bit command must be issued by the software for every packet that the driver posts to the hardware. The bit is set to ‘1’ using a Programmed IO write to the defined address. This bit becomes “self-cleared” after the command has been executed. This command is used as a “wake up” signal to the transmit DMA engine.

ETX Configuration Register (RW)

This 10-bit register determines the ETX-specific parameters that control the operation of the transmit DMA channel.

TABLE 6-10 ETX Configuration Register

Bits	Field Name	Description
[0]	Tx_DMA_Enable	When set to '1', the DMA operation of the channel is enabled. The Load Control state machine will respond to the next "tx_pending" command. When cleared to '0', the DMA operation of the channel will cease as soon as the transfer of the current data buffer has been completed
[9:1]	Tx_FIFO_Threshold	This field determines the number of packet data words that will be loaded into the TxFIFO before the frame transmission by the TX_MAC is enabled. The maximum allowable threshold is '1BF'. If the desire is to buffer an entire standard Ethernet frame before transmission is enabled, this field has to be programmed to a value greater than '1BF'.
[10]	Paced_Mode	When set to '1', the Tx_All interrupt (bit 25 in the Global Status Register) will become set only after the TxFIFO becomes empty. If cleared to '0', the Tx_All interrupt will function as described in Section 6.3.4.1.

Default: 0x3FE.

Transmit Descriptor Pointer (RW)

This 29-bit register points to the next descriptor in the ring. The 21 most significant bits are used as the base address for the descriptor ring, while the 8 least significant bits are used as a displacement for the current descriptor.

Programming Restrictions: The Transmit Descriptor Pointer must be initialized to a 2 Kbyte-aligned value after power-on or software reset.

Transmit Descriptor Ring Size (RW)

This 4-bit register determines the number of descriptor entries in the ring. The number of entries can vary from 16 through 256 in increments of 16.

Default: 0xF; 256 descriptor entries.

Transmit Data Buffer Base Address (RO)

This 32-bit register points to the beginning of the transmit data buffer in the host memory. It is loaded by the DMA state machine during the descriptor fetch phase. This register is used to generate the DVMA burst address by adding to it the Data Buffer Displacement.

Transmit Data Buffer Displacement (RO)

This 10-bit counter keeps track of the next DVMA read burst address. It is used as a displacement for the Data Buffer Base Address. The counter increments by 1, 2 or 4 (depending on the burst size) after a DVMA read burst cycle has been executed by the transmit DMA engine. The counter is cleared when the Data Buffer Base Address is loaded by the DMA state machine. This register is used to generate the DVMA burst address by adding it to the Buffer Base Address.

Transmit Data Pointer (RO)

This 32-bit register points to the next DVMA read burst address. Its contents is the sum of the Transmit Data Buffer Base Address and the Transmit Data Buffer Displacement.

TxFIFO Packet Counter (RW)

This 8-bit up/down counter keeps track of the number of frames that currently reside in the TxFIFO. The counter increments when a frame is loaded into the FIFO, and decrements when a frame has been transferred to the TX_MAC. This counter is used to enable frame transfer from the TxFIFO to the TX_MAC.

TxFIFO Write Pointer (RW)

This 9-bit loadable counter points to the next location in the FIFO that will be loaded with PCI Bus data, the checksum or the frame control word. The counter increments by 1 or 2 (depending on PCI Bus configuration) after a word (or double-word) was loaded into the FIFO. The counter is loaded with the contents of Shadow Write Pointer, plus the appropriate offset, when the checksum is “stuffed” into the frame. This counter is used to generate the “write” address for the TxFIFO memory core.

TxFIFO Shadow Write Pointer (RW)

This 9-bit register points to the first byte of the packet that is either currently being loaded or is about to be loaded into the FIFO. The register is loaded with the contents of the Write Pointer after the packet transfer from the PCI Bus to the FIFO has been completed. When the Write Pointer is used to “stuff” the checksum into the frame, this register serves as a temporary hold register for the Write Pointer.

TxFIFO Read Pointer (RW)

This 9-bit loadable counter points to the next location in the FIFO that will be read from to retrieve packet data that is transferred to the TX_MAC. The counter increments by 1 or 2 (depending on PCI Bus configuration) after a word (or double-word) was read from the FIFO. The counter is loaded with the contents of the Shadow Read Pointer, when a “retry” occurs due to a collision on the network. This counter is used to generate the “read” address for the TxFIFO memory core.

TxFIFO Shadow Read Pointer (RW)

This 9-bit register points to the first byte of the packet that is either currently being unloaded or is about to be unloaded from the TxFIFO. The register is loaded with the contents of the Read Pointer after the packet transfer from the FIFO to the TX_MAC has been completed. This register is used to “rewind” the Read Pointer for frame re-transmission due to a collision on the network.

ETX State Machine Register (RO)

This 23-bit register provides the current state for all the state machines in ETX.

TABLE 6-11 ETX State Machine Register

Bits	Field Name	Description
[4:0]		Checksum State Machine state
[11:5]		Chaining State Machine state
[16:12]		Unload Control State Machine state
[22:17]		Load Control State Machine state.

TxFIFO (RW)

For diagnostic purposes a PIO path has been provided into the TxFIFO. When using PIOs, the configuration of the TxFIFO will be 512x33bits. In order to be able to access all the bits in the memory core, the address space of the TxFIFO has been doubled and split into two “apertures” as follows:

- Writing to the Lower Aperture will load 32 bits of data and clear the tag bit to ‘0’ at the addressed location
- Writing to the Higher Aperture will load 32 bits of data and set the tag bit to ‘1’ at the addressed location
- Reading from the Lower Aperture will return 32 bits of data from the addressed location
- Reading from the Higher Aperture will return the tag bit from the addressed location on data line [0]

Programming Restrictions: The Tx_FIFO should never be accessed using PIOs during normal operation.

6.3.4.3 ERX Programmable Resources

ERX Configuration Register (RW)

This 23-bit register determines the ERX-specific parameters that control the operation of the receive DMA channel.

TABLE 6-12 ERX Configuration Register

Bits	Field Name	Description
[0]	Rx_DMA_Enable	When set to ‘1’, the DMA operation of the channel is enabled. The Load Control state machine will start responding to RX_MAC requests for data transfer. When cleared to ‘0’, the DMA operation of the channel will cease as soon as the transfer of the current frame has been completed.
[2:1]	Reserved	
[5:3]	First_Byte_offset	This field determines the offset of the first data byte of the packet within the first double-word of packet data in the RxFIFO and in the host data buffer.
[8:6]	Reserved	

TABLE 6-12 ERX Configuration Register *(Continued)*

Bits	Field Name	Description
[10:9]	Desc_Ring_Size	This field determines the number of descriptor entries in the ring. These bits are encoded as follows: 00: 32 entries 01: 64 entries 10: 128 entries 11: 256 entries
[15:11]	Reserved	
[22:16]	Checksum_Start_Offset	Checksum_Start_Offset. Indicates the number of half-words from the first byte of the packet that should be skipped before the TCP checksum calculation begins

Receive Descriptor Pointer (RW)

This 29-bit register points to the next descriptor in the ring. The 21 most significant bits are used as the base address for the descriptor ring, while the 8 least significant bits are used as a displacement for the current descriptor.

Programming Restrictions: The Receive Descriptor Pointer must be initialized to a 2KByte-aligned value after power-on or software reset.

Receive Data Buffer Pointer (RO)

This 28-bit loadable counter keeps track of the next DVMA write burst address. The counter increments by 1, 2 or 4 (depending on the burst size) after a DVMA write burst cycle has been executed by the receive DMA engine. The counter is loaded with the free_buffer_pointer during the descriptor fetch phase. This counter is used to generate the DVMA write burst address.

RxFIFO Packet Counter (RW)

This 8-bit up/down counter keeps track of the number of frames that currently reside in the RxFIFO. The counter increments when a frame is loaded into the FIFO, and decrements when a frame has been transferred to the host memory. This counter is used to enable a frame transfer to the host memory.

RxFIFO Write Pointer (RW)

This 9-bit loadable counter points to the next location in the RxFIFO that will be loaded with data from the RX_MAC. The counter increments by 1 or 2 (depending on PCI Bus configuration) after a word (or double-word) was loaded into the FIFO. The counter is loaded with the contents of Shadow Write Pointer, when an “early receive abort” needs to be performed. This counter generates the “write” address for the RxFIFO memory core.

RxFIFO Shadow Write Pointer (RW)

This 9-bit register points to the first word of the packet that is either currently being loaded or is about to be loaded into the FIFO. The register is loaded with the contents of the Write Pointer after the packet transfer from the RX_MAC to the FIFO has been completed. This register is used to perform an “early receive abort.”

RxFIFO Read Pointer (RW)

This 9-bit loadable counter points to the next location in the RxFIFO that will be read from to retrieve packet data that is transferred to the host memory. The counter increments by 1 or 2 after a word (or double-word) was read from the FIFO. This counter generates the “read” address for the RxFIFO memory core.

ERX State Machine Register (RO)

This 32-bit register provides the current state for all the state machines in ERX.

TABLE 6-13 ERX State Machine Register

Bits	Field Name	Description
[4:0]		Load Control State Machine state
[6:5]		FIFO Pointer state
[9:7]		Checksum State Machine state
[15:10]		Reserved
[19:16]		Data State Machine state
[23:20]		Descriptor State Machine state
[25:24]		ERX Memdone Counter state
[31:26]		Reserved

Default: 0x0.

RxFIFO (RW)

For diagnostic purposes a PIO path has been provided into the RxFIFO. When using PIOs, the configuration of the RxFIFO will be 512x33bits. In order to be able to access all the bits in the memory core, the address space of the RxFIFO has been doubled and split into two “apertures” as follows:

- Writing to the Lower Aperture will load 32 bits of data and clear the tag bit to ‘0’ at the addressed location
- Writing to the Higher Aperture will load 32 bits of data and set the tag bit to ‘1’ at the addressed location
- Reading from the Lower Aperture will return 32 bits of data from the addressed location
- Reading from the Higher Aperture will return the tag bit from the addressed location on data line [0]

Programming Restrictions: The Rx_FIFO should never be accessed using PIOs during normal operation.

6.3.4.4 MAC Programmable Resources

XIF Programmable Resources

XIF Configuration Register (RW)

This 10-bit register determines the parameters that control the operation of the transceiver interface.

TABLE 6-14 XIF Configuration Register

Bits	Field Name	Description
[0]	Tx_Output_Enable	When set to ‘1’, this bit enables the output drivers on the MII transmit bus
[1]	Loopback	This mode of operation implements the internal loopback for the Ethernet Channel. The entire channel is driven off the system clock, the MII transmit bus is looped back to the MII receive bus, and the MII Tx_En signal is looped back to the MII Rx_Dv input

TABLE 6-14 XIF Configuration Register *(Continued)*

Bits	Field Name	Description
[2]	MII_Loopback	This mode of operation supports the external loopback for the Ethernet Channel. The entire channel is driven off the system clock. An external loopback connector should be used to loop back the MII transmit bus to the MII receive bus, and the MII Tx_En signal to the MII Rx_Dv input
[3]	MII_Buffer_Enable	This bit has been provided to control an external tri-state buffer that may reside on the MII receive data bus
[4]	Rev. 2.1: SQE_Test_Enable	When set to '1', this bit enables the Signal Quality Error Test as defined in Chapter 14 of IEEE 802.3. This feature is applicable only if a 10Base-T transceiver is connected to the MII, that implements this function
	Rev. 2.2: LANCE_Mode	When set to '1', this bit enables the programmable extension of the Rx-to-Tx IPG. In this mode, the TxMAC will defer during IPG0 and IPG1 when timing the Rx-to-Tx IPG, and will not defer during IPG2. When cleared to '0', the Tx_MAC will ignore IPG0, defer during IPG1 when timing the Rx-to-Tx IPG, and will not defer during IPG2.
[9:5]	Rev. 2.1: SQE_Test_Window	This field defines the "time window" during which the MII COL signal should become asserted, after the completion of the last transmission. This field is only meaningful if the SQE_Test_Enable bit is set to '1'
	Rev. 2.2: IPG0	This field defines the value of InterPacketGap0. This field is valid only if the LANCE_Mode is enabled, and ignored otherwise. The time interval specified in this register is in units of media nibble time.

Default: 0x140.

Programming Restrictions: To ensure proper operation of the hardware, when a loopback configuration is entered or exited, a Global Initialization Sequence should be performed.

TX_MAC Software Reset Command (RW)

This 1-bit command performs a software reset to the logic in the TX_MAC. The bit is set to ‘1’ when a Programmed IO write is performed to the defined address. This bit becomes “self-cleared” after the command has been executed.

TX_MAC Configuration Register (RW)

This 11-bit register controls the operation of the TX_MAC.

TABLE 6-15 TX_MAC Configuration Register

Bits	Field Name	Description
[0]	Tx_MAC_Enable	When set to ‘1’, the TX_MAC will start requesting packet data from the ETX, and the transmit Ethernet protocol execution will begin. When cleared to ‘0’, it will force the TX_MAC state machines to either remain in the idle state, or to transition to the idle state and stay there at the completion of an ongoing packet transmission
[4:1]	Reserved	
[5]	Slow_Down	When set to ‘1’, this bit will cause the TX_MAC to check for carrier sense before **every** transmission on the medium, and for the **entire duration of the IPG**. For normal operation this bit should be cleared to ‘0’
[6]	Ignore_Collision	When set to ‘1’, this bit will cause the TX_MAC to ignore collisions on the medium. For normal operation this bit should be cleared to ‘0’
[7]	No_FCS	When set to ‘1’, this bit will cause the TX_MAC not to generate the CRC for the transmitted frame. For normal operation this bit should be cleared to ‘0’
[8]	No_Backoff	When this bit is set to ‘1’, the backoff algorithm in the Protocol Engine is disabled. The TX_MAC will not back off after a transmission attempt that collided on the medium. Effectively the random number chosen by the backoff algorithm is fixed to ‘0’. For normal operation this bit should be cleared to ‘0’

TABLE 6-15 TX_MAC Configuration Register *(Continued)*

Bits	Field Name	Description
[9]	Full_Duplex	When this bit is set to '1', the CSMA/CD protocol is modified such that the TX_MAC will never "give up" on a frame transmission. In effect no limit will exist on transmission attempts. If the backoff algorithm reaches the attempts_limit, it will clear the attempts_counter and continue trying to transmit the frame until it is successfully transmitted on the medium. For normal operation it is recommended that this bit is set to '1'
[10]	Never_Give_Up	When this bit is set to '1', the CSMA/CD protocol is modified such that the TX_MAC will never "give up" on a frame transmission. In effect no limit will exist on transmission attempts. If the backoff algorithm reaches the attempts_limit, it will clear the attempts_counter and continue trying to transmit the frame until it is successfully transmitted on the medium. For normal operation it is recommended that this bit is set to '1'

Programming Restrictions: To ensure proper operation of the TX_MAC, the TX_MAC_En bit must always be cleared to '0' and a delay imposed before a PIO write to any of the other bits in the TX_MAC Configuration register or any of the MAC parameters registers is performed. The MAC parameters' registers are: IPG1, IPG2, AttemptLimit, SlotTime, PA_Size, PA_Pattern, SFD_Pattern, JamSize, TxMinFrameSize and TxMaxFrameSize.

The amount of delay required will depend on the time required to transmit a maximum size frame, and is thus dependent on the value programmed into the TxMaxFrameSize register and the data rate on the medium. For a standard 1518-byte frame on a 100Mbps network the delay would be 125msec. To avoid the requirement for a variable time delay, the TX_MAC_En bit may be polled, and when this bit reads back as a '0', all the registers mentioned above may be written, including all the other bits in the Configuration register.

InterPacketGap1 Register (RW)

This 8-bit register defines the first 2/3 portion of the InterPacketGap, which is timed by the TX_MAC before each frame's transmission is initiated. For back-to-back transmissions, this value is added to the value in the InterPacketGap2 register, and during the entire period the CarrierSense input signal is ignored by the TX_MAC. For a reception followed by a transmission, the TX_MAC will monitor the CarrierSense input signal during the time interval specified in this register and will respond to it, but will ignore it during the time interval specified in the InterPacketGap2 register. The time interval specified in this register is in units of media byte time.

Default: 0x08.

InterPacketGap2 Register (RW)

This 8-bit register defines the second 1/3 portion of the InterPacketGap parameter.

Default: 0x04.

AttemptLimit Register (RW)

This 8-bit register specifies the number of attempts that the TX_MAC will make to transmit a frame, before giving up on the transmission

Default: 0x10.

SlotTime Register (RW)

This 8-bit register specifies the slot time parameter in units of media byte time. This parameter defines the physical span of the network.

Default: 0x40.

PA Size Register (RW)

This 8-bit register specifies the number of PreAmble bytes that will be transmitted at the beginning of each frame. The register must be programmed with a value of 2 or greater.

Default: 0x07.

PA Pattern Register (RW)

This 8-bit register specifies the bit pattern of the PreAmble bytes that are transmitted at the beginning of each frame. The most significant bit of this register is transmitted and received first.

Default: 0xAA.

SFD Pattern Register (RW)

This 8-bit register specifies the bit pattern of the Start of Frame Delimiter bytes that are transmitted at the beginning of each frame, after the preamble. The most significant bit of this register will be transmitted and received first.

Default: 0xAB.

JamSize Register (RW)

This 8-bit register specifies the number of bytes to be transmitted by the TX_MAC after detecting a collision on the media.

Default: 0x04.

TxMinFrameSize Register (RW)

This 8-bit register specifies the minimum number of bytes that the TX_MAC will transmit for any frame on the media.

Default: 0x40.

TxMaxFrameSize Register (RW)

This 16-bit register specifies the maximum number of bytes that the TX_MAC will transmit for any frame on the media.

Default: 0x05EE.

PeakAttempts Register (R-AC)

This 8-bit register indicates the highest number of collisions per successfully transmitted frame, that have occurred since this register was last read. The maximum value this register can attain corresponds to the value in the AttemptLimit register minus one. This register will automatically be cleared to ‘0’ after it is read.

Defer Timer (RW)

This 16-bit loadable timer increments when the TX_MAC is deferring to traffic on the network while it is attempting to transmit a frame. The time base for the timer is the media byte clock divided by 256. Thus, on a 10Mbps network the timer ticks are 200 *m*sec., and on a 100Mbps network the timer ticks are 20*m*sec.

Normal Collision Counter (RW)

This 16-bit loadable counter increments for every frame transmission attempt that experiences a collision.

First Successful Collision Counter (RW)

This 16-bit loadable counter increments for every frame transmission that collided on the first attempt, but succeeded on the second attempt.

Excessive Collision Counter (RW)

This 8-bit loadable counter increments for every transmit frame that has exceeded the AttemptLimit. It indicates the number of frames that the TX_MAC has given up transmitting due to excessive amount of traffic on the network.

Late Collision Counter (RW)

This 8-bit loadable counter increments for every transmit frame that has experienced a late collision. It indicates the number of frames that the TX_MAC has given up transmitting due to collisions that occurred after the TxMinFrameSize number of bytes have already been transmitted. Usually this is an indication that there is at least one station on the network that violates the maximum span of the network.

Random Number Seed Register (RW)

This 10-bit register is used as a seed for the random number generator in the backoff algorithm. The register has significance only after power-on reset, and it should be programmed with a random value which has a high likelihood of being unique for each MAC attached to a network segment (10 LSB of the MAC address). During normal operation, the register contents are updated constantly by the hardware, and a PIO read from this register will return an unpredictable result.

TX_MAC State Machine Register (RO)

This 8-bit register provides the current state for all the state machines in TX_MAC.

TABLE 6-16 TX_MAC State Machine Register

Bits	Field Name	Description
[3:0]		TLM State Machine state
[7:4]		Encapsulation State Machine state

Default: 0x0.

RX_MAC Programmable Resources

RX_MAC Software Reset Command (RW)

This 16-bit command performs a software reset to the logic in the RX_MAC. The defined address must be written with the value of 0x0000.

RX_MAC Configuration Register (RW)

This 13-bit register controls the operation of the RX_MAC.

TABLE 6-17 RX_MAC Configuration Register

Bits	Field Name	Description
[0]	Rx_MAC_Enable	When set to ‘1’, the RX_MAC will start requesting packet data transfers to the ERX, and the receive Ethernet protocol execution will begin. When cleared to ‘0’, it will force the RX_MAC state machines to either remain in the idle state, or to transition to the idle state and stay there
[4:1]	Reserved	
[5]	Strip_Pad	When set to ‘1’, this bit will cause the RX_MAC to strip the “pad” bytes of the receive frames
[6]	Promiscuous_Mode	When set to ‘1’, this bit will cause the RX_MAC to accept all valid frames from the network, regardless of the contents of the DA field of a frame
[7]	Err_Check_Disable	When set to ‘1’, this bit will cause the RX_MAC to receive frames from the network without checking for CRC, framing or length errors
[8]	No_CRC_Strip	When set to ‘1’, this bit will cause the RX_MAC not to strip the last four bytes (FCS) of a received frame
[9]	Reject_My_Frame	When set to ‘1’, this bit will cause the RX_MAC to discard frames with the SA field matching the station’s MAC address
[10]	Promisc_Group_Mode	When set to ‘1’, this bit will cause the RX_MAC to accept all valid frames from the network that have the “group” bit in the DA field set to ‘1’
[11]	Hash_Filter_Enable	When set to ‘1’, the RX_MAC will use the Hash Table to filter multicast addresses
[12]	Address_Filter_Enable	When set to ‘1’, the RX_MAC will use the Address Filtering registers to filter incoming frames

Programming Restrictions: 1. To ensure proper operation of the RX_MAC, the RX_MAC_En bit must always be cleared to ‘0’ and a delay of 3.2msec imposed before a PIO write to any of the other bits in the RX_MAC Configuration register or any of the MAC parameters’ registers is performed. The RX_MAC parameters’ registers are: RxMinFrameSize, RxMaxFrameSize and the MAC Address registers. To avoid the requirement for a fixed time delay, the RX_MAC_En bit may be polled, and when this bit reads back as a ‘0’, all the registers mentioned above may be written, including other bits in the Configuration register.

2. To ensure proper operation of the RX_MAC, the Hash_Filter_Enable bit in the RX_MAC Configuration register must always be cleared to ‘0’ and a delay of 3.2msec imposed before a PIO write to any of the Hash Table registers is performed. To avoid the requirement for a fixed time delay, the Hash_Filter_Enable bit may be polled, and when this bit reads back as a ‘0’, all the registers mentioned above may be written.

3. To ensure proper operation of the RX_MAC, the Address_Filter_Enable bit in the RX_MAC Configuration register must always be cleared to ‘0’ and a delay of 3.2msec imposed before a PIO write to any of the Address Filter registers is performed. To avoid the requirement for a fixed time delay, the Address_Filter_Enable bit may be polled, and when this bit reads back as a ‘0’, all the registers mentioned above may be written.

RxMinFrameSize Register (RW)

This 8-bit register specifies the minimum number of bytes in a frame that the RX_MAC will expect to see before it will recognize the frame to be valid.

Default: 0x40.

RxMaxFrameSize Register (RW)

This 13-bit register specifies the maximum number of bytes in a frame that the RX_MAC will expect to see before it will recognize the frame to be invalid.

Default: 0x05EE.

MAC Address 0 Register (RW)

This register contains the 16 least significant bits of the MAC Address. These bits will be compared against bits [15:0] of the DA field in every frame that arrives from the network.

MAC Address 1 Register (RW)

This register contains bits [31:16] of the MAC Address. These bits will be compared against bits [31:16] of the DA field in every frame that arrives from the network.

MAC Address 2 Register (RW)

This register contains the 16 most significant bits of the MAC Address. These bits will be compared against bits [47:32] of the DA field in every frame that arrives from the network.

Receive Frame Counter (RW)

This 16-bit loadable counter increments after a valid frame has been received from the network.

Length Error Counter (RW)

This 8-bit loadable counter increments when a frame, whose length is greater than the value programmed in the RxMaxFrameSize Register, is received from the network.

Alignment Error Counter (RW)

This 8-bit loadable counter increments when an alignment error was detected in a receive frame. An alignment error is reported when a receive frame fails the CRC checking algorithm, AND the frame does not contain an integer number of bytes (i.e. the frame size in bits modulo 8 is not equal to zero).

FCS Error Counter (RW)

This 8-bit loadable counter increments when a receive frame failed the CRC checking algorithm, but it did not cause an alignment error.

Rx Code Violation Counter (RW)

This 8-bit loadable counter increments when an Rx_Err indication is generated by the XCVR over the MII, while a frame is being received. This indication is generated by the transceiver when it detects an invalid code in the received data stream. A receive code violation is not counted as an FCS or an Alignment error.

RX_MAC State Machine Register (RO)

This 7-bit register provides the current state for all the state machines in the RX_MAC.

TABLE 6-18 RX_MAC State Machine Register

Bits	Field Name	Description
[4:0]		Receive Protocol State Machine state
[6:5]		Pad State Machine state

Hash Table 0 Register (RW)

This register contains bits [15:0] of the Hash Table.

Hash Table 1 Register (RW)

This register contains bits [31:16] of the Hash Table.

Hash Table 2 Register (RW)

This register contains bits [47:32] of the Hash Table.

Hash Table 3 Register (RW)

This register contains bits [63:48] of the Hash Table.

Address Filter 0 Register (RW)

This register contains bits [15:0] of the Address Filter.

Address Filter 1 Register (RW)

This register contains bits [31:16] of the Address Filter.

Address Filter 2 Register (RW)

This register contains bits [47:32] of the Address Filter.

Address Filter Mask Register (RW)

This register contains a 12-bit nibble mask for the Address Filter.

6.3.4.5 MIF Programmable Resources

MIF Bit-Bang Clock (RW)

This 1-bit register is used to generate the MDC clock waveform on the MII Management Interface when the MIF is programmed in the “Bit-Bang” Mode. Writing a ‘1’ after a ‘0’ into this register will create a rising edge on the MDC, while writing a ‘0’ after a ‘1’ will create a falling edge. For every bit that is transferred on the management interface, both edges have to be generated.

MIF Bit-Bang Data (RW)

This 1-bit register is used to generate the outgoing data (MDO) on the MII Management Interface when the MIF is programmed in the “Bit-Bang” Mode. The data will be steered to the appropriate MDIO based on the state of the PHY_Select bit in the MIF Configuration Register.

MIF Bit-Bang Output Enable (RW)

This 1-bit register is used to enable (’1’) and disable (’0’) the I-directional driver on the MII Management Interface when the MIF is programmed in the “Bit-Bang” Mode. The MDIO should be enabled when data bits are transferred from the MIF to the transceiver, and it should be disabled when the interface is idle or when data bits are transferred from the transceiver to the MIF (data portion of a read instruction). Only one MDIO will be enabled at a given time, depending on the state of the PHY_Select bit in the MIF Configuration Register.

MIF Configuration Register (RW)

This 10-bit register controls the operation of the MIF.

TABLE 6-19 MIF Configuration Register

Bits	Field Name	Description
[0]	PHY_Select	The MIF implements two independent management interfaces for two separate transceivers. Only one transceiver can be used at a given time. This bit determines which transceiver is currently in use. When cleared to '0', MDIO_0 is selected. When set to '1', MDIO_1 is selected.
[1]	Poll_Enable	When set to '1', this bit enables the polling mechanism as described in 3.2.2.1.2. If this bit is set to '1', the BB_Mode should be cleared to '0'
[2]	BB_Mode	This bit determines the mode of operation of the MIF. When set to '1', the "bit-bang mode" is selected. When cleared to '0', the "frame mode" will be used.
[7:3]	Poll_Reg_Addr	This field determines the register address in the transceiver that will be polled by the polling mechanism in the MIF. It is meaningful only if the Poll_Enable bit is set to '1'
[8]	MDI_0	This read-only bit is dual-purpose. **When the MDIO_0 interface is idle**, this bit will indicate whether a transceiver is connected to this line. If this bit reads as '1', the transceiver is connected. When the MIF is communicating with a transceiver that is hooked up to MDIO_0 in the bit-bang mode, this bit will indicate the incoming bit stream during a read operation
[9]	MDI_1	This read-only bit is dual-purpose. **When the MDIO_1 interface is idle**, this bit will indicate whether a transceiver is connected to this line. If this bit reads as '1', the transceiver is connected. When the MIF is communicating with a transceiver that is hooked up to MDIO_1 in the bit-bang mode, this bit will indicate the incoming bit stream during a read operation
[14:10]	Poll_Phy_Addr	This field determines the transceiver address to be polled

MIF Frame/Output Register (RW)

This 32-bit register serves as an “Instruction Register” when the MIF is programmed in the Frame Mode. In order to execute a read / write operation from / to a transceiver register, the software has to load this register with a valid instruction, as per IEEE 802.3u MII specification. After issuing an instruction, the software has to poll this register to check for instruction execution completion. During a read operation, this register will also contain the 16-bit data that was returned by the transceiver.

TABLE 6-20 MIF Frame / Output Register

Bits	Field Name	Description
[31:30]	ST	STart of frame. When issuing an instruction: This field should always be loaded with a ‘01’. When polling for completion: This field is always a “don’t care”.
[29:28]	OP	OPcode. When issuing an instruction: This field should be loaded with ‘01’ for a “write” and with ‘10’ for a “read”. When polling for completion: This field is always a “don’t care”.
[27:23]	PHYAD	PHY ADdress. When issuing an instruction: This field should be loaded with the XCVR address. When polling for completion: This field is always a “don’t care”.
[22:18]	REGAD	REGister ADdress. When issuing an instruction: This field should be loaded with the address of the register that is to be read / written. When polling for completion: This field is always a “don’t care”.

TABLE 6-20 MIF Frame/Output Register *(Continued)*

Bits	Field Name	Description
[17]	TA_MSB	Turn Around, Most Significant Bit. When issuing an instruction: This bit should always be loaded with a '1'. When polling for completion: This bit is always a "don't care".
[16]	TA_LSB	Turn Around, Least Significant Bit. When issuing an instruction: This bit should always be loaded with a '0'. When polling for completion: This bit serves as a "Valid Bit". When this bit is set to'1', the instruction execution has been completed.
[15:0]	DATA	Instruction Payload. When issuing an instruction: This field should be loaded with the 16-bit data to be written into a transceiver register for a "write", and is a "don't care" for a "read". When polling for completion: This field is a "don't care" for a "write", and contains the 16-bit data returned by the transceiver for a "read" (if the Valid Bit is set).

MIF Status Register (R-AC)

This 32-bit register is used in conjunction with the Poll Mode in the MIF. It contains two portions: Poll Data and Poll Status. The Poll Data field will always contain the latest and greatest "image update" of the XCVR register that is being polled, while the Poll Status field will indicate which bits in the Poll Data field have changed since the MIF Status Register was last read. The Poll Status field is "auto-cleared" after being read.

TABLE 6-21 MIF Status Register

Bits	Field Name	Description
[31:16]	Poll_Data	
[15:0]	Poll_Status	

MIF Mask Register (RW)

This 16-bit register is used to determine which bits in the Poll Status portion of the MIF Status Register will cause an interrupt. If a mask bit is cleared to '0', the corresponding bit of the Poll Status will generate the MIF Interrupt when set.

Default: 0xFFFF.

MIF State Machine Register (RO)

This 9-bit register provides the current state for all the state machines in the MIF.

TABLE 6-22 MIF State Machine Register

Bits	Field Name	Description
[2:0]		Control State Machine State
[6:5]		Execution State Machine State

6.3.5 Programming Notes

6.3.5.1 Initialization Sequences

▼ Global Initialization

A global initialization sequence should be performed after power-on or when a Global_Reset command is issued to the Ethernet Channel.

1. Issue a Global_Reset command to the Ethernet Channel.
2. Poll the Global_Reset bit until the execution of the reset has been completed.
3. Set up all the data structures in the host memory.
4. Program the TX_MAC registers / counters (excluding TX_MAC Configuration register).
5. Program the RX_MAC registers / counters (excluding RX_MAC Configuration register).
6. Program the Transmit Descriptor Ring Base Address in the ETX.
7. Program the Receive Descriptor Ring Base Address in the ERX.
8. Program the Global Configuration and the Global Interrupt Mask registers.
9. Program the ETX Configuration register (enable the transmit DMA channel).
10. Program the ERX Configuration register (enable the receive DMA channel).
11. Program the XIF Configuration register (enable the XIF).
12. Program the RX_MAC Configuration register (enable the RX_MAC).
13. Program the TX_MAC Configuration register (enable the TX_MAC).
14. Issue the Transmit_Pending command when ready.

Transmit Data Path Initialization

This initialization sequence should be performed for error recovery purposes in the transmit data path.

1. Program the XIF Configuration register (Disable the XIF).
2. Issue a TX_MAC Software Reset command.
3. Issue a ETX Software Reset command.
4. Poll the ETX Software Reset bit until the execution of the reset has been completed.
5. Set up the transmit data structures in the host memory.
6. Program the TX_MAC registers/counters (excluding TX_MAC Configuration register).
7. Program the Transmit Descriptor Ring Base Address in the ETX.
8. Program the ETX Configuration register (enable the transmit DMA channel).
9. Program the XIF Configuration register (enable the XIF).
10. Program the TX_MAC Configuration register (enable the TX_MAC).
11. Issue the Transmit_Pending command when ready.

Receive Data Path Initialization

This initialization sequence should be performed for error recovery purposes in the receive data path.

1. Issue a RX_MAC Software Reset command.
2. Issue a ERX Software Reset command.
3. Poll the ERX Software Reset bit until the execution of the reset has been completed.
4. Set up the receive data structures in the host memory.
5. Program the RX_MAC registers/counters (excluding RX_MAC Configuration register).
6. Program the Receive Descriptor Ring Base Address in the ERX.
7. Program the ERX Configuration register (enable the receive DMA channel).
8. Program the RX_MAC Configuration register (enable the RX_MAC).

6.3.6 Memory Map

TABLE 6-23 Ethernet Channel Engine Address Map

PA<27:0>	Access Size (bytes)	R/W	Description	Actual Size (bits)	Default
0x8C00000	4	RW	Global Software Reset Register	2	0x0
0x8C00004	4	RW	Global Configuration Register	5	0x00
0x8C00100	4	R-AC	Global Status Register	32	0x00000000
0x8C00104	4	RW	Global Interrupt Mask Register	32	0xFFFFFFFF
0x8C02000	4	RW	Transmit Pending Command	1	0x0
0x8C02004	4	RW	ETX Configuration Register	10	0x3FE
0x8C02008	4	RW	Transmit Descriptor Pointer	32	0x00000000
0x8C0200C	4	RO	Transmit Data Buffer Base Address	32	0x00000000
0x8C02010	4	RO	Transmit Data Buffer Displacement	10	0x000
0x8C02014	4	RW	TxFIFO Write Pointer	9	0x000
0x8C02018	4	RW	TxFIFO Shadow Write Pointer	9	0x000
0x8C0201C	4	RW	TxFIFO Read Pointer	9	0x000
0x8C02020	4	RW	TxFIFO Shadow Read Pointer	9	0x000
0x8C02024	4	RW	TxFIFO Packet Counter	8	0x00
0x8C02028	4	RO	ETX State Machine Register	23	0x000000
0x8C0202C	4	RW	Transmit Descriptor Ring Size	4	0xF
0x8C02030	4	RO	Transmit Data Pointer	32	0x00000000
0x8C03000–0x8C037FC	4	RW	TxFIFO Lower Aperture	32	0xXXXXXX
0x8C03800–0x8C03FFC	4	RW	TxFIFO Higher Aperture	32	0xXXXXXX
0x8C04000	4	RW	ERX Configuration Register	23	0x000000
0x8C04004	4	RW	Receive Descriptor Pointer	32	0x00000000
0x8C04008	4	RO	Receive Data Buffer Pointer	28	0x0000000
0x8C0400C	4	RW	RxFIFO Write Pointer	9	0x000
0x8C04010	4	RW	RxFIFO Shadow Write Pointer	9	0x000

TABLE 6-23 Ethernet Channel Engine Address Map *(Continued)*

PA<27:0>	Access Size (bytes)	R/W	Description	Actual Size (bits)	Default
0x8C04014	4	RW	RxFIFO Read Pointer	9	0x000
0x8C04018	4	RW	RxFIFO Packet Counter	8	0x00
0x8C0401C	4	RO	ERX State Machine Register	32	0x00000000
0x8C05000–0x8C057FC	4	RW	RxFIFO Lower Aperture	32	0xXXXXXX
0x8C05800–0x8C05FFC	4	RW	RxFIFO Higher Aperture	32	0xXXXXXX
0x8C06000	4	RW	XIF Configuration Register	10	0x000
0x8C06208	4	RW	TX_MAC Software Reset Command	1	0x0
0x8C0620C	4	RW	TX_MAC Configuration Register	11	0x000
0x8C06210	4	RW	InterPacketGap1 Register	8	0x08
0x8C06214	4	RW	InterPacketGap2 Register	8	0x04
0x8C06218	4	RW	AttemptLimit Register	8	0x10
0x8C0621C	4	RW	SlotTime Register	8	0x40
0x8C06220	4	RW	PA Size Register	8	0x07
0x8C06224	4	RW	PA Pattern Register	8	0xAA
0x8C06228	4	RW	SFD Pattern Register	8	0xAB
0x8C0622C	4	RW	JamSize Register	8	0x04
0x8C06230	4	RW	TxMaxFrameSize Register	16	0x05EE
0x8C06234	4	RW	TxMinFrameSize Register	8	0x40
0x8C06238	4	R-AC	Peak Attempts Register	8	0x00
0x8C0623C	4	RW	Defer Timer	16	0x0000
0x8C06240	4	RW	Normal Collision Counter	16	0x0000
0x8C06244	4	RW	First Successful Collision Counter	16	0x0000
0x8C06248	4	RW	Excessive Collision Counter	8	0x00
0x8C0624C	4	RW	Late Collision Counter	8	0x00
0x8C06250	4	RW	Random Number Seed Register	10	0x000
0x8C06254	4	RO	TX_MAC State Machine Register	8	0x00
0x8C06308	4	RW	RX_MAC Software Reset Command	16	0xXXXX

TABLE 6-23 Ethernet Channel Engine Address Map *(Continued)*

PA<27:0>	Access Size (bytes)	R/W	Description	Actual Size (bits)	Default
0x8C0630C	4	RW	RX_MAC Configuration Register	13	0x0000
0x8C06310	4	RW	RxMaxFrameSize Register	13	0x05EE
0x8C06314	4	RW	RxMinFrameSize Register	8	0x40
0x8C06318	4	RW	MAC Address 2 Register	16	0x0000
0x8C0631C	4	RW	MAC Address 1 Register	16	0x0000
0x8C06320	4	RW	MAC Address 0 Register	16	0x0000
0x8C06324	4	RW	Receive Frame Counter	16	0x0000
0x8C06328	4	RW	Length Error Counter	8	0x00
0x8C0632C	4	RW	Alignment Error Counter	8	0x00
0x8C06330	4	RW	FCS Error Counter	8	0x00
0x8C06334	4	RO	RX_MAC State Machine Register	7	0x00
0x8C06338	4	RW	Rx Code Violation Error Counter	8	0x00
0x8C06340	4	RW	Hash Table 3 Register	16	0x0000
0x8C06344	4	RW	Hash Table 2 Register	16	0x0000
0x8C06348	4	RW	Hash Table 1 Register	16	0x0000
0x8C0634C	4	RW	Hash Table 0 Register	16	0x0000
0x8C06350	4	RW	Address Filter 2 Register	16	0x0000
0x8C06354	4	RW	Address Filter 1 Register	16	0x0000
0x8C06358	4	RW	Address Filter 0 Register	16	0x0000
0x8C0635C	4	RW	Address Filter Mask Register	12	0x000
0x8C07000	4	RW	MIF Bit-Bang Clock	1	0x0
0x8C07004	4	RW	MIF Bit-Bang Data	1	0x0
0x8C07008	4	RW	MIF Bit-Bang Output Enable	1	0x0
0x8C0700C	4	RW	MIF Frame/Output Register	32	0x00000000
0x8C07010	4	RW	MIF Configuration Register	10	0xX00
0x8C07014	4	RW	MIF Mask Register	16	0xFFFF
0x8C07018	4	R-AC	MIF Status Register	32	0x00000000
0x8C0701C	4	RO	MIF State Machine Register	9	0x000

CHAPTER 7

EBus2 Channel Engine

7.1 Introduction

This chapter describes an interface which provides the ability to put ISA and traditional Intel-style peripherals in a SPARC based system with a minimal amount of glue logic. The interface is called EBus2 and is a part of the PCIO chip.

7.1.1 Features

The following features are supported by the EBus2 Channel Engine:

- Capable of supporting EPROM, TOD/NVRAM, Audio CODEC, SUPERIO, external serial ports and generic Intel-style (ISA) slave and slave DMA devices
- 8 chip selects
- Programmable cycle time control for slave and DMA accesses
- Supports up to 4 bytes stacking for slave cycles to Ebus2 devices
- Supports slave buffered writes
- Multiplexed address & data bus
- Four DMA controllers providing similar programming interface as SCSI DMA controller of DMA2
- Programmable chained or unchained mode of operation
- Programmable transfer size for each DMA controller
- 128 byte buffer for each DMA controller
- Programmable DMA priority algorithms

A block diagram of EBus2 channel engine is shown in FIGURE 7-1.

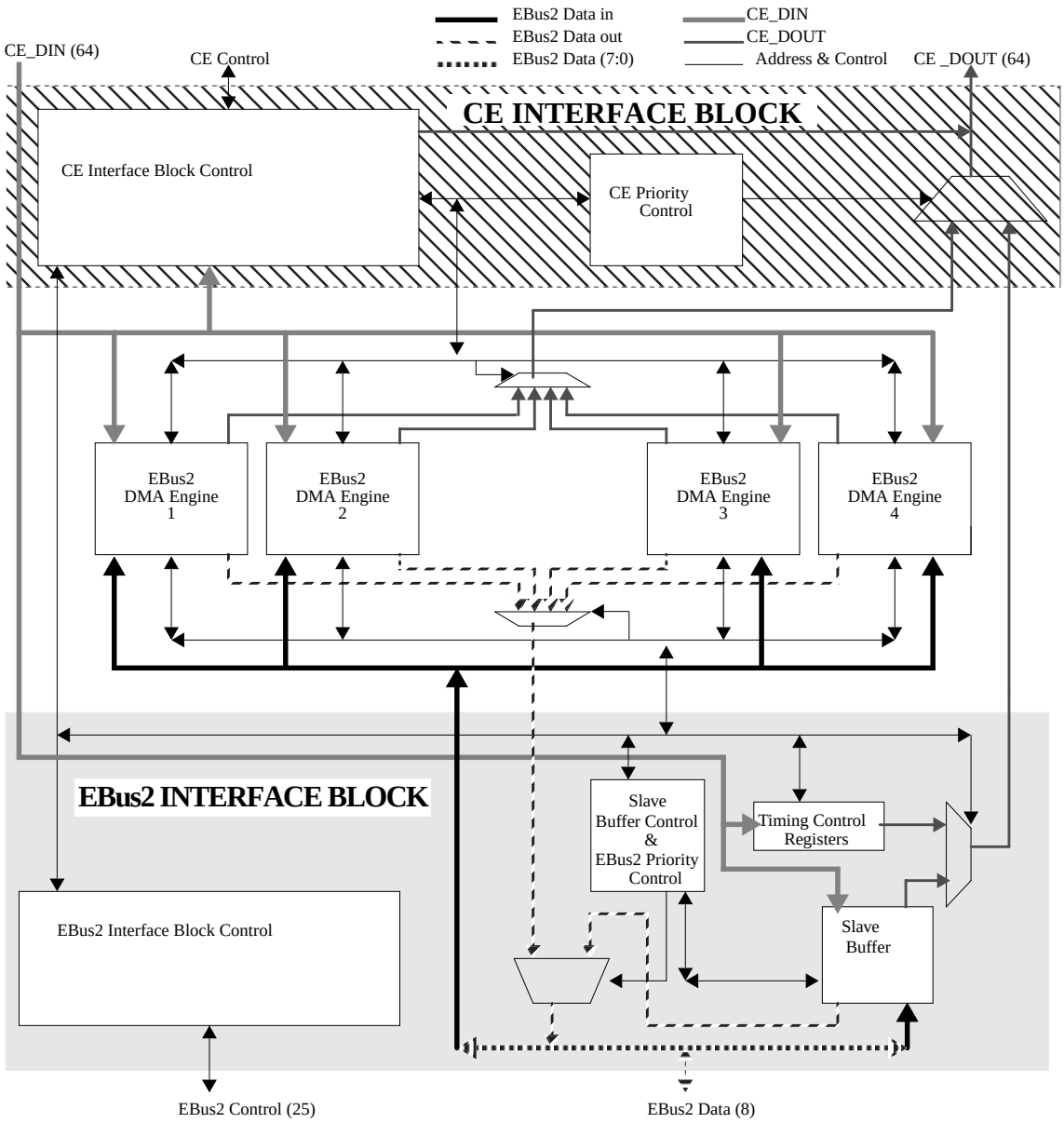

FIGURE 7-1 EBus2 channel engine block diagram

7.2 Address Map

TABLE 7-1 and TABLE 7-2 give address map for EBus2 channel engine. For Access Size, B=8 bits, H=16 bits and W=32 bits.

TABLE 7-1 EBus2 Address Map

Address	Name	Description	Device	R/W	Access
Base Address: EBus2 Base Address Register 0 (Boot ROM) - Config Space Address 0x010					
0x00 0000 - FF FFFF	EB_CS0_	EBus2 chip select 0	EPROM/Flash PROM	R/W	BHW
Base Address: EBus2 Base Address Register 1 (EBus2) - Config Space Address 0x014					
0x00 0000 - 0F FFFF	EB_CS1_	EBus2 chip select 1	TOD/NVRAM	R/W	BHW
0x10 0000 - 1F FFFF	EB_CS2_	EBus2 chip select 2	Generic Ebus2 device	R/W	BHW
0x20 0000 - 2F FFFF	EB_CS3_	EBus2 chip select 3	Audio	R/W	BHW
0x30 0000 - 3F FFFF	EB_CS4_	EBus2 chip select 4	Super IO	R/W	BHW
0x40 0000 - 4F FFFF	EB_CS5_	EBus2 chip select 5	sync. serial	R/W	BHW
0x50 0000 - 5F FFFF	EB_CS6_	EBus2 chip select 6	SC/Lab Console/ Freq.	R/W	BHW
0x60 0000 - 6F FFFF	EB_CS7_	EBus2 chip select 7	Generic EBus2 device	R/W	BHW
0x70 0000 - 70 1FFF	DCTL1	DMA controller 1	Parallel	R/W	W
0x70 2000 - 70 3FFF	DCTL2	DMA controller 2	Audio (play back)	R/W	W
0x70 4000 - 70 5FFF	DCTL3	DMA controller 3	Audio (capture)	R/W	W
0x70 6000 - 70 7FFF	DCTL4	DMA controller 4	Floppy	R/W	W
0x71 0000 - 71 0003	TCR1	Timing control register 1	Internal Control of EBus2	R/W	W
0x71 0004 - 71 0007	TCR2	Timing control register 2	Internal Control of EBus2	R/W	W

TABLE 7-1 EBus2 Address Map *(Continued)*

Address	Name	Description	Device	R/W	Access
0x71 0008 - 71 000B	TCR3	Timing control register 3	Internal Control of EBus2	R/W	W
0x72 0000	FLP_AUX	Floppy auxio register	Internal Register of Ebus2	R/W	W
0x72 2000	AUD_AUX	Audio auxio register	Internal Register of Ebus2	R/W	W
0x72 4000	PWR_AUX	Power auxio register	Internal Register of Ebus2	R/W	W
0x72 6000	LED_AUX	LED auxio register	Internal Register of Ebus2	R/W	W
0x72 8000	PMD_AUX	PCI/Mode auxio register	Internal Register of Ebus2	R	W
0x72 A000	FREQ_AUX	Frequency auxio register	Internal Register of Ebus2	R	W
0x72 C000	OSC_AUX	SCSI Oscillator auxio register	Internal Register of Ebus2	R/W	W
0x72 F000	TEMP_AUX	Temp. sense auxio register	Internal Register of Ebus2	R/W	W
0x73 0000 - 73 00FC	BNK0	DMA FIFO bank 0	Bank 0 (64 words)	R/W	W
0x73 0100 - 73 01FC	BNK1	DMA FIFO bank 1	Bank1 (64 words)	R/W	W

TABLE 7-2 DMA registers

Address Offset	Name	Description	R/W	Access
DMA Controller 1				
0x70 0000	DCSR1	DMA1 CSR register	R/W	W
0x70 0004	DACR1	DMA1 address count register	R/W	W
0x70 0008	DBCR1	DMA1 byte count register	R/W	W
DMA Controller 2				
0x70 2000	DCSR2	DMA2 CSR register	R/W	W
0x70 2004	DACR2	DMA2 address count register	R/W	W
0x70 2008	DBCR2	DMA2 byte count register	R/W	W
DMA Controller 3				
0x70 4000	DCSR3	DMA3 CSR register	R/W	W
0x70 4004	DACR3	DMA3 address count register	R/W	W
0x70 4008	DBCR3	DMA3 byte count register	R/W	W
DMA Controller 4				
0x70 6000	DCSR4	DMA4 CSR register	R/W	W
0x70 6004	DACR4	DMA4 address count register	R/W	W
0x70 6008	DBCR4	DMA4 byte count register	R/W	W

Note – Devices associated with Ebus chip selects in the preceding table are recommended devices. It is imperative that the PROM should be in the address range shown. The EBus2 has 24 Megabytes of address space. The address space for PROM is 16 Megabytes while the other EBus2 devices and internal registers occupy 8 Megabytes of address space. In a PCI based system the base address of the devices can be configured through configuration registers.

7.3 EBus2 Slave Interface Description

The EBus2 slave interface provides the ability to do slave cycles on the EBus2. The EBus2 slave interface provides eight chip selects. The Slave cycle timings on EBus2 is programmable. Timing control is provided for 7 address ranges which correspond to ranges shown for EB_CS1_ through EB_CS7_ in TABLE 7-1. The timing control for the EBus2 slave cycles is provided in timing Control register 2 (TCR2) and timing control register 3 (TCR3). At the boot time OBP programs these registers. The timing access to address range corresponding to EB_CS0_ is hardwired. The PROM resides in the address space corresponding to EB_CS0_.

The EBus2 is a multiplexed address and data bus and requires external logic to latch the address for each slave cycle. External latches are used to store the address 23:8.

7.3.1 Functional Description

Each EBus2 slave cycle consists of two phases, an address phase and a data phase.

7.3.1.1 Address Phase

The Ebus2 provides 8 data lines and 8 address lines. The EBus2 slave cycle begins with an address phase. During the address phase all data and address lines in conjunction with CLKEN signals are used to latch address bits 23:8 in external latches. When the EBus2 slave interface is in byte stacking mode, the first transfer requires latching of address bits 23:8, while each successive transfer can take place without incurring an extra cycle for latching the address as Ebus2 address lines provides 8 least significant bits of address.

7.3.1.2 Data Phase

The data phase starts at the end of the address phase by the assertion of EB_CSx_. The actual data is transferred during this phase. The cycle time for the data phase can be programmed through Timing control registers. Refer to Section 7.5.2. This section describes the register set for the EBus2 channel engine for details on the parameters that can be programmed. During the Data phase, Data is provided on Ebus2 data lines and lower byte of address is provided on address lines. This operation is in contrast to the address phase where all data and address lines are used to provide address 23:8.

7.3.1.3 Byte Stacking

The Ebus2 slave interface supports two and four bytes stacking for the Ebus2 devices. The byte stacking is done in the little endian format. A four byte slave buffer is provided to aid in byte stacking and buffered writes. During the byte stacking a word (32 bit) or a half word (16 bit) can be initiated on the PCI side. The Ebus2 controller will break the half word or word needed to be transferred on the PCI side in bytes on the Ebus2. This mechanism allows better utilization of PCI bus.

7.3.1.4 Buffered Slave Transfers

EBus2 channel engine supports buffered write cycles for host writes which implies that it acknowledges the completion of a write cycle to the PCI bus adapter, before the write cycle is actually completed on the EBus2. This reduces the duration of write cycle on the PCI bus. The EBus2 channel engine has a 4 byte slave buffer. It acknowledges the bus interface immediately if the buffer has space for the transfer size required.

7.3.1.5 IOCHRDY

Any EBus2 slave or DMA device can extend the slave or DMA cycle by deasserting the IOCHRDY signal. The EB_RD or EB_WR signal remains asserted for the entire time IOCHRDY signal is negated. If the strobe width (Tstrb), as programmed in timing control registers, is greater than this time, EB_RD or EB_WR will remain asserted for Tstrb width. If IOCHRDY is not negated two EBus2 clock cycles before the deassertion of EB_RD or EB_WR then IOCHRDY is ignored.

7.3.1.6 Slave Transfer Size

The Ebus2 address space can be accessed as byte, half word (two bytes) and word (four bytes). EBus2 channel engine does the byte stacking for half word and word transfers. Any other transfer size will result in an error acknowledgment.

7.4 EBus2 DMA Interface Description

This section details the operation of the EBus2 DMA interface. The PCIO EBus2 has four DMA engines. Each DMA controller provides similar programmers interface as SCSI DMA controller (ESP interface) of the DMA2 chip, though there are some differences between two implementations. These are listed in Section 7.4.1.7, “Differences between EBus2 DMA Engine and SCSI DMA of DMA2”. All controllers

are functionally identical and general purpose to interface with ISA / Intel style slave DMA devices. All DMA controllers support both chained and unchained modes of operation.

Each engine has a DMA address counter (DACR), a DMA byte counter (DBCR), a DMA next address register (DNAR), a DMA next byte count register (DNBR) and a DMA control and status register (DCSR). A 128 byte buffer is provided for each DMA controller. The transfer size to system memory is programmable for each DMA controller.

The DMA cycle time for each controller can be programmed through timing control register 1 (TCR1). The priority algorithm of the EBus2 DMA controllers is also programmable and can be programmed through the timing control register 3 (TCR3).

The functional description of DMA controllers follows. The description applies to all DMA controllers.

7.4.1 Functional Description

EBus2 DMA controllers support programmable transfer sizes. Transfer sizes of four, sixteen, thirty two and sixty four bytes are supported. The transfer size (TSIZE) can be programmed by the “BURST_SIZE” bits of the DCSR.

7.4.1.1 Transfers From System Memory (DMA Read)

When the buffer of any DMA controller has space for TSIZE bytes, it will initiate a DMA read from system memory. A read ahead is done if it does not cross the page boundary. These bytes are transferred to the EBus2 device, associated with the DMA controller, when it asserts the DMA request (DREQx) signal and the priority of the device is highest in the priority queue. Refer to timing diagrams for the EBus2 DMA cycles.

7.4.1.2 Transfers To System Memory (DMA Write)

The TSIZE bytes are gathered by the DMA controller before initiating a DMA write to system memory. These bytes are transferred from the EBus2 device, associated with the DMA controller, when it asserts the DREQx signal and the priority of the device is highest in the priority queue.

7.4.1.3 Chained Mode

Each EBus2 DMA controller provides an optional chained mode of operation. Chaining provides a mechanism of overcoming the interrupt latency at the end of transfer by providing a set of DMA next address register (DNAR) and DMA next byte count register (DNBR). The software can load these registers while DMA is occurring. The contents of DNAR and DNBR are copied to DACR and DBCR respectively, when the value in DBCR expires. Thus the DMA is not subjected to the delay for updating DACR and DBCR.

The chaining can be enabled in any EBus2 DMA controller by setting the EN_NEXT bit of DCSR of the respective DMA engine.

7.4.1.4 End of Transfer (Terminal Count)

The DBCR decrements for an EBus2 DMA engine on each DMA byte transferred to or from the EBus2 device. When the byte count expires a terminal count signal (TCS) is given to the associated EBus2 device to terminate the DMA transfer on the EBus2. On a terminal the count following events happen on the host side of the EBus2 DMA engine.

- For the DMA read, all the remaining bytes in the DMA buffer are discarded then an interrupt is generated to the host if TCI_DIS = 0 and INT_EN =1, in DCSR
- For the DMA write, the remaining bytes in the buffer are transferred to system memory with a succession of the largest legal transfer size (64 bytes, 32 bytes, 16 bytes, 4 bytes, 2bytes, 1 byte). Hardware will ensure that all the bytes from the FIFO will be drained. The terminal count signal (TCS) to the EBus2 device is asserted as soon as byte count expires but the terminal count interrupt to the host is held off until the DMA transfer to the memory is complete (i.e.- automatic draining until DMA FIFO is empty)

7.4.1.5 EBus2 Device Acknowledgment

During a DMA transfer on the EBus2, once a device is given the use of the EBus2 it can transfer data until the following conditions happen.

- Device deasserts DMA request (DREQx) signal
- A slave cycle on the EBus2 occurs. An ongoing DMA burst cycle on the EBus2 will be held off to let the slave cycle complete. The DMA burst cycle will resume after the completion of the slave cycle
- The EBus2 DMA buffer will become empty while doing DMA read, or full in case of doing DMA write. In this situation the next EBus2 device in the priority queue will be acknowledged
- A host bus Error occurs. The next EBus2 device in the priority queue will be acknowledged

- During chaining the DBCR expires and the DNBR and DNAR are not loaded. The next device in the priority queue will be acknowledged
- DMA is disabled by writing a “0” to EN_DMA bit of DCSR. In this case no more data is transferred to or from the currently active device on the EBus2. DBCR of the controller holds the current value. If a transfer from or to the memory is in progress it is completed. When DMA is disabled through this mechanism the next device in the priority queue gets the use of the EBus2
- Sixty four successive bytes are transferred to a single device. After that the DMA controller has to rearbitrate

7.4.1.6 Host Bus Errors

When an EBus2 DMA controller has communicated the information regarding the host bus errors it halts and does not give any more data to the currently active DMA device on the EBus2. No more data is transferred to or from the memory. DACR and DBCR hold the current values. Bus error ERR_PEND bit gets set in DCSR, if INT_EN bit is “1” in DCSR, an interrupt is generated. The DMA _ON bit gets cleared in the DCSR.

7.4.1.7 Differences between EBus2 DMA Engine and SCSI DMA of DMA2

This section describes the differences between the EBus2 DMA engine and the DMA2 SCSI DMA engine. This section will be updated as additional differences between the two designs become apparent.

1. EBus2 DMA engine issues the terminal count interrupt when all the FIFO data is drained after the expiration of DBCR for DMA writes. The DMA2 SCSI DMA engine issues a terminal count interrupt as soon as the byte counter expires without waiting for all the FIFO data to be drained to the memory.
2. Transfer size of 1 byte, 2 bytes, 4 bytes, 16 bytes, 32 bytes and 64 bytes are supported by EBus2 DMA engine compared to 1 byte, 2 bytes, 4 bytes, 16 bytes and 32 bytes supported by DMA2 SCSI DMA engine.
3. These bits of SCSI DMA CSR of DMA2 will not be supported by the EBus2 DMA CSR: D_SLAVE_ERR, D_DIAG, D_TWO_CYCLE, D_FASTER, D_DRAINING (one bit instead of two).
4. The DMA cycles for the EBus2 DMA engine offers greater programmability compared to DMA2. Separate timing control registers are provided which offer range of programmable parameters.
5. The diagnostic mechanism will be different for EBus2 DMA engine and DMA2 SCSI DMA engine.

6. The Address counter in the EBus2 DMA engine always points to the next byte that has to be accessed in the memory. The DMA2 SCSI DMA engine points to the next byte that will be accessed by the SCSI device, independent of which bytes in the memory have been accessed by the DMA engine.

7.4.2 Priority Mechanism

Since EBus2 has limited bandwidth and it is possible that bandwidth requirements for different devices on the EBus2 differ from each other, two priority schemes are provided for the DMA devices on the EBus2 to optimize the use of the EBus2. The Timing control register 3 (TCR3) provides a bit (PR) which can be programmed on the boot time to choose between the two schemes. The encoding of the bit in TCR3 is shown in TABLE 7-3.

TABLE 7-3 Encoding of Timing control register 3 PR bit

PR	Priority Level
0	2
1	1

The priority of EBus2 DMA controller 1 can be increased by level 2 priority, since Parallel port is the device which may require much more bandwidth than the other identified devices on the EBus2, it is recommended that parallel port should be connected to EBus2 DMA controller 1 with Level 2 priority.

The priority schemes are as follows:

7.4.2.1 Level 1

The scheme for level 1 priority is round robin, starting from controller 1. The scheme is as follows:

controller 1 → controller 2 → controller 3 → controller 4 → controller 1 → controller 2 → controller 3 → controller 4 →...

7.4.2.2 Level 2

This scheme gives every second priority to controller 1. The scheme is as follows:

controller 1 → controller 2 → controller 1 → controller 3 → controller 1 → controller 4 →...

7.4.3 Data Rates of EBus2 DMA Devices

Currently two DMA devices are identified on the EBus2. These devices are SUPERIO (provides DMA channels for parallel port and floppy disk) and Audio Codec (provides DMA channels for Play back and capture). TABLE 7-4 illustrates the maximum data rate supported by these channels and latency tolerance per effective size of DMA FIFO. Where,

Effective Size of DMA FIFO = Total FIFO bytes - TSIZE bytes

e.g. If an EBus2 DMA engine is used for parallel port and the TSIZE is programmed by BURST_SIZE bits in DCSR as 64 bytes, then

Effective Size of DMA FIFO = 128 - 64 = 64 bytes

If the parallel port is transferring data at 2 Mbyte/sec, the DMA latency tolerance for the parallel port channel engine is 32 µs (0.5 × 64= 32). This assumes that SUPERIO does not have internal buffering in the parallel port channel. However, SUPERIO has at least 16 bytes of internal buffering in the parallel port channel so the DMA latency tolerance for the parallel port channel is actually 40 µs (32 + 0.5 × 16 = 40).

TABLE 7-4 Data rate and Latency tolerance of EBus2 DMA devices

Channel	Max. Data Rate	Latency tolerance per byte of effective size of DMA fifo
Parallel Port	2 Mbyte/sec	0.5 µs
Audio (play back)	0.2 Mbyte/sec	5 µs
Audio (capture)	0.2 Mbyte/sec	5 µs
Floppy	0.125 Mbyte/sec	8 µs

7.4.4 DMA Testing

Three bits are provided in DMA control and status register which help in testing the DMA FIFO. The bits are DIAG_EN, DIAG_RD_DONE, DIAG_WR_DONE. The following steps should be followed to make use of this testability feature.

- After the reset the DMA address count register is written with the address from which transfer is to be performed. The default direction of DMA transfer is READ - from system memory to PCIO FIFO's - following reset.
- The DIAG_EN and bit is written as '1' which puts the EBus2 DMA engine in the diagnostic mode and initiates DMA activity.

- The EBus2 DMA engine starts to request the read data from the channel engine interface with the size specified by the BURST_SIZ field in the DCSR register. This will repeat until all transfers are made. At this point the 128 byte DMA FIFO is full and DIAG_RD_DONE bit gets set and no more data transfer will take place.
- The host will poll till the DIAG_RD_DONE bit is set. When the DIAG_RD_DONE bit gets set the host will write a '1' in RESET bit which will clear the DIAG_RD_DONE status and initialize FIFO pointers.
- Next the host will load the DACR with the address to which the DMA write is to be performed. The DACR address needs to be the same offset as specified by the BURST_SIZ field, i.e.: If BURST_SIZ is specified for 1 word, the DACR needs to be at a minimum word aligned (pa[1:0] = 0's).
- The host will write a '1' in the WRITE and DIAG_EN bits which will cause the EBus2 DMA engine to start transferring data from the EBus2 FIFO across the channel engine interface with the size specified by the BURST_SIZ field in the DCSR register. This will repeat until the channel engine FIFO becomes empty.
- When the channel engine interface indicates to the EBus2 channel engine that it has dispatched the last byte of data (128th byte) to system memory, the DIAG_WR_DONE bit gets set.
- Now the host can verify that data it has received is actually the same data it wrote to EBus2 channel engine.
- To come out of diagnostic mode the RESET bit should be set in the DCSR register.
- During the diagnostic mode only RESET and ERR_PEND bit are functional. The other bits should be ignored.

7.5 EBus2 Register Description

This section describes the register set for the EBus2 channel engine.

7.5.1 AUXIO Registers

Eight registers are provided to support auxiliary I/O functions required by Sun systems.

7.5.1.1 Floppy AUXIO Register

This register provides functions required by Sony ™floppy drives.

TABLE 7-5 Floppy AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	Floppy density sense	R/Input	1 = high Density
1	Floppy density select	RW/Output	1 = 2 MB, 0 = 1.6 MB (HDD only)
31:2	Unused	R/Internal	Read as Zero

7.5.1.2 Audio AUXIO Register

This register provides power down for the Audio chip.

TABLE 7-6 Audio AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	CODEC powerdown	RW/Output	1 = power down
31:1	Unused	R/Internal	Read as Zero

7.5.1.3 Power AUXIO Register

This register provides system power supply control bits.

TABLE 7-7 Power AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	System poweroff	RW/Output	1 = off
1	Courtesy poweroff	RW/Output	1 = off
31:2	Unused	R/Internal	Read as Zero

7.5.1.4 LED AUXIO Register

This register provides system LED control.

TABLE 7-8 LED AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	LED	RW/Output	1 = on
31:1	Unused	R/Internal	Read as Zero

7.5.1.5 PCI/Mode AUXIO Register

This register provides PCI slot present bits and provide a bit to select between motherboard and addin card mode.

TABLE 7-9 PCI/Mode AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	PCI slot0 prsnt1	R/Input	1st bit for PCI slot 0
1	PCI slot0 prsnt2	R/Input	2nd bit for PCI slot 0
2	PCI slot1 prsnt1	R/Input	1st bit for PCI slot 1
3	PCI slot1 prsnt2	R/Input	2nd bit for PCI slot 1
4	PCI slot2 prsnt1	R/Input	1st bit for PCI slot 2
5	PCI slot2 prsnt2	R/Input	2nd bit for PCI slot 2
6	PCI slot3 prsnt1	R/Input	1st bit for PCI slot 3
7	PCI slot3 prsnt2	R/Input	2nd bit for PCI slot 3
8	Mode	R/Input	0 = Motherboard
31:9	Unused	R/internal	Read as Zero

7.5.1.6 Frequency AUXIO Register

This register provides bits for system frequency margining.

TABLE 7-10 Frequency AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	Freq0	R/Input	Frequency Margining status bit 0
1	Freq1	R/Input	Frequency Margining status bit 1
2	Freq2	R/Input	Frequency Margining status bit 2
31:2	Unused	R/Internal	Read as Zero

7.5.1.7 SCSI Oscillator AUXIO Register

This register provides bits for SCSI oscillator control.

TABLE 7-11 SCSI oscillator AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	Int. SCSI OscEN	RW/Internal	Reset to 1, 0 = disable SCSI clk
1	Ext. SCSI OscEN	R/Input	Reflects scsi_oscen pin
31:2	Unused	R/Internal	Read as Zero

7.5.1.8 Temperature Sense AUXIO Register

This register provides bits for temperature sensor control.

TABLE 7-12 Temperature sense AUXIO register bit definitions

Bit #	Bit Name	Access/Direction	Comments
0	tsens clk	RW/Output	Temperature sensor clock
1	tsens cs_l	RW/Output	Temperature sensor chip select
2	tsens en_l	RW/Internal	Temperature sensor enable
3	tsens data out	R/Output	Temperature sensor data out
4	tsens data in	R/Input	Temperature sensor data in
31:2	Unused	R/Internal	Read as Zero

7.5.2 Timing Control Registers

Three registers are provided to control the following functions:

- Setup time (Tsu) and Hold time (Thld) of EB_CSx_ or DACKx_ with respect to EB_RD_ or EB_WR_ strobe.
- Minimum deassertion time or Recovery time (Trec) between consecutive EB_RD_ or EB_WR_ strobes.
- Width (Tstrb) of EB_RD_ or EB_WR_ strobes.
- Selection of DMA priority Algorithm.

FIGURE 7-2 illustrates the timing parameters which can be controlled through timing control registers.

Timing control registers are programmed at boot time. They should not be touched after the boot time. Note the timing given in timing control register tables is in terms of number of EBus2 clocks. EBus2 clock is same as the PCI clock which has a period of 30 ns.

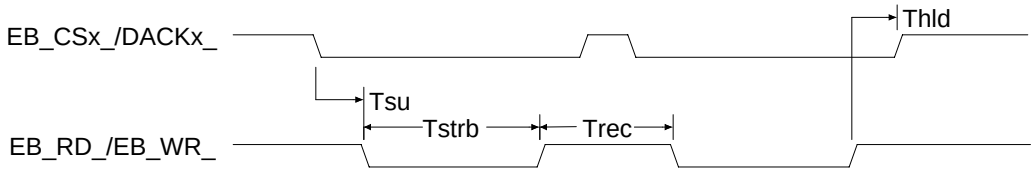

FIGURE 7-2 Programmable timing parameters

7.5.2.1 Timing Control Register 1 (TCR1)

This register controls the strobe width (Tstrb) of EB_RD_ and EB_WR_ signals and recovery time (Trec) for them during DMA cycles on the EBus2. Timing control is provided for all four DMA controllers. This register also controls the setup time (Tsu) and hold time (Thld) for EB_RD_ or EB_WR_ signals with respect to DACKx signal. The general encoding of bits is in TABLE 7-13, TABLE 7-5 and TABLE 7-15, where DTrec0n, DTrec1n, DTrec2n selects the recovery time (Trec) for EB_RD_ or EB_WR_ strobes. DTstrb0n, DTstrb1n, DTstrb2n selects the EB_RD_ and EB_WR_ strobe width (Tstrb). DTsun and DThldn shows the setup (Tsu) and hold time (Thld) for the DMA controller ‘n’. TABLE 7-16 shows bit definitions of TCR1.

TABLE 7-13 Recovery Time (Trec) based on TCR1 bit encoding

DTrec2n	DTrec1n	DTrec0n	Trec (Number of EBus2 Clocks)
0	0	0	3
0	0	1	4
0	1	0	5
0	1	1	6
1	0	0	7
1	0	1	8
1	1	0	9
1	1	1	10

TABLE 7-14 Strobe width (Tstrb) based on TCR1 bit encoding

DTstrb2n	DTstrb1n	DTstrb0n	Tstrb (EBus2 Clocks)
0	0	0	2
0	0	1	3
0	1	0	4
0	1	1	5
1	0	0	6
1	0	1	7
1	1	0	8
1	1	1	9

TABLE 7-15 Tsu and Thld based on TCR1 bit encoding

DTsun	DThldn	Tsu (Number of EBus2 Clocks)	Thld (Number of EBus2 Clocks)
0	0	1	1
0	1	1	2
1	0	2	1
1	1	2	2

TABLE 7-16 Timing control register 1 (TCR1) bit definitions

Bit #	Name	Description	R/W
2:0	DTrec21, DTrec11, DTrec01	Bits to control Trec for DMA controller 1	R/W
5:3	DTstrb21, DTstrb11, DTstrb01	Bits to control Tstrb for DMA controller 1	R/W
6	DTsu1	Bit to control Tsu for DMA Controller 1	R/W
7	DThld1	Bit to control Thld for DMA controller 1	R/W
10:8	DTrec22, DTrec12, DTrec02	Bits to control Trec for DMA controller 2	R/W
13:11	DTstrb22, DTstrb12, DTstrb02	Bits to control Tstrb for DMA controller 2	R/W
14	DTsu2	Bit to control Tsu for DMA Controller 2	R/W
15	DThld2	Bit to control Thld for DMA controller 2	R/W
18:16	DTrec23, DTrec13, DTrec03	Bits to control Trec for DMA controller 3	R/W
21:19	DTstrb23, DTstrb13, DTstrb03	Bits to control Tstrb for DMA controller 3	R/W
22	DTsu3	Bit to control Tsu for DMA Controller3	R/W
23	DThld3	Bit to control Thld for DMA controller 3	R/W
26:24	DTrec24, DTrec14, DTrec04	Bits to control Trec for DMA controller 3	R/W
29:27	DTstrb24, DTstrb14, DTstrb04	Bits to control Tstrb for DMA controller 3	R/W
30	DTsu4	Bit to control Tsu for DMA Controller 3	R/W
31	DThld4	Bit to control Thld for DMA controller 3	R/W

7.5.2.2 Timing Control Register 2 (TCR 2)

TCR2 provides timing control for EB_CS1_ through EB_CS4_This register controls the strobe width (Tstrb) of EB_RD_ and EB_WR_ signals and recovery time (Trec) for them during the slave cycles on the EBus2. This register also controls the setup time (Tsu) and hold time (Thld) for EB_RD_ or EB_WR_ with respect to EB_CSx signal. The general encoding of bits is in TABLE 7-17, Table 7-18 and TABLE 7-19, where STrec0n, STrec1n, STrec2n selects the recovery time (Trec) for EB_RD_ or EB_WR_

strobes. STstrb0n, STstrb1n, STstrb2n selects the EB_RD_ and EB_WR_ strobe width (Tstrb). STsun and STholdn shows the setup (Tsu) and hold time (Thld) for the Chip selects ‘n’. TABLE 7-20 shows bit definitions of TCR2.

TABLE 7-17 Recovery Time (Trec) based on TCR2 bit encoding

STrec2n	STrec1n	STrec0n	Trec (Number of EBus2 Clocks)
0	0	0	3
0	0	1	4
0	1	0	5
0	1	1	6
1	0	0	7
1	0	1	8
1	1	0	9
1	1	1	10

TABLE 7-18 Strobe width (Tstrb) based on TCR2 bit encoding

STstrb2n	STstrb1n	STstrb0n	Tstrb (EBus2 Clocks)
0	0	0	2
0	0	1	3
0	1	0	4
0	1	1	5
1	0	0	6
1	0	1	7
1	1	0	8
1	1	1	9

TABLE 7-19 Setup time (Tsu) and hold time (Thld) based on TCR2 bit encoding

STsun	SThldn	Tsu (Number of EBus2 Clocks)	Thld (Number of EBus2 Clocks)
0	0	1	1
0	1	1	2
1	0	2	1
1	1	2	2

TABLE 7-20 Timing control register 2 (TCR2) bit definitions

Bit #	Name	Description	R/W
2:0	STrec21, STrec11, STrec01	Bits to control Trec for EB_CS1_	R/W
5:3	STstrb21, STstrb11, STstrb01	Bits to control Tstrb for EB_CS1_	R/W
6	STsu1	Bit to control Tsu for EB_CS1_	R/W
7	SThld1	Bit to control Thld for EB_CS1_	R/W
10:8	STrec22, STrec12, STrec02	Bits to control Trec for EB_CS2_	R/W
13:11	STstrb22, STstrb12, STstrb02	Bits to control Tstrb for EB_CS2_	R/W
14	STsu2	Bit to control Tsu for EB_CS2_	R/W
15	SThld2	Bit to control Thld for EB_CS2_	R/W
18:16	STrec23, STrec13, STrec03	Bits to control Trec for EB_CS3_	R/W
21:19	STstrb23, STstrb13, STstrb03	Bits to control Tstrb for EB_CS3_	R/W
22	STsu3	Bit to control Tsu for EB_CS3_	R/W
23	SThld3	Bit to control Thld for EB_CS3_	R/W
26:24	STrec24, STrec14, STrec04	Bits to control Trec for EB_CS4_	R/W

TABLE 7-20 Timing control register 2 (TCR2) bit definitions *(Continued)*

Bit #	Name	Description	R/W
29:27	STstrb24, STstrb14, STstrb04	Bits to control Tstrb for EB_CS4_	R/W
30	STsu4	Bit to control Tsu for EB_CS4_	R/W
31	SThld4	Bit to control Thld for EB_CS4_	R/W

Note – On reset all the bits are “1”.

7.5.2.3 Timing Control Register 3 (TCR3)

This register controls the Trec, Thld, Tsu and Thld for EB_CS5_ through EB_CS7_. TCR3 provides timing for devices which need extra long strobe width or recovery time. The encoding for the timing parameter is in TABLE 7-21.

TABLE 7-21 Encoding of timing parameters

Parameter programmed in TCR3	Actual Timing (# of clocks)
Tsun	Tsu+1
Thldn	Thld+1
Tstrbn	Tstrb+2
Trecn	Trec+ 3

TABLE 7-21 also provides timing with respect to actual parameters programmed in TCR3 register e.g. Tstrb value of “0” programmed in Tstrb bits of a particular device gives an actual Tstrb width of 2 clocks (0 + 2 = 2) on Ebus2 transfers to that device.

This register also provides a bit to choose between Level 1 and Level 2 DMA priority algorithms. TABLE 7-22 shows bit definitions for TCR3

TABLE 7-22 Timing control register 3 (TCR3) bit definitions

Bit #	Name	Description	R/W
4:0	STrec45, STrec35, STrec25, STrec15, STrec05	Bits to control Trec for EB_CS5_	R/W
7:5	STstrb25, STstrb15, STstrb05	Bits to control Tstrb for EB_CS5_	R/W
8	STsu5	Bit to control Tsu for EB_CS5_	R/W

TABLE 7-22 Timing control register 3 (TCR3) bit definitions *(Continued)*

Bit #	Name	Description	R/W
9	SThld5	Bit to control Thld for EB_CS5_	R/W
13:10	STrec36, STrec26, STrec16, STrec06	Bits to control Trec for EB_CS6_	R/W
17:14	STstrb36,STstrb26, STstrb16, STstrb06	Bits to control Tstrb for EB_CS6_	R/W
18	STsu6	Bit to control Tsu for EB_CS6_	R/W
19	SThld6	Bit to control Thld for EB_CS6_	R/W
24:20	STrec47, STrec37, STrec27, STrec17, STrec07	Bits to control Trec for EB_CS7_	R/W
28:25	STstrb37,STstrb27, STstrb17, STstrb07	Bits to control Tstrb for EB_CS7_	R/W
29	STsu7	Bit to control Tsu for EB_CS7_	R/W
30	SThld7	Bit to control Thld for EB_CS7_	R/W
31	PR	DMA priority select. 0 = Level 2	R/W

Note – On reset all the bits are “1”

7.5.3 DMA Registers

This section describes registers associated with each EBus2 DMA controller. The description is given for one DMA controller and is the same for the other three DMA controllers.

All unused register bits will be read as zero (RAZ). A write to these bits will have no effect.

7.5.3.1 DMA Control and Status Register (DCSR)

This register contains all control and status bits associated with the EBus2 DMA controller. The bit definitions are shown in TABLE 7-23.

TABLE 7-23 EBus2 DMA CSR Register

Bit #	Name	Description	R/W
0	INT_PEND	Set when EBus2 device associated with the DMA engine issues an interrupt or when the EBus2 DMA engine issues an interrupt	R
1	ERR_PEND	Set when a host bus error is detected	R
2	DRAIN	This bit is set when the FIFO data is being drained to memory	R
3		Reserved - Read as zero	
4	INT_EN	When set enables the interrupt to be generated when INT_PEND or ERR_PEND are set	R/W
5		Reserved - Read as zero	
6		Reserved - Read as zero	
7	RESET	When set invalidates the FIFO, resets the channel DMA engine. Software must write ‘0’ to clear	R/W
8	WRITE	DMA direction 0 — DMA Read from memory to device 1 — DMA Write from device to memory	R/W
9	EN_DMA	If set enables DMA function of the DMA engine	R/W
10	CYC_PENDING	When set, indicates a DMA cycle is currently active or pending. Not safe to clear RESET when this bit is set	R
11	DIAG_RD_DONE	In diagnostic mode when this bit is set, the DMA read is completed	R
12	DIAG_WR_DONE	In diagnostic mode when this bit is set, the DMA write is completed	R
13	EN_CNT	When set enables byte counter	R/W
14	TC	Terminal count; set when byte count has expired.Write ‘1’ to clear	R/W
15		Reserved - Read as zero	

TABLE 7-23 EBus2 DMA CSR Register *(Continued)*

Bit #	Name	Description	R/W
16	DIS_CSR_DRN	When set disables draining of FIFO on slave writes to DCSR.	R/W
17		Reserved - Read as zero	
19:18	BURST_SIZE	Defines transfer size to and from system memory: 00 — 16 bytes (4 word) 01 — 32 bytes (8 word) 10 — 4 bytes (1 word) 11 — 64 bytes (16 word)	R/W
20	DIAG_EN	When set, DMA diagnostic loopback mode is enabled	R/W
21		Reserved - Read as zero	
22	DIS_ERR_PEND	When set, an Error Pending condition will not cause an external interrupt or halt DMA activity	R/W
23	TCI_DIS	When set, disables TC from generating an interrupt. Defaults to 0	R/W
24	EN_NEXT	When set enables next address auto load mechanism. EN_CNT must also be set	R/W
25	DMA_ON	When set indicates that DMA engine is able to respond to DMA requests	R
26	A_LOADED	Address loaded. Set when DACR written or DNAR is copied to DACR	R
27	NA_LOADED	Next address loaded. Set when DNAR is written	R
31:28	DEV_ID	Device ID = 0xC	R

Note – On Reset all the register bits except ID will be “0” and CYC_PENDING will reflect the status of any pending requests.

DCSR Bit Function Notes

INT_PEND - Bit 0

Set when the EBus2 device associated with the DMA engine issues an interrupt, or when TC is set and TCI_DIS not set, Cleared otherwise.

ERR_PEND - Bit 1

DMA to/from the channel engine is stopped while this bit is set. ERR_PEND is reset by setting RESET.

DRAIN - Bit 2

When the FIFO is draining to memory this bit is set. Do not assert RESET or write to DACR register when set. DRAINING bit is not valid while ERR_PEND is set or during transfers to the EBus2 device. In these cases, the DRAINING bit should be ignored.

INT_EN - Bit 4

When set enables the interrupt to be generated when INT_PEND or ERR_PEND is set.

RESET - Bit 7

RESET will remain active once written as a one until written as a zero, unless cleared by a host Bus reset. Setting RESET or asserting host bus reset will invalidate the FIFO and reset all the channel engine state machines to their idle states. If ERR_PEND=0 when RESET is set, all dirty data in the channel engine FIFO will first be drained to memory. When this occurs, RESET must not be cleared until draining is complete, as indicated by DRAINING = 0. If ERR_PEND=1 when RESET is set, no draining will take place and all dirty data in the FIFO will be discarded.

WRITE - Bit 8

Controls the direction of the DMA transfer. ‘1’ indicates DMA write from a device to memory.

EN_DMA - Bit 9

This bit enables the DMA function of the DMA engine.

DIAG_RD_DONE - Bit 11

This bit gets set when in diagnostic read mode the DMA engine FIFO is full. This is cleared when DIAG_RD_EN bit gets cleared.

EN_CNT- Bit 13

It enables the EBus2 DMA engine byte counter to be used during DMA transfers.

TC - Terminal Count - Bit 14

The TC bit will be set when DBCR makes a transition from 0x000001 to 0x000000. When it is set, an interrupt will be generated, if enabled by INT_EN and not disabled by TCI_DIS.

When EN_NEXT= 0, TC is cleared by RESET or host bus reset. When EN_NEXT= 1, TC can also be cleared by writing a 1 to the TC bit itself.

DIAG_WR_DONE - Bit 15

This bit gets set when the host bus adapter tells the EBus2 DMA engine that it has transferred the 128th byte to the system memory. This is cleared by writing a “1” to the RESET bit.

DIS_CSR_DRN - Bit 16

When set disables the draining of FIFO on writes to DCSR.

BURST_SIZE - Bits 19:18

This field defines the transfer size used by the DMA engine for host bus transfers. All reads from memory will be one size, either 16, 8, 4, or 1 word. Writes to memory can be byte, half-word or one of the burst sizes given in the TABLE 7-24. The DMA engine will always use the largest possible size for writes, which is dependent on BURST_SIZE and the number of bytes that need to be drained. Also, BURST_SIZE determines the draining level of the FIFO. When the FIFO has been filled with this amount of data, it will always be drained to memory. The sizes given in the following table are in words. Where word is 4 bytes.

TABLE 7-24 Encoding for the BURST_SIZE bits

BURST_SIZE	Read Burst Size	Write Burst Sizes	FIFO Draining Level
00	4 words	4 words	4 words
01	8 words	8 words	8 words
10	1 word	1 word	1 word
11	16 words	16 words	16 words

TCI_DIS - Bit 23

When set, disables TC from generating an interrupt. Defaults to 0.

EN_NEXT - Bit 24

When set, enables next address auto loading mechanism.

DMA_ON - Bit 25

Reads as 1 when (A_LOADED or NA_LOADED) & EN_DMA & NOT (ERR_PEND); otherwise reads as 0. When set, indicates that DMA engine is able to respond to DMA requests from the EBus2 device.

A_LOADED & NA_LOADED - Bits 26 and 27.

These bits define the validity of the values stored in the DACR and DNAR registers. A_LOADED is set when DACR is written directly or when DNAR is copied to DACR, and is reset by RESET or DBCR expiring.

NA_LOADED is set when DNAR is written. It is reset by RESET, EN_NEXT = 0, or DNAR being copied to DACR.

When the state is reached where a valid DNAR has been loaded and the current DACR has been marked as invalid (NA_LOADED = 1 & A_LOADED = 0), then the contents of the DNAR register are copied to the DACR register. The copy takes place on the same clock edge where the stated condition is sampled as true. If address chaining has been set up to take place when the byte count expires, the actual sequence of events will be the following: First, A_LOADED will be cleared (on expiration of byte count). Second, the DNAR will be copied to the DACR register. These two events will probably be seen as one by software.

DEV_ID - Bits 31:28

These bits give device ID for the DMA device.

7.5.3.2 DMA Address Count Register (DACR) and DMA Next Address Register (DNAR)

TABLE 7-25 DACR and DNAR bits

Bit	Mnemonic	Description	Type
31:0	DACR	Address Counter	R/W
31:0	DNAR	Next Address Register	W

The value in these registers after a RESET is indeterminate.

The Address Register is a 32-bit loadable counter which always points to the next byte that will be accessed in the memory.

If the EN_NEXT (enable next address) bit in the DCSR is set, then a write to the DACR register will write to the DNAR register instead. If EN_NEXT is set when the byte counter (DBCR) expires, and the DNAR register has been written since the last time the byte counter expired, then the contents of DNAR are copied into DACR. If EN_NEXT is set when the byte counter (DBCR) expires, but the DNAR register has not been written since the last time the byte counter expired, then DMA activity is stopped and DMA requests from the EBus2 device will be ignored until DNAR is written, or EN_NEXT is cleared. (Also, the DMA_ON bit will read as 0 while DMA

is stopped because of this). When DMA is re-enabled by writing to the DNAR register, the contents of DNAR are copied into DACR before DMA activity actually begins.

If the DNAR register is written before the DACR register has been written, the address written to DNAR will immediately be copied into DACR. When this occurs, it also causes the value in the DNBR register to be copied into the DBCR register. This allows for a shortcut in loading both DACR and DNAR along with DBCR and DNBR by writing the registers in the following sequence: DNBR, DNAR, DNBR, DNAR. When the first value is written into DNAR, it is immediately copied into DACR since DACR hasn't been loaded yet. This causes the first value that was written into DNBR to be copied into DBCR. The second values written into DNBR and DNAR then remain there as the actual next address and byte count. This allows the loading of both the current and next address and byte count registers without having to write the DCSR to change the EN_NEXT bit in between.

Note – A write to the DACR register will invalidate the FIFO. A write to the DNAR register does not have this effect.

7.5.3.3 DMA Byte Count Register (DBCR) and DMA Next Byte Register (DNBR)

TABLE 7-26 DBCR and DNBR bits

Bit	Mnemonic	Description	Type
23:0	DBCR	Byte Count; counts down to 0, then sets the TC bit in the DCSR	R/W
23:0	DNBR	Next Byte Count Register	W

The value in these registers after a RESET is indeterminate.

When reading this register as a word, bits 31:24 will read as 0's.

When enabled, the Byte Counter is decremented every time a byte is transferred between the DMA engine and the EBus2 device in a EBus2 DMA cycle. It is decremented immediately after the byte has been transferred. It is not decremented on slave accesses to the EBus2 or on transfers between the DMA engine and system memory.

If the EN_NEXT bit in the DCSR is set, then a write to the DBCR register will write to the DNBR register instead. Whenever the DNAR register is copied into the DACR register, the DNBR register is copied into the DBCR register at the same time.

If DNAR is being copied into DACR and DNBR has not been written since the last time DNBR was copied into DBCR, the last value that was written into DNBR will again be copied into DBCR. This provides a shortcut in setting up consecutive DMA transfers of equal size from different addresses, in that DNBR only needs to be written once as long as DNAR is loaded for each successive transfer.

If EN_NEXT is not set when DBCR expires (changes from 0x000001 to 0x000000), then DMA activity between the EBus2 device and the DMA engine will be stopped and the DMA_ON bit will read as 0 until DACR is written.

7.6 Programming Notes

7.6.1 Timing Control Register Programming

The OBP at the boot time program values in the TCR1, TCR2 and TCR3 registers according to timing and the nature of the EBus2 devices supported in a particular system.

Here is an example of how timing control registers can be programmed for a particular system. Let us consider that there is a system which is using 33 Mhz PCI bus and EBus2 channel engine internal clock is running at the same frequency as PCI bus i.e 33 Mhz (cycle time = ~30 ns). Let us further assume that there are 7 slave devices and 4 DMA devices connected to the EBus2. OBP has to program the following functions for this system:

7.6.1.1 Slave Cycle Time Programming

The Timing control is provided for 7 chip selects. These chip selects correspond to EB_CS1_ through EB_CS7_ of TABLE 7-1. Here is an example of how to program a slave cycle for a device “x” connected to EB_CS1_. Let us assume that the device “x” has the following timing parameters (refer to FIGURE 7-2 for the definition of these parameters):

Tsu = 20 ns

Thld = 0

Tstrb = 75

Trec = 70 ns

To program Tsu for chip, select 1, bit 6 (STsu1) of TCR2 should be programmed as “0” which gives a Tsu = 30 ns (1 EBus2 clock).

To program Thld for chip select 1, bit 7(SThld1) of TCR2 should be programmed as “0” which gives a Thld = 30 ns (1 EBus2 clock).

To program Tstrb for address range 1, bits 5:3 (STstrb21, STstrb11, STstrb01) of TCR2 should be written as “001” which gives a Tstrb = 90 ns (3 EBus2 clocks).

To program Trec for chip select 1, bits 2:0 (Trec21,Trec11,Trec01) of TCR2 should be programmed as “000” which gives a Trec = 90 ns (3 EBus2 clocks).

7.6.1.2 DMA Priority Programming

Let us suppose that one of the DMA channels in our example system has a much higher data rate than the other three channels, so we will attach that device to DMA controller 1 and program the priority algorithm as level 2. The DMA controller 1 will potentially get every second DMA grant by the EBus2 internal priority controller. To program the level 2 priority bit 31 (PR) of TCR3 should be programmed as “0”.

7.6.1.3 DMA Cycle Time Programming

The Timing control is provided for all four DMA controllers. Here is an example of how to program DMA controller 1 which talks to a slave DMA device on the EBus2, which has the following timing parameters:

Tsu = 40 ns

Thld = 20 ns

Tstrb = 100 ns

Trec = 100 ns

To program Tsu for DMA controller 1, bit 6 (DTsu1) of TCR1 should be programmed as “1” which gives a Tsu = 60 ns (2 EBus2 clocks).

To program Thld for DMA controller 1, bit 7(DThld1) of TCR1 should be programmed as “0” which gives a Thld = 30 ns (1 EBus2 clock).

To program Tstrb for DMA controller 1, bits 5:3 (DTstrb21, DTsrrb11, DTstrb01) of TCR1 should be written as “010” which gives a Tstrb = 120 ns (4 EBus2 clocks).

To program Trec for DMA controller 1, bits 2:0 (DTrec21, DTrec11, DTrec01) of TCR1 should be programmed as “001” which gives a Trec = 120 ns (4 EBus2 clocks).

7.6.2 DMA Register Programming

7.6.2.1 To set up a transfer to or from the EBus2 device using the DMA engine

The transfer is set up by programming the internal byte count and address registers of the DMA engine. To ensure that no error bits are set in the DCSR, the software driver issues a RESET command. The driver then programs the WRITE, INT_EN, EN_CNT, and TCI_DIS bits in the DCSR. These map to the following usages:

WRITE:	determines direction of transfer
EN_CNT:	set to override the EBus2 device byte count register and use that of DMA engine
TCI_DIS:	set to disable interrupts upon byte count = 0.
INT_EN:	set to enable interrupts upon error conditions and byte count = 0.

The EBus2 device is then programmed with the appropriate data for the particular transfer. The EN_DMA bit is then set. This bit acts as a gate such that the DMA engine will immediately begin to respond to EBus2 device requests for service.

The transfer will complete with one of three results: an error, in which case the driver will poll the EBus2 device or the DMA engine for status, an interrupt, for which the driver must provide service, or expiration of the byte count register.

7.6.2.2 To stop a transfer to or from the EBus2 device using the DMA engine

The driver may suspend transfers between the EBus2 device and DMA engine at any time by simply clearing the EN_DMA bit. The transfer is easily restarted by again setting the EN_DMA bit.

7.6.2.3 Use of Internal Byte Counter with Next Address feature disabled

When using the internal Byte Counter and the TC flag in the DCSR with EN_NEXT = 0, it is necessary to perform the following procedure for correct operation:

Load Byte Count into DBCR

Load DMA address into DACR

Load EBus2 Device with relevant command(s)

Load DCSR with enables and direction bits (EN_DMA, EN_CNT, and WRITE)

Data will be transferred as directed until the Byte Count expires, at which point the TC flag will be set in the DCSR and an interrupt will be generated, if enabled. DMA will also be stopped at this time (DMA_ON will be cleared). DMA will remain stopped, independent of the value of EN_DMA, until DACR is loaded with a new value.

Note – This implies that the interrupt service routine should clear EN_DMA before writing a new address to DACR.

7.6.2.4 Use of Internal Byte Counter with Next Address feature enabled

When using the internal Byte Counter, the TC flag in the DCSR and the NEXT ADDRESS feature, it is necessary to perform the following procedure for correct operation:

Note – This is a suggested procedure since several methods of programming the chip are possible.

INITIALIZATION

Initialization if the state of the chip is not defined such as after an error.

Write Control register

TCI_DIS, EN_DMA, EN_NEXT, RESET = 0

EN_CNT, INT_EN = 1

WRITE = value as desired transfer direction

MULTIPLE BLOCK TRANSFERS

To do a multiple block transfer with an interrupt after each block:

Write Control register

INT_EN, EN_DMA, EN_CNT, EN_NEXT = 1

TCI_DIS, RESET = 0

WRITE = value as read or desired

Write DBCR with byte count of the first block.

Write DACR with the starting address of the first block. Set-up and start the EBus2 device chip to do its transfer.

Write DNBR with the byte count of the “next” block.

Loading of the DNBR is optional because the initial loading of the DBCR also loads the next count register.

Write DNAR with the address of the “next” block. The transfer of the first block is enabled and the transfer of the next block will happen automatically when Terminal Count is reached, i.e.: the next address and byte counts will be used. (This assumes the loading of the NEXT count and address occur before the first block transfer is complete.)

INTERRUPT

After each interrupt:

Read DCSR

TC and DMA_ON should both = 1.

If DMA_ON = 0 then, the DNAR register did not get updated or there is an error pending, D_ERR_PEND. (It could also mean the DMA is not enabled, EN_DMA, or the next address feature is not enabled, EN_NEXT, but they were set = 1 so this should not be the case.)

Write DNBR with the byte count of the’next’ block.

Loading of the DNBR register is optional, but if loaded, must be loaded before the DNAR register.

Write DNAR with the address of the’next’ block.

LAST BLOCK

If no Terminal Count interrupt is desired after the last block is transferred: (because we expect an interrupt from the EBus2 device chip at the end of the transfer.)

On interrupt of the next to last block:

Read DCSR

TC and DMA_ON should both = 1.

Write DCSR

TCI_DIS, INT_EN, EN_DMA, EN_CNT, EN_NEXT = 1

RESET = 0

WRITE = value as read or desired

The DNAR register is not loaded so the transfer will stop.

If terminal count interrupt is expected from the DMA engine then do the above procedure except TCI_DIS = 0.

NEXT TRANSFER

The initialization for the next multi-block transfer is:

Write DCSR

INT_EN, EN_DMA, EN_CNT, EN_NEXT = 1

TCI_DIS, RESET = 0

WRITE = value as read or desired

This assumes the chip is in a known state because, if not, the write to the DBCR and DACR registers may go to the NEXT registers (See INITIALIZATION).

Write DBCR with byte count of the first block.

Write DACR with the starting address of the first block.

Set-up and start the EBus2 device chip to do its transfer.

Write DNBR with the byte count of the'next' block.

Loading of the DNBR is optional because the initial loading of the DBCR also loads the next count register

Write DNAR with the address of the'next' block.

TAKING ADVANTAGE OF 'NEXT' FEATURE

If an interrupt occurs and the 'next' address is not available, there is no room in the buffer or data is not available, the processor can take advantage of the fail-safe capability by not loading the DNAR and the DMA will stop when the count goes to 0.

After the 'next' address becomes known the processor must determine if the transfer of the current block has completed. If it has not, the processor should load the "next" count and address to continue the transfer. If the transfer has completed, the processor should load the current count and address and the next count and next address to restart the transfer.

Notes –
There will be an interrupt generated when the count goes to zero.

The firmware must understand that if the transfer has stopped because the DNAR was not loaded before the terminal count was reached, it is effectively loading the DBCR and DACR registers, not the NEXT registers. This will enable the DMA to restart the transfer.

It is necessary to load the count before the address because the loading of the address register causes the restart of the transfer.

Interrupts from the EBus2 device are visible as INT_PEND in the DCSR. The INT_EN bit is provided to enable or disable the generation of an interrupt. If an error condition exists during a memory access the ERR_PEND bit will be set. This will cause an interrupt (if enabled). Similarly, expiration of the Byte Counter will cause the INT_PEND bit to be set and an interrupt (if enabled). The ERR_PEND bit can only be cleared by a RESET command, or host bus reset.

Software should never set the RESET bits in the DCSR or write to the DACR register while the EN_DMA bit is set or while the DRAINING bit is set.

7.7 Timing Diagrams

7.7.1 EBus2 Slave Cycles

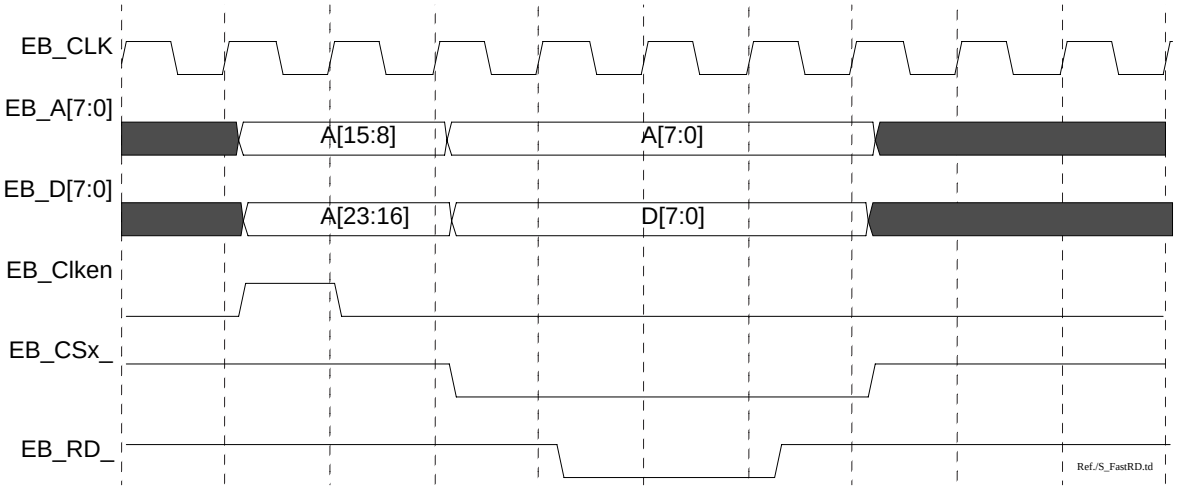

FIGURE 7-3 EBus2 slave read cycle

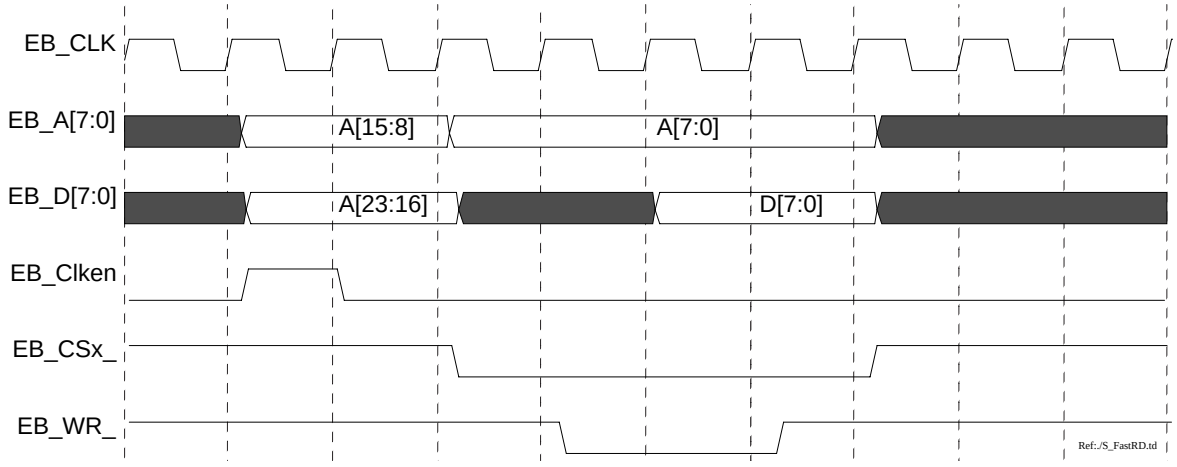

FIGURE 7-4 EBus2 Slave write cycle

7.7.2 Ebus2 DMA Cycles

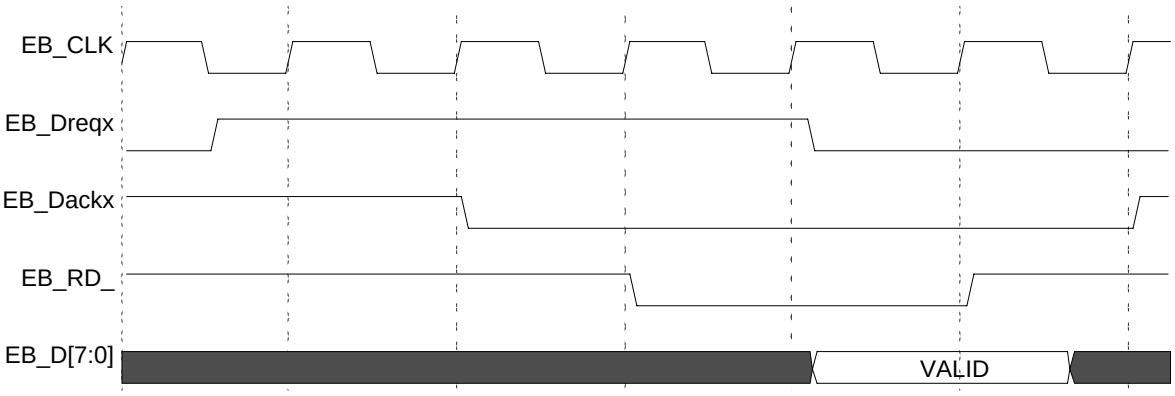

FIGURE 7-5 EBus2 DMA read cycle

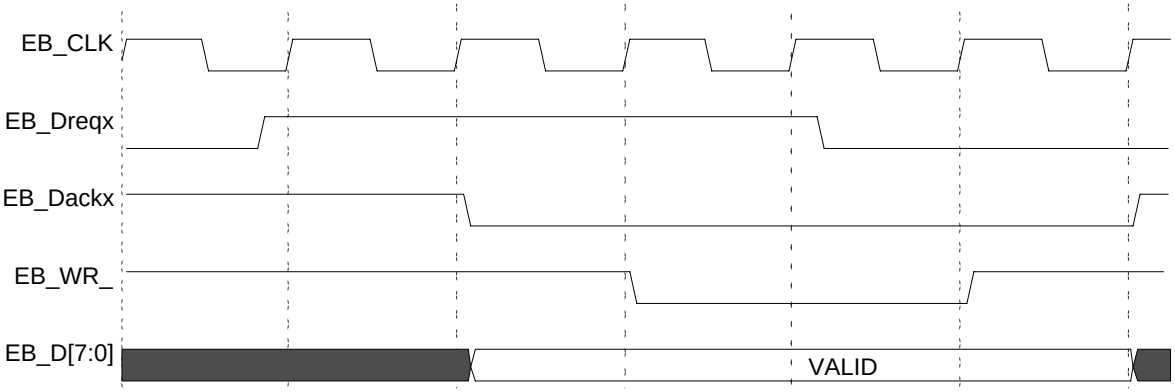

FIGURE 7-6 EBus2 DMA write cycle

CHAPTER 8

Clock and Scan Control

8.1 Introduction

This section describes the testability features implemented in PCIO. These features have been incoporated to provide a structured test approach to both ASIC fabrication testing and board-level testing and debug.

8.2 Test and Debug Modes

PCIO provides several different test modes which can be used to verify the correct functional behavior of the internal logic and I/O pad ring. These modes are accessible via a JTAG compliant TAP controller. For more information on the JTAG standard, refer to IEEE standard 1149.1-1990.

8.2.1 Boundary Scan Modes

PCIO utilizes boundary scan to provide control and visibility of the I/O pins during manufacturing test. This control includes the ability to tristate the output pins. All boundary scan modes required by the JTAG standard are supported by the TAP controller.

8.2.2 ATPG Mode

The ATPG scan chain provides a means to apply test pattern vectors to detect stuck-at faults in the internal logic and is accessed by selecting the “atpg” instruction of the TAP contoller. In this mode, the internal and boundary scan chains are muxed together to form the ATPG chain. A single internal clock pulse is generated when exiting the Capture-DR TAP state to capture the test vector result.

While the “atpg” instruction is selected, both the chip inputs and outputs are defined by the contents of the boundary scan register.

ATPG vectors can also be applied via the “intscan” instruction. This instruction also generates a clock pulse upon exiting Capture-DR but only the internal scan chain is selected. I/O stimuli is driven and read by the tester instead of from the boundary scan register. This instruction can be used for testers that have to serialize scan vectors. Since the boundary scan chain no longer has to be shifted, fewer tester vectors are required to apply the vector suite.

8.2.3 Debug modes

8.2.3.1 Dumping internal state

The internal scan chain can also be selected via the “debug” instruction. This instruction is provided to allow for non-destructive internal node visibility during lab debug. No capture clock is issued for this instruction.

While the “debug” instruction is selected, both the chip inputs and outputs are defined by the contents of the boundary scan register.

8.2.3.2 Clock Controller

The clock controller deterministically stops PCIO’s internal clocks upon the occurrence of an external event. The clock controller can only be accessed via the ccr scan chain. This chain is selected via the “sel_ccr” instruction.

The clock controller consists of a stop enable bit and three synchronizers, one for each of PCIO’s clock domains. When the stop enable bit is set, the stop_clock signal will switch the source of the internal clocks from the clock pins to the JTAG controller. This clock source switching is synchronized to the rising edge for each clock domain.

8.3 JTAG Controller

The JTAG controller consists of the following elements:

- TAP controller and decode logic
- Scan datapath

The JTAG controller allows the various test modes to be activated via the standard JTAG signals (TCK, TMS, TDI, TRSTB and TDO). Each test mode may be activated by shifting the appropriate instruction into the JTAG Instruction Register. The test mode selected becomes active upon the TAP state machine leaving the Update-IR state. When the JTAG controller is placed into its Test-Logic-Reset state, PCIO will return to its normal operating mode.

8.3.1 Control logic

The JTAG controller utilizes a custom TAP controller to instantiate the JTAG state machine. The outputs of the TAP controller are then combined with the outputs of the instruction register decoder to generate the test mode select and test clock signals.

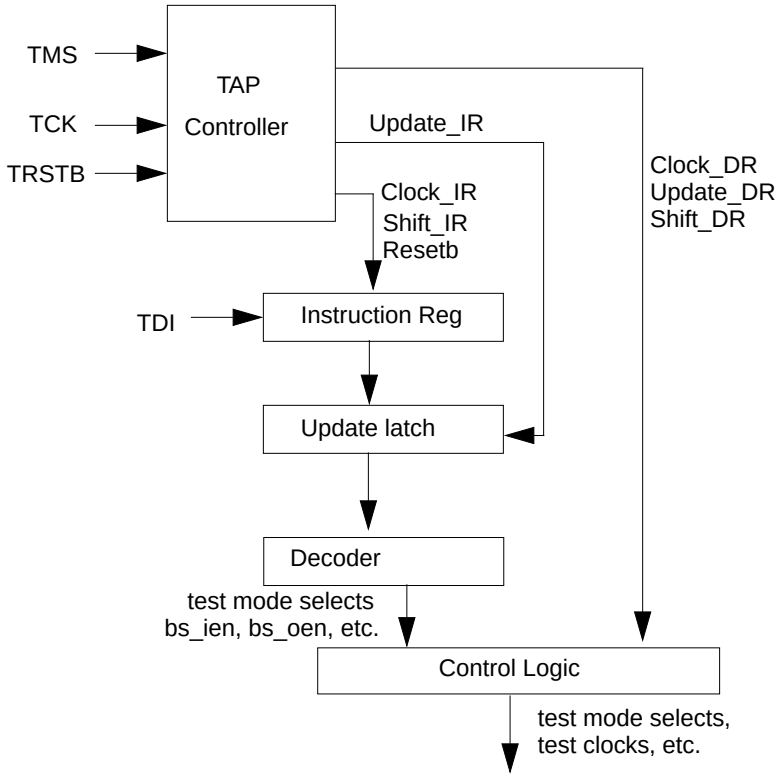

FIGURE 8-1 JTAG Logic Block Diagram

8.3.2 Scan Data Paths

The scan data paths are the various scan chains that are selected during a selected test mode. The scan chains are as follows:

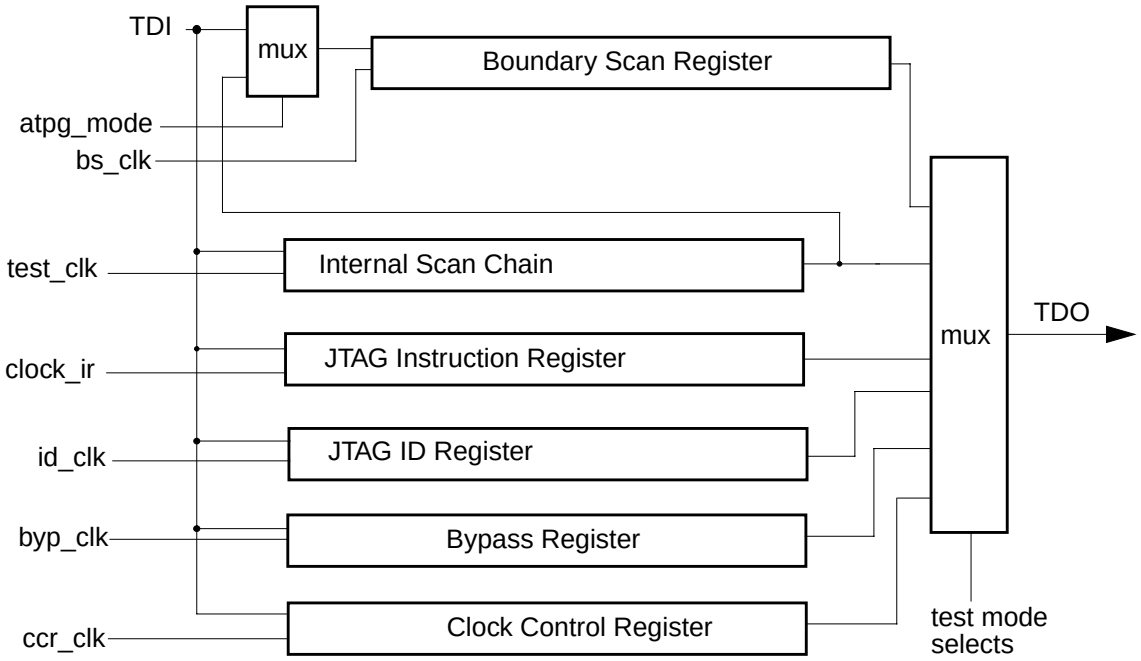

FIGURE 8-2 PCIO Scan Registers

All of the scan datapaths can operate at clock speeds up to 10 MHz.

The following table lists the correct scan chain lengths for each scan chain.

TABLE 8-1 Scan Chain Lengths

Chain	Length
boundary	248
internal	5272
atpg	boundary+internal
bypass	1

TABLE 8-1 Scan Chain Lengths *(Continued)*

Chain	Length
id	32
instruction	4
ccr	1

8.3.3 JTAG Instructions and ID

TABLE 8-2 JTAG ID fields

Version	Part Number	Manufacturer Identity	LSB
0x1	0x1791	0x022	0x1

The following instructions are supported by the PCIO TAP. The table contains the bit-value and mnemonic for each instruction and the scan chain accessed. Instructions marked by "*" are required by the IEEE JTAG standard.

TABLE 8-3 JTAG Instructions

Value	Instruction	Scan Chain	IMC	OMC	BCAP	ICAP
0000	extest*	boundary	0	1	1	0
0001	sample*	boundary	0	0	1	0
0010	intscan	internal	0	0	0	1
0011	atpg	atpg	1	1	1	1
0100	debug	internal	1	1	0	0
0101	reserved	bypass	0	0	0	0
0110	clamp	bypass	1	1	0	0
0111	intest	boundary	1	1	1	0
1000	reserved	bypass	0	0	0	0
1001	scsi_test	bypass	0	0	0	0
1010	reserved	bypass	0	0	0	0
1011	reserved	bypass	0	0	0	0
1100	sel_ccr	ccr	0	0	0	0

TABLE 8-3 JTAG Instructions *(Continued)*

Value	Instruction	Scan Chain	IMC	OMC	BCAP	ICAP
1101	reserved	bypass	0	0	0	0
1110	idcode	id	0	0	0	0
1111	bypass*	bypass	0	0	0	0

* IMC 1 = core driven by BS cell, 0 = core driven by pin
* OMC 1 = pin driven by BS cell, 0 = pin driven by core
* BCAP 1 = Capture clock generated for BS cell, 0 = no clock
* ICAP 1 = Capture clock generated for internal flops, 0 = no clock