

V350EPC, V360EPC

Local Bus to PCI Bridge

User's Manual

Revision 1.05

©V3 Semiconductor

V3 Semiconductor makes no warranties for the use of its products. V3 does not assume any liability for errors which may appear in this document, however, we will attempt to notify customers of such errors.

V3 Semiconductor retains the right to make changes to either the documentation, specification or component without notice. Please verify with V3 Semiconductor to be sure you have the latest specifications before finalizing a design.

©V3 Semiconductor 1997-1998

The Embedded Chipset Company is a trademark of V3 Semiconductor.
All other trademarks are the property of their respective owners.

Contents

Chapter 1	Introduction	1
1.1	How to Use this Manual.....	3
1.2	Getting Help from V3 Semiconductor.....	3
1.3	Getting Answers to PCI Related Questions.....	4
1.4	Getting Information About the i960/Am29K Family	4
1.5	Disclaimer.....	5
1.6	Revision History.....	5
Chapter 2	Bridge Operation Overview	7
2.1	Operational Example.....	8
2.1.1	Direct Local Bus Write to PCI Space.....	9
2.1.2	Direct Local Bus Read from PCI Space	9
2.1.3	PCI Write to Local Space	10
2.1.4	PCI Reads from Local Space	10
2.1.5	DMA Transfers	11
2.1.6	Mailbox Registers and Doorbell Interrupts	11
Chapter 3	Internal Register Apertures	13
3.1	Local Bus Access to Internal Registers.....	14
3.2	PCI Bus Access to Internal Registers.....	14
Chapter 4	Data Transfer Apertures	17
4.1	PCI-to-Local Bus Apertures.....	17
4.1.1	Setting the PCI-to-Local Aperture Base Address and Size	18
4.1.2	Selecting PCI Memory or I/O Space Mapping.....	21
4.1.3	PCI-to-Local Address Translation.....	21
4.1.4	Byte Order Conversion.....	21
4.1.5	Enabling Read Prefetching.....	22
4.1.6	Disabling PCI-to-Local Bus Apertures	22
4.1.7	Overlapping Apertures.....	22
4.1.8	Special Function Modes for PCI-to-Local Bus Apertures	23
4.2	Local-to-PCI Bus Apertures.....	23
4.2.1	Setting the PCI Command Type.....	23
4.2.2	Setting the Local-to-PCI Aperture Base Address and Size	25
4.2.3	Local-to-PCI Address Translation.....	25
4.2.4	Byte Order Conversion.....	25

4.2.5	Enabling Read Prefetching.....	25
4.2.6	Enabling Local-to-PCI Bus Apertures.....	25

Chapter 5 FIFO Architecture and Operation 27

5.1	Dynamic Bandwidth Allocation FIFO Architecture.....	27
5.2	Write FIFO Operation and Programming.....	28
5.2.1	Write FIFO Draining Strategies	29
5.3	Read FIFO Operation and Programming	31
5.3.1	Prefetching and Read FIFO Filling Strategies	31
5.4	FIFO Prioritization Options	33
5.5	FIFO Data Coherency Options	33
5.5.1	Ensuring Strict Data Coherency	34
5.5.2	Monitoring the Status of Read and Write FIFOs.....	34
5.5.3	Ensuring the Completion of a Posted Write.....	35
5.6	FIFO Latency	35

Chapter 6 DMA Controller 37

6.1	DMA Transfers	37
6.1.1	Local Bus to PCI Bus DMA Transfers	37
6.1.2	PCI Bus to Local Bus DMA Transfers	38
6.1.3	DMA Block Chaining.....	38
6.1.4	DMA Transfer Size	40
6.1.5	Relationship to the Data Transfer Apertures	40
6.1.6	Automatic DMA Throttling.....	40
6.1.7	DMA Interrupts	40
6.2	Programming the DMA Controller	41
6.2.1	Setting the Starting Addresses	41
6.2.2	Setting the Transfer Count	42
6.2.3	Setting the Transfer Direction.....	42
6.2.4	Byte Order Conversion	42
6.2.5	Using DMA Block Chaining	43
6.2.6	Starting DMA Operation	43
6.2.7	Early Termination of a DMA Process	43
6.2.8	Setting Priority Between the DMA Channels	43

Chapter 7 PCI Bus Interface 45

7.1	Target Transfers.....	45
7.1.1	Target Reads.....	45
7.1.2	Target Writes	46
7.1.3	PCI Exceptions During EPC Target Cycles.....	47

7.1.3.1	Recoverable Exception: Target Disconnect	47
7.1.3.2	Recoverable Exception: Target Retry	47
7.1.4	PCI Access of EPC Internal Registers	47
7.2	Initiator Transfers.....	48
7.2.1	Initiator Reads	48
7.2.2	Initiator Writes.....	49
7.2.3	PCI Exceptions During EPC Initiated Cycles.....	58
7.2.3.1	Fatal Exception: Master Abort (Reads)	58
7.2.3.2	Fatal Exception: Master Abort (Writes)	58
7.2.3.3	Fatal Exception: Target Abort (Reads)	58
7.2.3.4	Fatal Exception: Target Abort (Writes)	59
7.2.3.5	Recoverable Exception: Target Disconnect	59
7.2.3.6	Recoverable Exception: Target Retry	59
7.2.4	Initiator Pre-Emption.....	59

Chapter 8 Local Bus Interface 61

8.1	Target Mode	61
8.1.1	Local Bus CPU Configuration	61
8.1.2	Local Reads and Writes to Internal Registers	61
8.1.3	Local Read from Local-to-PCI Apertures.....	64
8.1.4	Local Write to Local-to-PCI Apertures	64
8.1.5	Target Mode PCI Error Signalling.....	73
8.1.6	Deadlock Conditions and Resolution.....	73
8.2	Master Mode.....	74
8.2.1	Requesting the Local Bus.....	74
8.2.2	i960 Local Bus Reads and Writes	74
8.2.3	Am29K Local Bus Reads and Writes	74
8.2.3.1	Strict Compatibility Mode.....	75
8.2.3.2	High-Performance Mode	75
8.3	Burst Support.....	76
8.4	$\overline{\text{BTERM}}$ Operation (V961EPC and V962EPC Only).....	77
8.4.1	$\overline{\text{BTERM}}$ as an Input	77
8.4.2	Deadlock Avoidance using the $\overline{\text{BTERM}}$ as an Output	78
8.5	Local Bus Parity.....	79
8.5.1	Relationship between Local Parity and PCI Parity	79
8.5.2	Local Bus Parity Generation	80
8.5.3	Local Bus Parity Checking.....	80

Chapter 9 PCI Configuration 83

9.1	Configuration as a System Host Bridge.....	83
9.1.1	EPC Host Configuration Mechanism	83

9.1.2	Controlling Target IDSEL Lines	83
9.1.3	Generating Configuration Reads and Writes	84
9.1.4	Using Configuration Information	84
9.1.5	Determining the Presence of Target Devices During Configuration	85
9.2	Configuration as a Target Bridge	85
9.2.1	EPC Base Register Response to Configuration Inquiries	85
9.2.2	EPC Expansion ROM Base Register Response to Configuration Inquiries	87

Chapter 10 PC Compatibility **89**

10.1	Real Mode DOS Compatibility Aperture	89
10.2	Example: VGA Peripheral	91

Chapter 11 Mailbox Registers **93**

11.1	Overview	93
11.1.1	Accessing the Mailbox Registers	93
11.1.2	Doorbell Interrupts	94
11.2	Programming the Mailbox Registers	95
11.2.1	Enabling Doorbell Interrupt Requests	95
11.2.2	Clearing Doorbell Interrupt Requests	95

Chapter 12 Interrupt Control **97**

12.1	Local Interrupt Control Unit	97
12.1.1	Overview	97
12.1.2	Local Interrupt Requests	98
12.1.3	Masking Local Interrupt Requests	98
12.1.4	Local Interrupt Event Signal	98
12.2	PCI Interrupt Control Unit (PICU)	99
12.2.1	Overview	99
12.2.2	PCI Interrupt Pins (\overline{INTA} through \overline{INTD})	100
12.2.2.1	Configuring a PCI Interrupt Pin as an Interrupt Request Output	100
12.2.2.2	Configuring a PCI Interrupt Pin as an Interrupt Request Input	100
12.2.2.3	Crosspoint Interrupt Routing Mechanism	100
12.2.3	Internal PCI Interrupt Requests	101
12.2.3.1	Mailbox and DMA PCI Interrupt Requests	101
12.2.3.2	Local Direct Interrupt Request	102
12.2.3.3	Routing the Internal PCI Interrupt Requests to an INTx Pin	102
12.2.4	PICU Configuration Example	102
12.3	Generating PCI Interrupt Acknowledge Cycles	103

Chapter 13 Initialization	105
13.1 Reset Direction	105
13.2 Initializing the Internal Registers.....	108
13.2.1 Initialization Using the Local Processor	108
13.2.1.1 i960 Processor Configuration Note	110
13.2.2 Initialization Using the PCI Configuration Space	110
13.2.3 Initialization Using the Serial EEPROM interface	111
13.2.3.1 Programming the Serial EEPROM	111
13.2.3.2 Timing Considerations when Initializing via the Serial EEPROM.....	112
13.2.4 Re-Initialization Using the PCI I/O or Memory Space.....	113
Chapter 14 Register Descriptions	115
14.1 Register Map	115
Glossary	143
Index	145

Chapter 1

Introduction

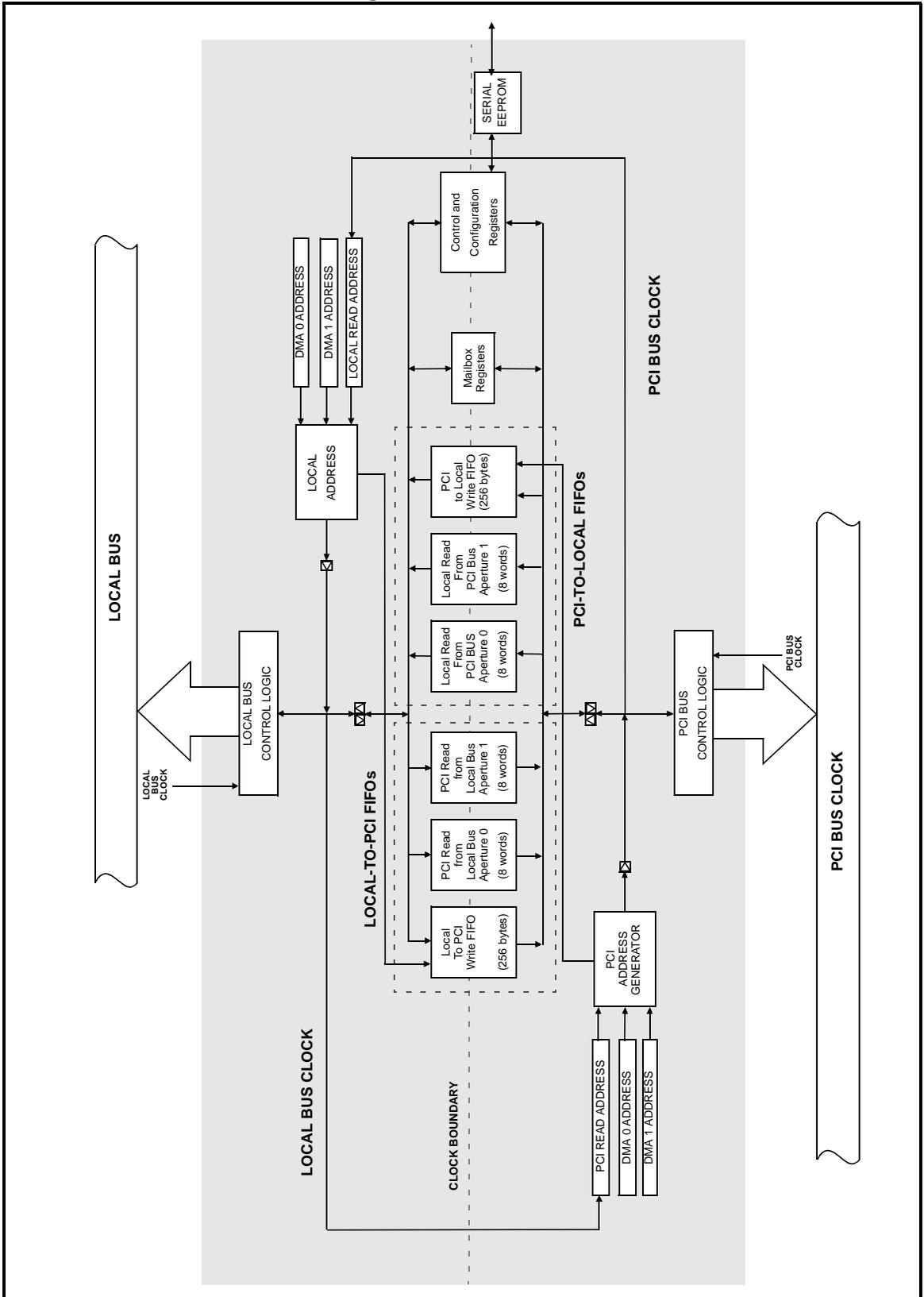
In a very short period of time the PCI bus standard has moved beyond the PC to become the most widely accepted high-performance bus standard for embedded applications. As a leader in providing chipset solutions for high-end embedded applications, V3 Semiconductor has developed the EPC family of PCI Bridge Components for the Intel i960® and the AMD Am29K™ processor family. The EPC is specifically designed to take advantage of the key features of the i960/Am29K family of processors to deliver the highest performance possible for embedded PCI systems.

The EPC is the new enhanced version of the previous generation of PCI Bridges known as the PBC. The V350EPC is backward compatible (register and pin) with the V960PBC and the V961PBC devices. The V360EPC is backward compatible (register and pin) with the V962PBC and the V292PBC. A block diagram of the EPC is shown in Figure 1.

Some of the key features of the EPC are:

- Glueless interface to i960/Am29K processors
- Compliant with PCI 2.1 specification
- PICMG CompactPCI Hot Swap Capable
- Configurable for system host, bus master, and target operation
- Burst access support on both local and PCI interfaces
- PCI-to-Local and Local-to-PCI address space remapping
- 2 PCI-to-Local and 2 Local-to-PCI data transfer apertures
- 640 bytes of programmable FIFO storage with *DYNAMIC BANDWIDTH ALLOCATION™*
- On-the-fly byte order (endian) conversion
- 2 channel DMA controller with DMA Chaining
- Bi-directional mailbox registers with doorbell interrupts
- Power on configuration via serial EEPROM
- Direct connect to VxBMC/CMC or V96SSC memory controllers

Figure 1: EPC BLOCK DIAGRAM



1.1 HOW TO USE THIS MANUAL

The EPC User's Manual includes detailed information regarding the programming and design of systems based on the EPC. However there are other complementary documents to assist designers. A complete set of documentation includes the following:

V350EPC

- EPC User's Manual (this document)
- V350EPC Data Sheet
- Reference design manual (Quatro, Avalanche)

V360EPC

- EPC User's Manual (this document)
- V360EPC Data Sheet
- Reference design manual (Quatro, Avalanche)

The electrical specifications are found in the V350EPC, V360EPC Data Sheets. Before finalizing a system design based on the EPC, please contact V3 Semiconductor to verify that you have the most recent specifications. Other application notes and useful design aids can be found on the V3 Web site.

V3 is constantly trying to improve the quality of its product documentation. If you have any questions or comments, please contact V3 Customer Assistance.

1.2 GETTING HELP FROM V3 SEMICONDUCTOR

If you need assistance with a technical question, please contact V3 through our AppsFax or Email hotlines. Email is the quickest and most efficient way to get technical support from V3.

If you do not have access to Email or a fax machine, feel free to call us from 9AM to 5PM Pacific Standard Time.

V3 AppsFax: (408) 988-2601 (Santa Clara, California)

V3 Customer Assistance: (800) 488-8410 (US and Canada)

(408) 988-1050 (Outside North America)

v3help@vcubed.com

Introduction

Getting Answers to PCI Related Questions

Some technical support information is also posted on the V3 Web site. This is the source of the most up-to-date data sheets and user's manuals and is located at:

www.vcubed.com

1.3 GETTING ANSWERS TO PCI RELATED QUESTIONS

This manual assumes a basic understanding of the PCI bus specification. If you are looking for a copy of the specification please contact the PCI special interest group at 800.433.5177.

If you are not intimately familiar with the PCI specification, a good place to start is by reading one of several books on the subject. One of the most popular is "PCI System Architecture" written by Tom Shanley and Don Anderson (published by MindShare Inc.).

1.4 GETTING INFORMATION ABOUT THE i960/Am29K FAMILY

This manual assumes that you are familiar with the i960/Am29K processors for which the EPC was designed. For information about products, you may call Intel / Advanced Micro Devices at the numbers listed below.

Intel: (800) 879-4683 (US and Canada)
(408) 987-8080 (Outside North America)
www.intel.com (Web site)

AMD: (800) 222-9323 (US and Canada)
(408) 749-5703 (Outside North America)
www.amd.com (Web site)

1.5 DISCLAIMER

V3 Semiconductor makes no warranties for the use of its products. V3 does not assume any liability for errors which may appear in this document, however, we will attempt to notify customers of such errors. V3 Semiconductor retains the right to make changes to either the

documentation, specification or component without notice.

Please verify with V3 Semiconductor to be sure you have the latest specifications before finalizing your design.

1.6 REVISION HISTORY

Table 1: Revision History

Revision Number	Date	Comments
0.2	2/98	First release without NDA.

Introduction

Revision History

Chapter 2 *Bridge Operation Overview*

The EPC supports four modes of PCI operation:

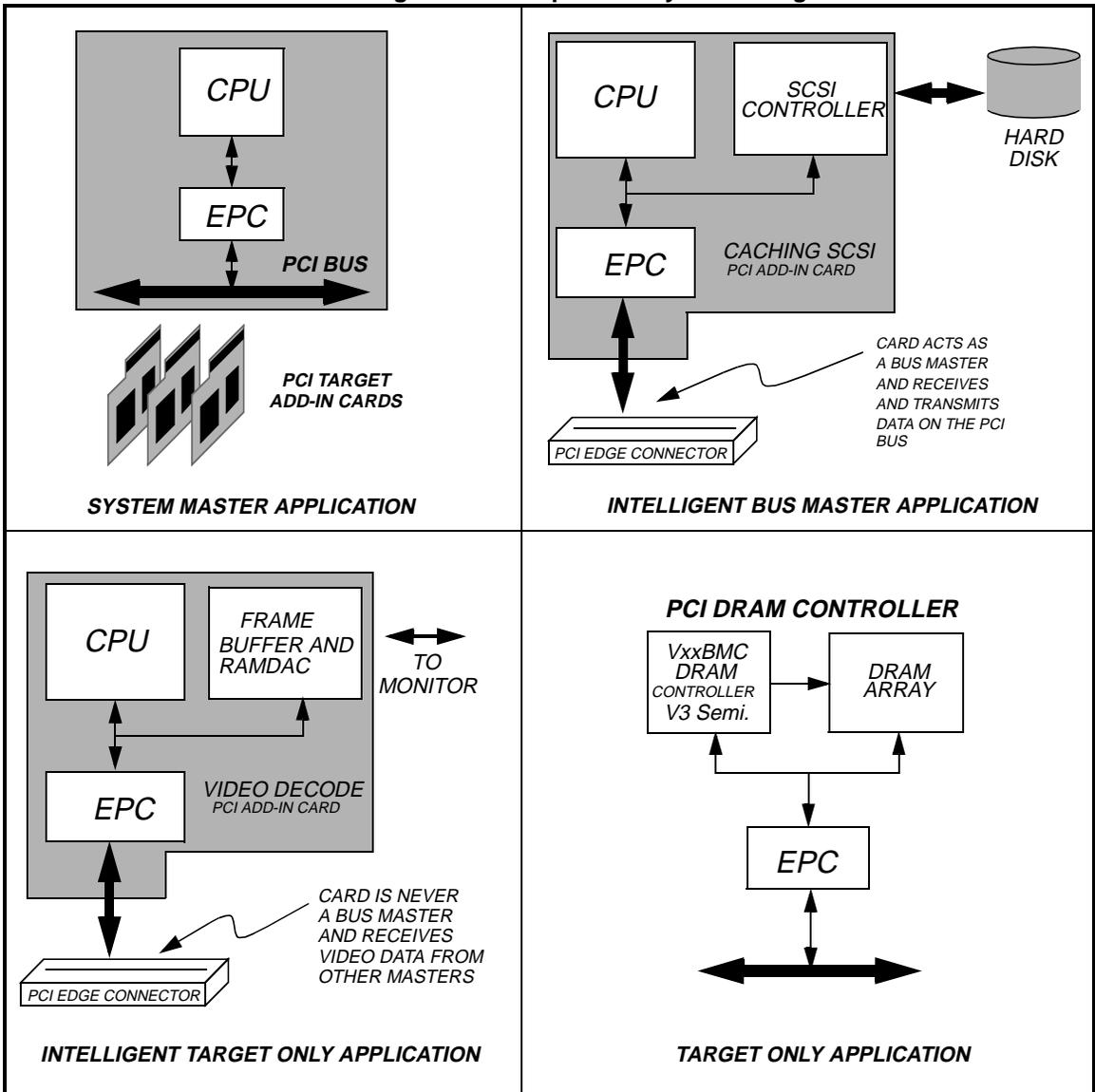
- **System Master** - The i960/Am29K processor is the PCI system master and PCI broadcasts configuration commands to attached PCI subsystems. This mode would be used, for example, in embedded systems using PCI as the system mezzanine bus.
- **Bus Master** - The i960/Am29K processor is part of a PCI subsystem that receives configuration commands from a primary system master, yet will act as a bus master during PCI transactions. This mode would be used, for example, for intelligent PCI add-in cards based on the i960/Am29K processor.
- **Target** - The i960/Am29K processor is part of a PCI subsystem that receives configuration commands from a host CPU and does not act as a PCI bus master.
- **Stand Alone Target** - The EPC can be used as a stand alone bridge component without an attached processor. For example, the EPC can be used with the VxBMC/CMC or V96SSC Burst DRAM Controller to build a two-chip PCI-to-DRAM interface.

Figure 2 shows example designs covering each of the four modes of operation.

Bridge Operation Overview

Operational Example

Figure 2: Example EPC System Designs



2.1 OPERATIONAL EXAMPLE

The easiest way to understand the operation of the EPC is to cover several simple operational examples. In this section we will briefly describe: PCI-to-Local transfers, Local-to-PCI transfers, DMA operation, and mailbox register usage. Initialization and configuration is deferred to later chapters.

After initialization the EPC monitors both the local and PCI buses simultaneously, waiting for access within pre-programmed regions of local memory or PCI space. Two independent programmable apertures are provided for local memory to PCI bus transfers; two more

apertures are provided for PCI to local memory transfers. The apertures can also perform address translation and byte-order conversion on accesses flowing across the bridge.

For our operational example, let's assume the following:

- The base address for PCI-to-Local aperture 0 is set at 1000.0000H with a size of 1 megabyte
- The base address for Local-to-PCI aperture 0 is set at E000.0000H with a size of 4 megabytes
- PCI-to-Local aperture 1 is disabled
- Local-to-PCI aperture 1 is disabled
- No errors occur during the transfers

2.1.1 Direct Local Bus Write to PCI Space

When the local bus master needs to perform a write to a device on the PCI bus, it simply writes data to a local memory location within a Local-to-PCI aperture. In this example, all local bus writes with an address within the range E000.0000H to E03F.FFFFH (base at E000.0000H with 4 megabyte window), will be "captured" and converted to PCI bus transfers. To the local bus master it appears that the write has completed. However, the address and data for the write have been placed into the Local-to-PCI bus write FIFO for completion when the PCI bus becomes available.

Both the address and data for the transfer can be changed as they flow across the bridge. If address translation is selected, the target address for the write can be changed from the local bus address to a different address in PCI space. For example, the bridge can be programmed to convert a write to location E000.0020H in local space, into a write to location A000.0020H in PCI space. In addition, the data for the write may optionally be converted from big endian to little endian byte order or vice-versa. Byte order conversion is useful in systems where the local processing uses one byte order and the main CPU uses another byte order. A good example is a networking PC add-in card that processes data in big-endian order, but has to share that data with an x86 processor host, which expects little endian byte order.

Once the "captured" write access reaches the PCI side of the FIFO, the EPC requests ownership of the PCI bus (how long the EPC waits before requesting the bus is programmable). When the central PCI arbiter grants the EPC the bus, the EPC then bursts the data queued in the write FIFO to the target PCI address. The EPC relinquishes ownership of the PCI bus once the write transfer is completed.

2.1.2 Direct Local Bus Read from PCI Space

A local bus read from PCI space progresses similarly to the write example above. When the local bus reads from a location in the Local-to-PCI aperture (in this case E000.0000H to E03F.FFFFH), these reads are converted into read transfers on the PCI bus. Unlike the write case, however, the local master cannot complete its read operation until the data is

Bridge Operation Overview

Operational Example

delivered from PCI space.

For example, let's look at the case where the local master wants to read from a register on an add-in card located at A000.0010H in PCI space. Let's assume the EPC is programmed to translate accesses to Local-to-PCI aperture 0 from E00n.nnnn in local memory to A00n.nnnnH in PCI space. To read from A000.0010H, the local master initiates a read to E000.0010H. The EPC sees this read, recognizes it as being within Local-to-PCI aperture 0 and "captures" it for bridging to the PCI bus. Since the data is not immediately available to return to the host, the bridge returns a NOT READY indication to the local processor. Simultaneously, the EPC requests the PCI bus. Once granted the bus, the EPC begins to read from the translated address (A000.0010H) and transfers that data back across the bridge to the local processor. As each datum becomes available on the local side, READY is returned to complete the local bus request. Just as in the write transfer case, the EPC can be programmed to perform byte order conversion as the data flows through the bridge. In order to improve system bandwidth, the EPC can be programmed to prefetch data from the PCI bus. The prefetched data will be 'cached' in the bridge until the next access.

2.1.3 PCI Write to Local Space

PCI bus masters gain access to the local memory space through the PCI-to-Local bus apertures. In our example, the PCI-to-Local aperture has been programmed to respond to PCI accesses within the 1000.0000H to 100F.FFFFH range (a 1 megabyte window). For a PCI master to write into the EPC's local memory, it simply writes to a PCI location falling within a PCI-to-Local aperture window.

For example, let's assume the PCI master wants to write to location 2000.0030H in the EPC's local memory space. Since the bridge is programmed to respond to accesses in the 1000.0000H to 100F.FFFFH range we will need to re-map the address so that the proper translation occurs. The PCI master now writes data to PCI location 1000.0030H, that access is captured by the bridge and buffered within the PCI-to-Local write FIFO. The PCI master completes the write and relinquishes the bus. Simultaneously, the address of the access is translated within the bridge, and the EPC requests access to the local bus (the protocol used to gain local bus mastership is processor version dependent). Once granted the local bus, the EPC will write the captured data in the new location in local memory space (2000.0030H). Data byte order translation is also available in the PCI-to-Local direction.

2.1.4 PCI Reads from Local Space

As you might expect, PCI reads from local space closely mirror local reads from PCI space. Let's assume the same conditions as the previous example, except in this case the PCI master wants to read from location 2000.0030H in local memory space. In this case, the PCI master reads from location 1000.0030H in PCI space (remember address translation will do this automatically once configured). The bridge immediately responds by deasserting "target ready" ($\overline{\text{TRDY}}$) to inform the PCI master that it will need to wait for the data to be fetched from the local side. Simultaneously, the EPC requests the local bus. Once granted the local bus, the EPC will read from location 2000.0030H and forward that data across the bridge to the PCI side. When the data is valid on the PCI side, $\overline{\text{TRDY}}$ will be asserted to signal completion of the read. As with the Local-to-PCI transfer, the EPC can prefetch data and cache it.

2.1.5 **DMA Transfers**

All four of the above examples assumed that the local processor, or a PCI master, was involved in the data transfer across the bridge. Often, higher overall system performance can be achieved by allowing simple transfers to be handled by a less-intelligent agent, such as a DMA controller. The EPC provides 2 DMA controllers, each capable of transferring data across the bridge without processor intervention.

Let's take the example of a caching disk controller PCI add-in card. The local processor is responsible for retrieving data from the hard drive and building buffers in local memory.

When the system host processor, a PowerPC processor for example, requests a specific buffer be moved into system memory, the local processor programs the bridge with the start address and transfer count for the data buffer, and the target address in PCI space for the data. The DMA controller then transfers this data autonomously, allowing both the local processor and the system processor to go about their business. The DMA controller can also be programmed to transfer a chain of buffers. This is useful when, in the above example, several 512 byte sectors must be assembled into a 16K byte logical block in the host's memory.

2.1.6 **Mailbox Registers and Doorbell Interrupts**

Often it is not practical for the EPC to request access to the local or PCI bus to transfer small amounts of information. Let's use the caching disk card as an example again. Perhaps the PowerPC system master wants to know if the disk access has been completed (i.e. the data has been moved from the local memory buffer into system memory). One way to do this would be to set up a semaphore in local memory that the local processor sets whenever a transfer was completed. To read the semaphore, the PowerPC system master would need to perform a cross-bridge read of local memory space, a potentially time consuming transfer to retrieve 1 bit of information.

A higher performance method to transfer small amounts of information is provided by the mailbox registers. These registers reside within the EPC and may be read or written from either side of the bridge. Each register is eight bits and the bit definition is application dependent. Using our caching disk controller example, one bit may be used to indicate "transfer complete". The local processor sets this bit by writing to a specific location within the EPC's configuration space (independent of the transfer apertures). The system master processor would then read this register directly from the EPC; no local bus access is required.

In addition to transferring data, the mailbox registers can be used to generate interrupts on either side of the bridge. For example, you may use one register to indicate transfer data *NOW*, in which case you would want to immediately interrupt the local processor.

Bridge Operation Overview

Operational Example

Chapter 3 *Internal Register Apertures*

The EPC does not use traditional chip-selects to access either its configuration registers or bridging functions. The chip-select function is replaced by address apertures, which compare addresses on both the PCI and Local bus. If an access is seen within one of these programmable apertures, then an internal chip-select is generated by the EPC.

There are a total of nine apertures implemented in the EPC:

- Four data transfer apertures that are used in the movement of data across the bridge. There are two apertures for data movement from PCI-to-Local and two for Local-to-PCI transfers. The data transfer apertures are described in detail in the next section.
- One PCI EPROM data transfer aperture is provided to allow location of a host system boot ROM on an adapter card. The boot ROM aperture can only transfer data from the local bus to PCI. Boot ROM support is discussed in the “PCI Configuration” chapter.
- One DOS Compatibility data transfer aperture is provided to allow real-mode DOS access to EPC local memory from the lower 1 megabyte of memory space and from DOS I/O holes. DOS memory and I/O support is discussed in the “PC Compatibility” chapter.
- One aperture is provided for access to the EPC’s internal registers from the Local bus.
- One aperture is provided for access to the EPC’s internal registers from the PCI bus (memory or IO cycles).
- One aperture is provided for access to the EPC’s internal registers from the PCI bus (PCI configuration cycles).

The function of the apertures is summarized in Figure 3.

Internal Register Apertures

Local Bus Access to Internal Registers

3.1 LOCAL BUS ACCESS TO INTERNAL REGISTERS

Local bus access to internal registers is controlled by the Local-to-Internal Register aperture. The base address for this register is set in the LB_IO_BASE register. The LB_IO_BASE register is initialized by special bus cycles on the local bus immediately following a reset (see "Initialization") or by serial EEPROM.

The LB_IO_BASE aperture has a fixed size of 64 kilobytes, although only a small part of this is actually used.

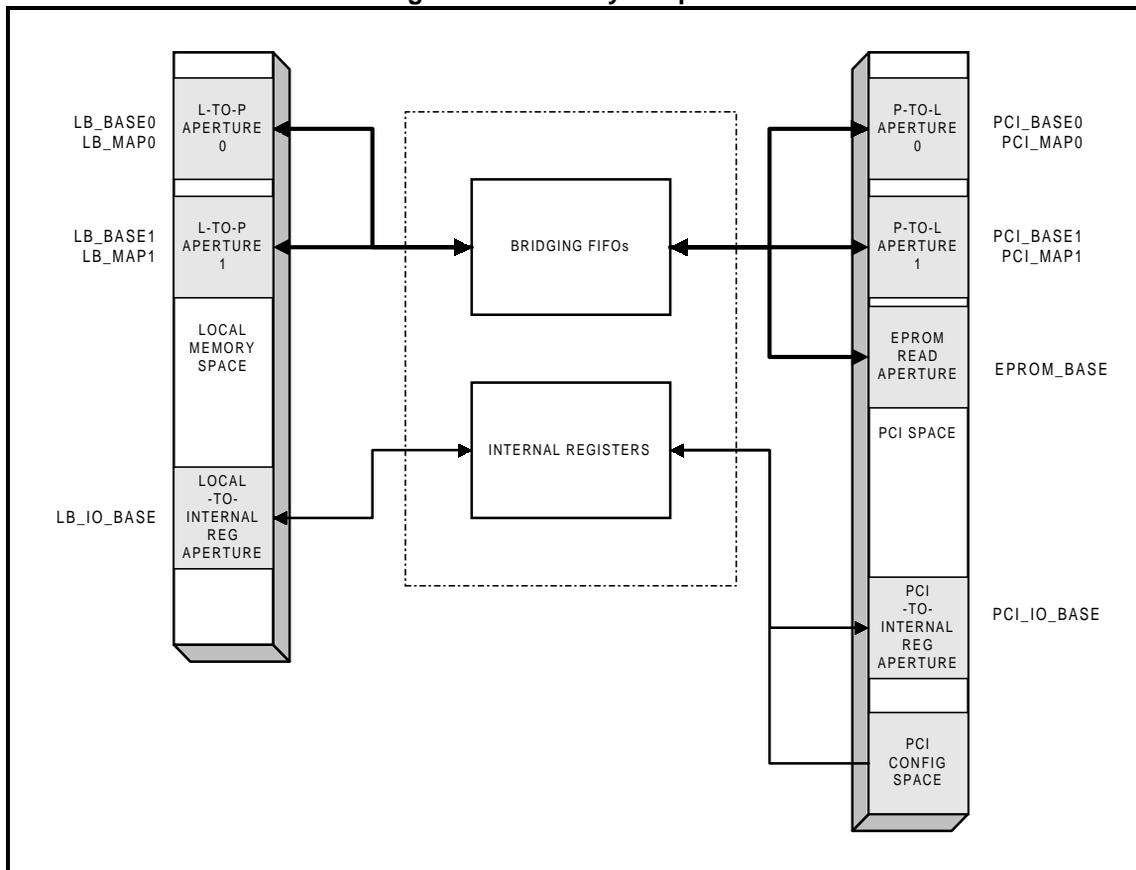
To write to a specific register through the LB_IO_BASE aperture, add the offset of the target register to the starting address of the Local-to-Internal Register aperture and use that address for the write. For example, if the LB_IO_BASE register is programmed to place the Local-to-Internal Register aperture at E123.0000H and you want to write to the PCI Command Register (offset 4H), then the target address would be E123.0004H.

3.2 PCI BUS ACCESS TO INTERNAL REGISTERS

PCI bus access to internal registers is controlled by the PCI-to-Internal Register aperture. The base address for this register is set in the PCI_IO_BASE register. The PCI_IO_BASE register is initialized either by the serial EEPROM, PCI configuration cycles, or local bus register accesses (see "Initialization"). The PCI_IO_BASE aperture has a fixed size of 256 bytes and may be located in either PCI memory or IO space.

The EPC's internal registers can also be accessed by standard PCI configuration cycles. Configuration cycles are initiated by a PCI system master by driving active the EPC's IDSEL pin, and then performing configuration reads and writes. For configuration reads/writes, only the low 8 bits of the address are used to index the internal registers. AD[31:8] are ignored during the address phase.

Figure 3: Summary of Aperture Function



Internal Register Apertures

PCI Bus Access to Internal Registers

Chapter 4

Data Transfer Apertures

There are four apertures provided for cross-bridge data transfers:

- PCI-to-Local apertures 0 and 1
- Local-to-PCI apertures 0 and 1

These apertures are tightly coupled to the read and write FIFOs, as well as to the address translation and data order conversion logic as shown in Figure 4.

Each aperture includes an address comparator that sets the base and size of the aperture. Accesses recognized by the address comparator are forwarded through the address remapper, byte order converter, and FIFO for that particular aperture.

Additional data transfer apertures are provided for PC compatibility. These apertures are discussed in the "PC Compatibility" chapter.

4.1 PCI-TO-LOCAL BUS APERTURES

The PCI-to-Local bus apertures control the following accesses: writes from PCI to Local memory and reads from local memory destined for PCI space. The PCI-to-Local bus apertures are implemented using the standard *Base Register* formats in the PCI configuration header. Unused base registers return all zeros when read or interrogated during configuration (see "PCI Configuration").

The programming of the PCI-to-Local apertures is controlled via the PCI_BASEx and PCI_MAPx registers. The following options are programmable for each aperture:

- Base address of aperture
- Aperture size
- Type of PCI accesses to respond to (IO or memory)
- Address translation
- Data byte ordering conversion
- Read prefetch enable/disable

Data Transfer Apertures

PCI-to-Local Bus Apertures

4.1.1 Setting the PCI-to-Local Aperture Base Address and Size

The base address for a PCI-to-Local memory aperture is set in the ADR_BASE field of the PCI_BASEx register (see "Register Descriptions" chapter for register layouts). In non-DOS mode, only AD[31:20] are significant yielding a minimum base address granularity of 1Mbyte.

The size of a PCI-to-Local aperture is set via the ADR_SIZE field in the PCI_MAPx register. Supported sizes for memory access are from 1 megabyte to 2 gigabytes increasing as powers-of-2 (1M, 2M, 4M, etc.)

When an aperture size greater than 1 megabyte is selected, the corresponding bits in the MAP_ADR field of the PCI_BASEx register are ignored. For example, if you choose an aperture size of 8 megabytes, then address bits A20, A21 and A22 are "masked off" by the PCI-side aperture address comparators.

In DOS Mode, the address comparators function differently allowing greater granularity for I/O and memory accesses (down to 8 bytes). Please see the chapter titled "DOS Compatibility" for more details.

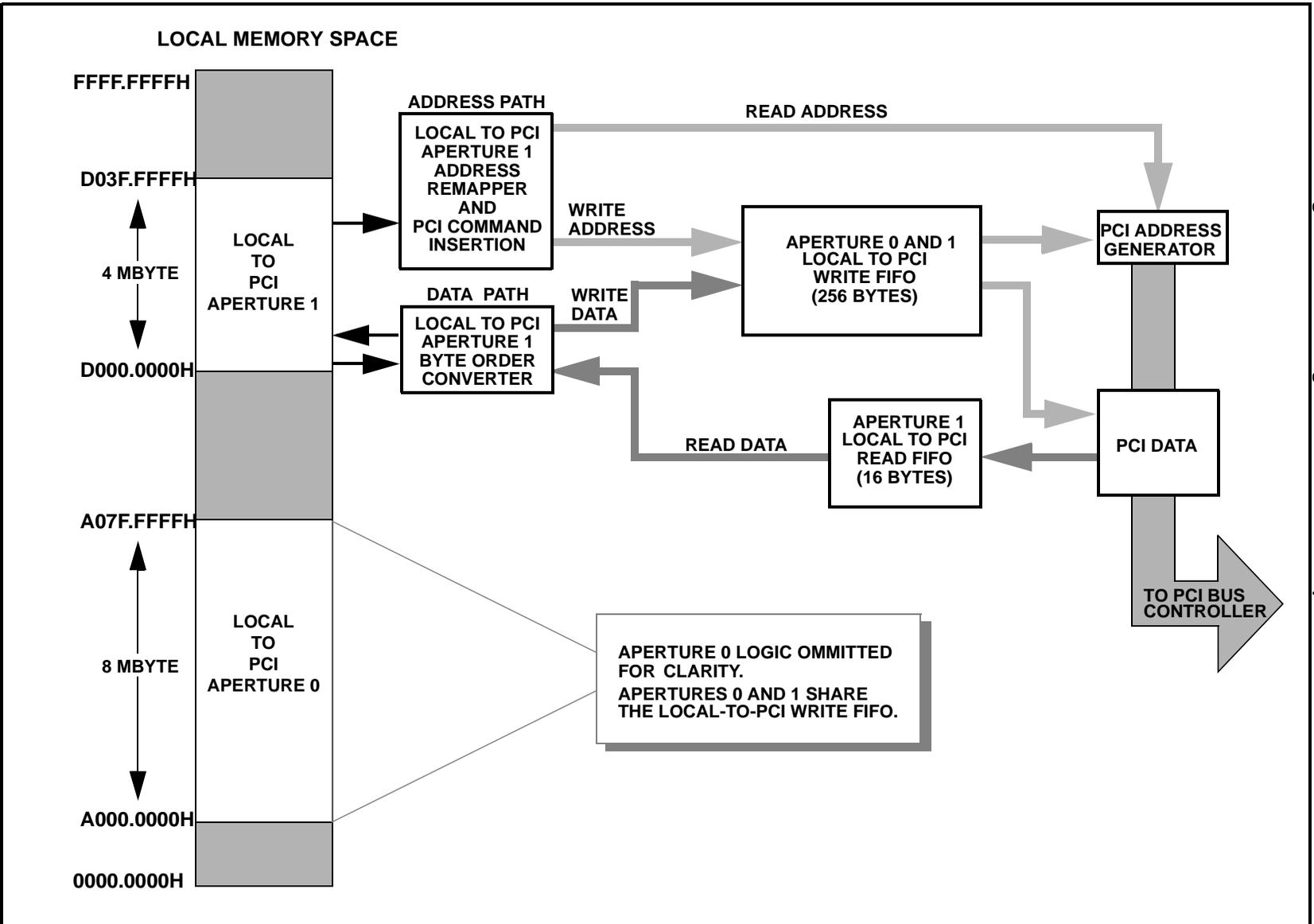
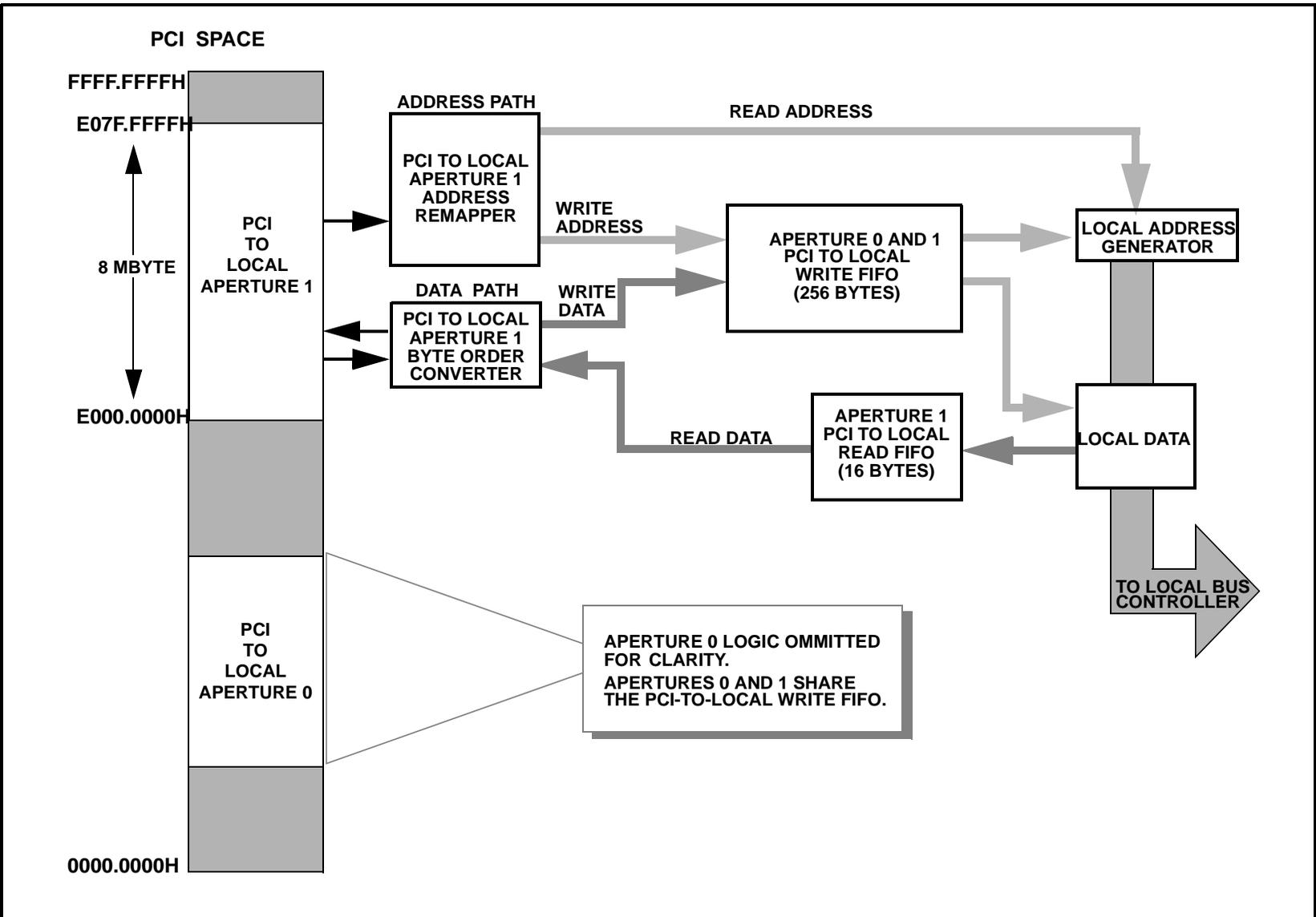


Figure 4: Block Diagram of Local-to-PCI Aperture/FIFO Connections

Data Transfer Apertures

PCI-to-Local Bus Apertures

Figure 5: Block Diagram of a PCI-to-Local Aperture/FIFO Connections



4.1.2 Selecting PCI Memory or I/O Space Mapping

The PCI-to-Local apertures may be mapped into PCI memory space or PCI I/O space. The IO bit in the PCI_BASEx registers controls this mapping. When IO=1, the aperture will only "capture" transfers on the PCI bus that are to I/O space and fall within the bounds set by the address base and size. Similarly, when IO=0, the corresponding PCI-to-Local aperture will only respond to memory space transfers on the PCI bus.

4.1.3 PCI-to-Local Address Translation

PCI-to-Local address translation is controlled via the MAP_ADDR field in the PCI_MAPx register. When an access is bridged from PCI-to-Local, the upper address bits of the PCI address are *always* replaced with the significant bits in the MAP_ADDR field. Which bits are considered "significant" is controlled by the size of the aperture. For example, with a 1 megabyte aperture size, the EPC will create the local bus address by replacing the A[31:20] of the PCI address with the entire MAP_ADR field. With larger apertures, less of upper address bits are replaced. For example with a 2 megabyte aperture, only PCI address bits A[31:21] will be replaced. Address remapping is "disabled" by simply setting the MAP_ADR and ADR_BASE fields for a particular aperture to the same value.

Table 2: PCI Address Remapping By Aperture Size^a

Aperture Size	PCI Address Bits Replaced by MAP_ADR Bits
1 meg	A[31:20]
2 meg	A[31:21]
4 meg	A[31:22]
8 meg	A[31:23]
16 meg	A[31:24]
32 meg	A[31:25]
64 meg	A[31:26]
128 meg	A[31:27]
256 meg	A[31:28]

a. Some combinations are used for special DOS Compatibility apertures.

4.1.4 Byte Order Conversion

The PCI-to-Local bus apertures also control the conversion of data byte ordering as data flows across the bridge. Writes from PCI space to local space are converted before entry into the write FIFO, reads from local space destined for PCI space are converted on their way from FIFO to the PCI bus (see Figure 5). Three modes of endian conversion are supported as is shown in Table 3. When byte order conversion is enabled, the byte enable signals are automatically corrected for the bridge transfer.

Data Transfer Apertures

PCI-to-Local Bus Apertures

Byte order conversion is controlled through the SWAP field in the PCI_MAPx registers.

Table 3: Byte Order Conversion Options

Swap Mode	Input Data Bytes				Output Data Bytes
	D[31:24]	D[23:16]	D[15:8]	D[7:0]	
32-bit (no swap)	D[31:24]	D[23:16]	D[15:8]	D[7:0]	
16-bit (half-word swap)	D[15:8]	D[7:0]	D[31:24]	D[23:16]	
8-bit (word swap)	D[7:0]	D[15:8]	D[23:16]	D[31:24]	

4.1.5 Enabling Read Prefetching

The EPC is capable of prefetching data for PCI-to-Local bus reads. Prefetching will often improve performance in applications that perform many sequential reads from the same aperture. Prefetching is discussed in more detail in “FIFO Architecture”.

Read prefetching is enabled for a PCI-to-Local aperture by setting the PREFETCH bit in the PCI_BASEx register.

4.1.6 Disabling PCI-to-Local Bus Apertures

The PCI specification does not provide a method to explicitly disable PCI apertures that are implemented as *base registers*. Disabling of the PCI-to-Local apertures is achieved by programming the ENABLE bit in the PCI_MAPx registers.

4.1.7 Overlapping Apertures

If data transfer apertures 0 and 1 overlap, the EPC will use aperture 0 (aperture 0 has priority). PCI_IO_BASE has the lowest priority if it overlaps either of the PCI_BASEx regions.

4.1.8 Special Function Modes for PCI-to-Local Bus Apertures

PCI-to-Local bus aperture 0 shares some functionality with the expansion ROM base aperture (see “PC Compatibility”). The address decoder for PCI-to-Local aperture 0 is shared with the expansion ROM base register. When the expansion ROM base is enabled, the decoder will only bridge accesses within the ROM window. When the ROM is disabled, PCI-to-Local bus aperture 0 will function as described above. Typically, the expansion ROM is used only during BIOS boot, if at all. The expansion ROM base register can be completely disabled via software.

PCI-to-Local bus aperture 1 includes special logic for compatibility with legacy DOS

systems. These functions are described in the "PC Compatibility" chapter.

4.2 LOCAL-TO-PCI BUS APERTURES

The Local-to-PCI bus apertures control the following accesses: writes from local memory to PCI space and reads from PCI space destined for the local processor/memory space.

The programming of the Local-to-PCI apertures is controlled via the LB_BASEx and LB_MAPx registers. Many of the features for the Local-to-PCI apertures are identical to those for the PCI-to-Local apertures. The following options are programmable for each Local-to-PCI aperture:

- Base address of aperture
- Aperture size
- Type of PCI command generated (memory, I/O, configuration, etc.)
- Address translation
- Endian conversion
- Read prefetch enable/disable
- Local-to-PCI aperture enable/disable

4.2.1 Setting the PCI Command Type

Each PCI address cycle includes information about the type of access being attempted (i.e. memory, I/O, configuration). This "type of access" information is also called a PCI command, and is encoded on the C/BE[3:0] lines of the PCI bus.

On the local side of the bridge, the EPC sees only memory reads and writes. The PCI bus, however, supports many types of accesses. When an access is "captured" on the local bus it must be converted into a specific type of PCI access. This conversion is controlled by the TYPE bits in the LB_MAPx register. During the address phase of a PCI access, the TYPE bits are copied directly to the C/BE[3:1] pins, while C/BE0 is set according to the direction of the access (read or write). Table 4 shows the command encodings per the revision 2.1 PCI specification and the appropriate TYPE field entries to generate these encodings.

The EPC will generate commands that are "reserved" in the specification, and performs no error checking on the validity of the encodings. The EPC will also generate a "Dual Address Cycle (DAC)" command if programmed to do so, however, Dual Addressing is not supported by the bridge and thus will fail during the second phase of a dual address cycle.¹

Data Transfer Apertures

Local-to-PCI Bus Apertures

Table 4: PCI Command Encodings and Corresponding TYPE Field Values

C/BE[3:0]	Command	TYPE Field Value
0000	Interrupt Acknowledge	000
0001	Special Cycle	000
0010	I/O Read	001
0011	I/O Write	001
0100	reserved	010
0101	reserved	010
0110	Memory Read	011
0111	Memory Write	011
1000	reserved	100
1001	reserved	100
1010	Configuration Read	101
1011	Configuration Write	101
1100	Memory Read Multiple	110
1101	Dual-Address Cycle (DO NOT USE)	110
1110	Memory Read Line	111
1111	Memory Write and Invalidate	111

4.2.2 **Setting the Local-to-PCI Aperture Base Address and Size**

The base address for a Local-to-PCI memory aperture is set in the ADR_BASE field of the LB_BASEx register. Only AD[31:20] are significant yielding a minimum base address granularity of 1Mbyte.

The size of a Local-to-PCI aperture is set via the ADR_SIZE field in the LB_BASEx register. Supported sizes are from 1 to 2¹ gigabytes increasing as powers-of-2 (1M, 2M, 4M, etc.).

1. DAC is used to support 64-bit targets on a 32-bit PCI bus. It requires that the 64-bit address be given in two back-to-back address phases followed by the burst 32-bit data phases. See the PCI specification for more details. THE EPC DOES NOT SUPPORT DAC.

1.Version A0 of the EPC or newer will allow the Local-to-PCI apertures size up to 2Gb (on 512Mb boundary), and the PCI-to-Local aperture size up to 1Gb (on 1Gb boundary). However, the older PBC devices only support up to 256 Mb apertures.

4.2.3 Local-to-PCI Address Translation

Local-to-PCI address translation is controlled via the MAP_ADDR field in the LB_BASEx register. Address remapping from local-to-PCI is implemented identically to remapping in the PCI-to-Local direction. Please see "PCI-to-Local Address Translation" on page 21 for details.

4.2.4 Byte Order Conversion

Byte order conversion for local-to-PCI transfers is controlled through the SWAP field in the LB_BASEx registers. For a description of byte order conversion, please see the PCI-to-Local description above.

4.2.5 Enabling Read Prefetching

The EPC is capable of prefetching data for Local-to-PCI bus reads. Prefetching will often improve performance in applications that perform many sequential reads from the same aperture. Prefetching is discussed in more detail in the "FIFO Operation" section of this chapter.

Read prefetching is enabled for a Local-to-PCI aperture by setting the PREFETCH bit in the LB_BASEx register.

4.2.6 Enabling Local-to-PCI Bus Apertures

Unlike the PCI-to-Local registers, it is possible to explicitly enable and disable Local-to-PCI apertures. The Local-to-PCI apertures are **disabled following a reset** and must be enabled before the EPC will recognize Local-to-PCI transfers. To enable an Local-to-PCI aperture, set the ENABLE bit in the appropriate LB_BASEx register.

Data Transfer Apertures

Local-to-PCI Bus Apertures



Chapter 5 FIFO Architecture and Operation

The Local bus and PCI bus are decoupled from each other through the use of FIFOs. The FIFOs provide “elastic” storage for transfers passing across the bridge. The FIFOs also provide the synchronization necessary when running the PCI bus and the Local bus at different frequencies.

FIFOs are necessary to prevent performance bottlenecks that would arise if the Local and PCI buses were connected directly. As an example, imagine a bridge with no FIFO storage. When a local master wanted to write to the PCI bus, it would have to wait (i.e. be held NOT READY) until the PCI bus was available before continuing operation. With FIFOs however, the local master simply writes data into the FIFO and expects the bridge to complete the transfer at a later time (*write posting*).

The size of the FIFO storage in a PCI bridge is very important, especially in high-bandwidth applications. It is very possible that a small FIFO could be filled by a single data transfer, leaving one of the buses "hanging" and seriously degrading system performance. The EPC bridge includes a large amount of FIFO storage to prevent such situations from occurring.

Another important capability provided by the EPC FIFOs is called *Dynamic Bandwidth Allocation*. This feature allows the programmer to control precisely how empty, or full the FIFOs get before initiating a data transfer as well as setting the priority between reads and writes in each direction. Dynamic Bandwidth Allocation is critical for applications using high-bandwidth peripherals such as advanced networking, communications, and graphics devices. The EPC also includes FIFO performance monitoring logic to allow the tuning of system code to maximize performance.

5.1 DYNAMIC BANDWIDTH ALLOCATION FIFO ARCHITECTURE

The FIFO architecture is logically divided in two blocks: one for PCI-to-Local transfers, another for Local-to-PCI transfers. The FIFO architecture is symmetrical: the same amount of buffering and programmability are provided in each direction. The FIFO organization is shown in the block diagram in Figure 6.

There are three FIFOs within the Local-to-PCI FIFO block:

- **Local-to-PCI Write FIFO (256 bytes)**. Buffers local bus writes to local bus data transfer apertures 0 and 1. Also buffers DMA writes to the PCI bus. This FIFO contains the initial address and data for a transaction.

FIFO Architecture and Operation

Write FIFO Operation and Programming

- **PCI Read Aperture 0 FIFO (32 bytes).** Buffers PCI bus reads from local memory issued through PCI aperture 0. This FIFO also buffers prefetch reads from local memory when this feature is enabled.
- **PCI Read Aperture 1 FIFO (32 bytes).** Buffers PCI bus reads from local memory issued through PCI aperture 1. This FIFO also buffers fetch-ahead reads from local memory when this feature is enabled.

In addition, there are three FIFOs within the PCI-to-Local FIFO block:

- **PCI-to-Local Write FIFO (256 bytes).** Buffers PCI bus writes to PCI bus apertures 0 and 1. Also buffers DMA writes to the local bus. This FIFO contains both the address and data for a transaction.
- **Local Read Aperture 0 FIFO (32 bytes).** Buffers local bus reads from PCI space issued through local bus aperture 0. This FIFO also buffers fetch-ahead reads from PCI space when this feature is enabled.
- **Local Read Aperture 1 FIFO (32 bytes).** Buffers local bus reads from PCI space issued through PCI aperture 1. This FIFO also buffers fetch-ahead reads from PCI space when this feature is enabled.

The “Dynamic Bandwidth Allocation” feature is the ability to use the large 256-byte FIFO in each direction for multiple transfers; both DMA and Aperture. At any one instant in time, a FIFO can contain 1 or more of the following items:

- Aperture 0 posted writes
- Aperture 1 posted writes
- DMA 0 data
- DMA 1 data

5.2 WRITE FIFO OPERATION AND PROGRAMMING

Each write FIFO stores address, byte enable, and data information for write transfers bridged across the EPC. The operation and programming of the PCI-to-Local and Local-to-PCI write FIFOs are nearly identical. For brevity, this section will refer to a generic “write FIFO operation”.

The write FIFOs are used for the following transactions:

- Writes to either data transfer aperture 0 or aperture 1
- DMA transfers destined for the PCI (local) bus

When an access is bridged from one bus to the other, the starting address of the access is captured, translated, and then stored in the write FIFO. Subsequent data cycles are also captured, byte-order converted, and then stored in the write FIFO as well. The state of the byte enable lines is also captured, byte-order corrected and then stored with the data.¹ This process is shown in Figure 7 for a four word burst write from Local-to-PCI with a starting address of 1000.4020H.

For writes from Local-to-PCI the command type to be generated for the PCI write is also generated and stored in the Local-to-PCI write FIFO. For example, if you have programmed Local-to-PCI aperture 1 to generate “Configuration” commands, then as the address for Local-to-PCI Aperture 1 writes passes through the Local-to-PCI Aperture 1 address remapper it will be “tagged” with the type of PCI command to be generated. This information is then stored in the Local-to-PCI FIFO.

5.2.1 Write FIFO Draining Strategies

The target bus for a write transaction must be requested by the EPC in order for the write to complete. The $\overline{\text{REQ}}/\overline{\text{GNT}}$ protocol is used on the PCI bus for this purpose. The local bus mastership protocol is dependent on the processor selected and is described in greater detail in the “Local Bus Interface” chapter. How rapidly the target bus is requested is programmable through the selection of a write FIFO *draining strategy*. The options for draining strategies are shown in Table 5.

The choice of strategy is highly application dependent. Draining strategy “00” will result in the fastest transfer of data from one bus to the other; however, it will also result in the most bus traffic. Strategy “10” has the advantage of preserving bus bandwidth, but will allow small transfers (< 3 words) to sit in the write FIFO indefinitely. Strategy “11” will preserve bus bandwidth, but will complete the write by requesting the bus as soon as the data write transaction filling the FIFO is complete (i.e. data will not sit in the FIFO indefinitely). The write FIFO draining strategies are controlled via the FIFO_CFG register.

Table 5: Summary of Write FIFO Draining Strategies

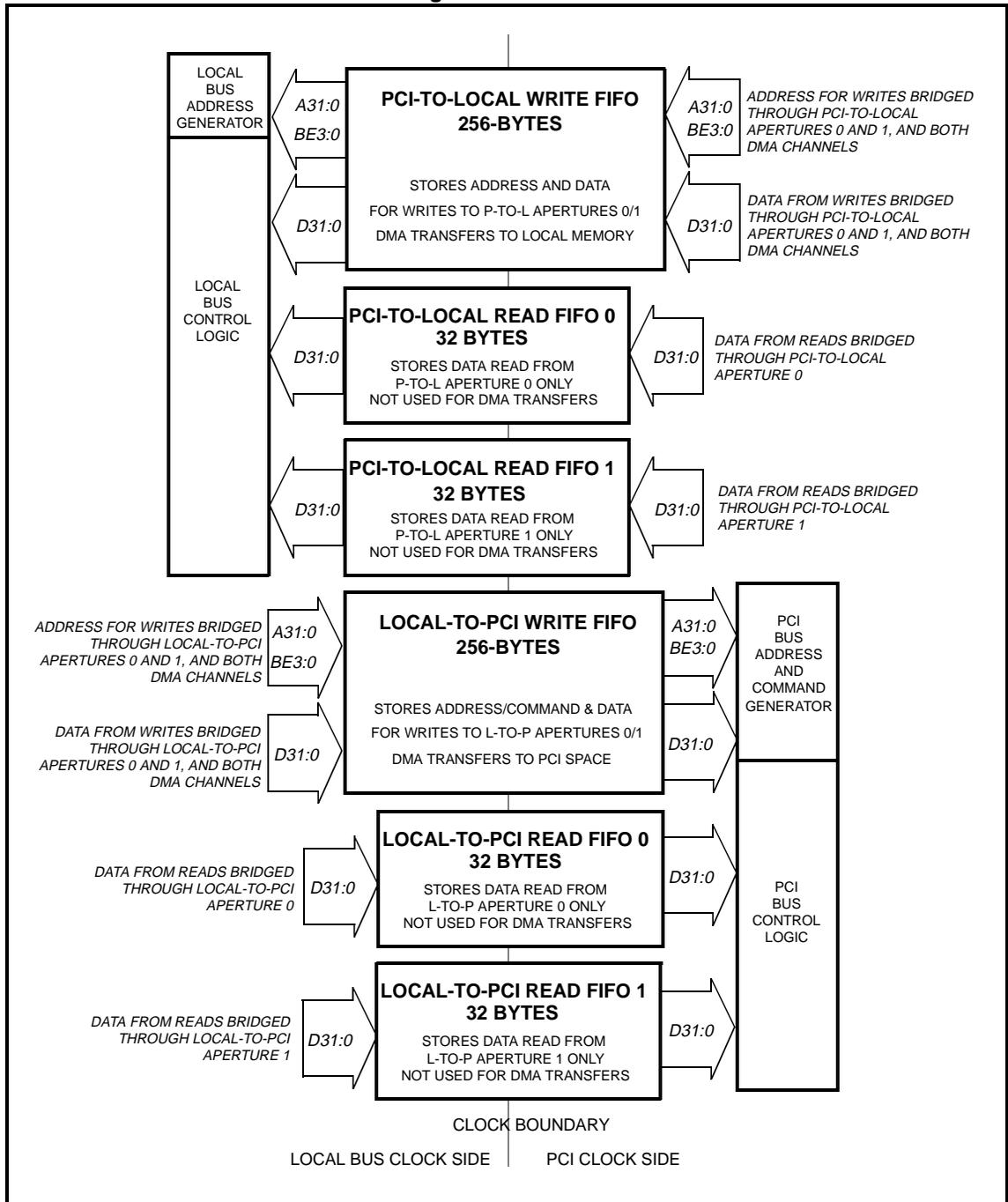
Strategy	Description
00	Request target bus whenever corresponding write FIFO is not empty
01	Reserved: do not use
10	Request target bus when 3 or more words of data have been posted in the corresponding write FIFO
11	Request target bus when 3 or more words of data have been posted in the write FIFO <i>or</i> a burst write has been completed at the “filling end” of the FIFO

1. The storage space for the byte enable state is in addition to the 256-byte FIFO size.

FIFO Architecture and Operation

Write FIFO Operation and Programming

Figure 6: FIFO Architecture



5.3 READ FIFO OPERATION AND PROGRAMMING

Two 16-byte read FIFOs are provided for transfers in each direction. Each transfer aperture has a separate read FIFO as shown in Figure 6. Like the write FIFOs, the operation of each read FIFO is identical.

The read FIFOs store data for the following transactions:

- PCI reads from local space through PCI-to-Local apertures 0 and 1
- Local bus reads from PCI space through Local-to-PCI apertures 0 and 1

The read FIFOs serve two purposes: to allow synchronization between the PCI and Local buses and to provide storage for read prefetching. The read FIFOs only store the data for read transfers, the address is not stored since reads request the target bus immediately (i.e. reads are not “posted”).

5.3.1 Prefetching and Read FIFO Filling Strategies

Performance for reads from sequential locations can be improved by enabling read prefetching. With prefetching enabled, a read FIFO will “guess” that additional read transfers will occur and will perform burst reads to fill the FIFO with data from subsequent locations. The aggressiveness of the prefetch is controlled via a programmable “filling strategy” for each read FIFO. The filling strategies are controlled via the FIFO_CFG register.

As an example, let’s assume prefetching is enabled for Local-to-PCI aperture 0 (active from 1000.0000H to 107F.FFFFH, no address remapping). When the local bus master performs a read from location 1000.0020H, the Local-to-PCI FIFO 0 will initiate a PCI burst read from 1000.0020H to 1000.002FH (4 words). The read FIFO will continue to fill itself as long as its programmable “filling strategy” dictates.

The read FIFOs contain an address comparator that is used to determine whether or not a new read request can be serviced from data already fetched. For example, if the Local CPU performs a two word burst access from 100H and 104H, the read FIFO will continue to fetch ahead from locations 108H 10CH, etc. If the Local CPU then performs a burst read from 108H and 10CH, the EPC will supply the data directly from the read FIFO. If the next access to a read FIFO is not the next highest word address, then all entries in the read FIFO are invalidated and the EPC fetches new data from the PCI (or Local) bus.

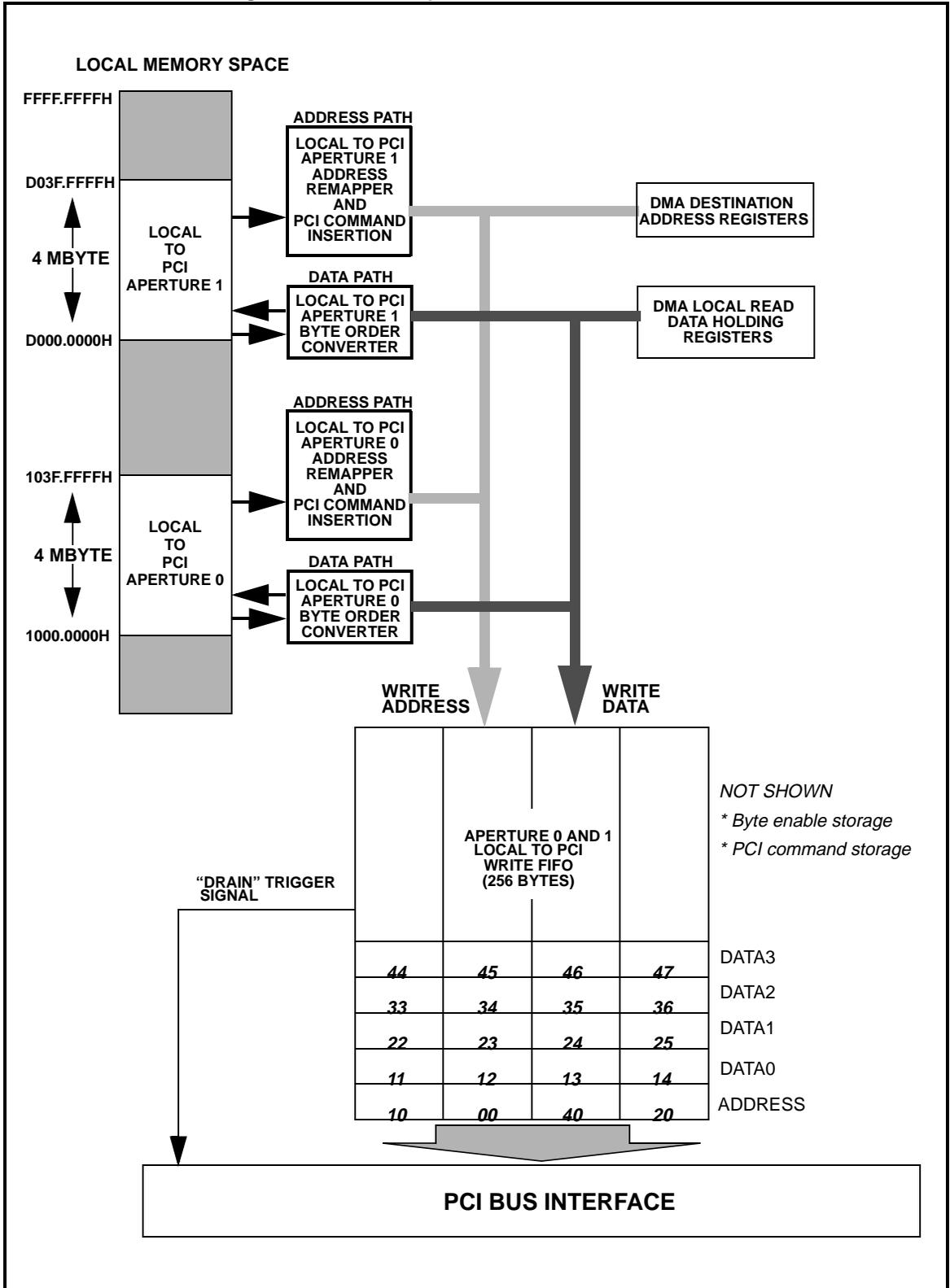
Table 6: Summary of Read FIFO Filling Strategies

Strategy	Description
00	Request the PCI (or Local) bus whenever there is room for at least 1 word in the read FIFO
01	Request the PCI (or Local) bus whenever there is room for at least 2 words in the read FIFO
10	Request the PCI (or Local) bus whenever the read FIFO is empty
11	Reserved: do not use

FIFO Architecture and Operation

Read FIFO Operation and Programming

Figure 7: Detailed Operation of the Local-to-PCI Write FIFOs



5.4 FIFO PRIORITIZATION OPTIONS

The EPC allows reads and writes bound for completion on the same bus to be prioritized with respect to each other. This capability is critical in systems with many PCI or local bus masters.

For example, let's assume that a number of writes have been posted in the Local-to-PCI FIFO while waiting for the EPC to be granted the PCI bus. Subsequently, the local bus master requests a read from the PCI bus. When the EPC is finally granted the PCI bus, which transfer proceeds? If the EPC were designed like many bridges, the answer would be "first in first out"; in other words the local processor would be forced to wait while all of the writes in the Local-to-PCI write FIFO had completed. Luckily, the EPC includes prioritization options that allow the read to proceed first, "unlocking" the local bus and allowing the local CPU to get back to work.

FIFO prioritization options are described in Table 7, below.

Table 7: Read/Write Prioritization Options

Prioritization Option	Result
Local bus reads ahead of writes	Pending local bus reads will complete before writes are allowed to complete. Prevents PCI bus from experiencing multiple disconnect/retries while attempting a read of local memory. May result in data coherency hazards (see below).
Local bus writes ahead of reads	Pending local bus writes will complete ahead of reads. Prevents data coherency hazards, but may cause PCI bandwidth degradations.
PCI bus reads ahead of writes	Pending PCI bus reads will complete before writes are allowed to complete. Prevents local bus from experiencing extended lockup while attempting a read of PCI space. May result in data coherency hazards (see below).
PCI bus writes ahead of reads	Pending PCI bus writes will complete ahead of reads. Prevents data coherency hazards, but may cause local bus bandwidth and CPU performance degradation.

The FIFO priority is only considered by the EPC on arbitration boundary. For example, when reads and writes are pending, the priority decision will be made after EPC's HOLDA is deasserted.

5.5 FIFO DATA COHERENCY OPTIONS

Some applications will require strict data coherency. With programmable FIFOs priority, it is possible that a write to location in memory may be prioritized behind a read to the same location that occurred later in time. Allowing such a read to proceed would result in "stale" data being returned to the initiator of the read. The EPC provides a programmable data coherency mechanism to prevent this situation from occurring. Table 8 shows the coherency mechanisms available. Coherency options are set through the FIFO_PRIORITY register.

FIFO Architecture and Operation

FIFO Data Coherency Options

In addition to the options shown in Table 8, all FIFOs can be flushed immediately under program control. Direct FIFO data flushing is performed by writes to appropriate “flush” bits in the SYSTEM register. A FIFO flush results in all data within the FIFO being invalidated. This is true of the write FIFOs as well: flushing a write FIFO *does not* result in the data in the FIFO being written to the target bus (the data in the FIFO is simply “thrown away”). Using this method to flush does NOT alter the state machines that fill or drain the FIFO. Consequently, this method of flushing should not be used under normal operations. This flushing mechanism is intended to be used only for test or fault recovery situations.

When prefetching is disabled for a particular aperture, then data flushing for that aperture should be disabled. This does **not** create a coherency problem because when prefetching disabled stale data can never remain sitting in the FIFO (only the exact amount of data asked for will be fetched). Enabling flush for a non-prefetch aperture will result in unpredictable read behaviour. .

Table 8: FIFO Data Coherency Options

Coherency Strategy	Programmable Options for Flushing the READ FIFO
Local Bus Aperture 1 Read-Ahead FIFO Flush Strategy	<ul style="list-style-type: none">· Local bus to PCI writes never cause a flush· Local bus to PCI writes to aperture 1 <i>only</i> cause a flush· Local bus to PCI writes to either aperture cause a flush
Local Bus Aperture 0 Read-Ahead FIFO Flush Strategy	<ul style="list-style-type: none">· Local bus to PCI writes never cause a flush· Local bus to PCI writes to aperture 0 <i>only</i> cause a flush· Local bus to PCI writes to either aperture cause a flush
PCI Bus Aperture 1 Read-Ahead FIFO Flush Strategy	<ul style="list-style-type: none">· PCI to local bus writes never cause a flush· PCI to local bus writes to aperture 1 <i>only</i> cause a flush· PCI to local bus writes to either aperture cause a flush
PCI Bus Aperture 0 Read-Ahead FIFO Flush Strategy	<ul style="list-style-type: none">· PCI to local bus writes never cause a flush· PCI to local bus writes to aperture 0 <i>only</i> cause a flush· PCI to local bus writes to either aperture cause a flush

5.5.1 Ensuring Strict Data Coherency

In systems where data coherency must be strictly maintained, the following options should be selected:

- If prefetch is enabled, cause a flush of the read prefetch FIFO(s) whenever a write occurs by programming FIFO_PRIORITY for LB_RD0,1/PCI_RD0,1 = “11”
- Program FIFO_PRIORITY so that writes have priority over reads
- Program FIFO_CFG so that the write FIFO drain strategy is “00” or “11” (this will prevent write data from sitting in the FIFO)

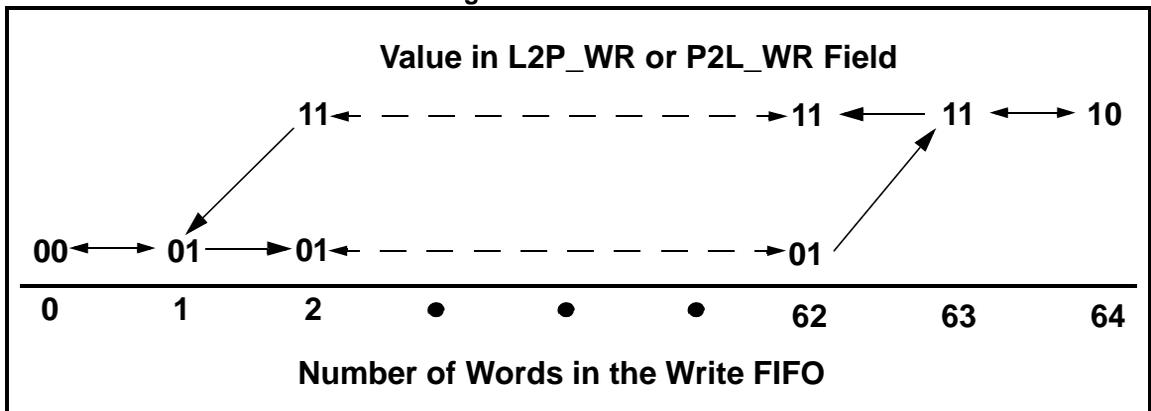
5.5.2 Monitoring the Status of Read and Write FIFOs

The two write FIFOs and four read FIFOs each provide an indication of their relative “fullness” through bits in the FIFO_STATUS register. This information can be used to help tune “filling” and “draining” strategies, as well as to determine when there is room available in the FIFOs for additional transactions.

The status bits for the write FIFOs indicate both the fullness of the FIFOs as well as whether the FIFOs are in the process of filling or draining. Two bits are used in the P2L_WR and L2P_WR fields in the FIFO_STATUS register. A description of these bits is shown in Figure 8.

Each write FIFO has a two bit flag that indicates whether there is room for additional data. The read FIFO status bit encoding is shown in the *Register Descriptions* chapter.

Figure 8: Write FIFO Status Bits



5.5.3 Ensuring the Completion of a Posted Write

Occasionally, the system software will need to ensure that a posted write has completed on the PCI bus. There are two methods available:

- **Software Polling.** Before issuing the write transfer, wait for the corresponding write FIFO to become empty (by monitoring the FIFO status bits). Then post the write and begin monitoring the status bits again. It is a good idea to check the PCI_STATUS register to see if any errors occurred (e.g. a Master or Target Abort).
- **Hardware Stall.** Program the FIFO priority for writes-ahead-of-reads. First post the write in the write FIFO, then attempt a dummy read from the same aperture. The dummy read will lock the local bus until the write completes. (Note: systems using V3 Semiconductor memory controllers must take into account the effect of the bus watch timers if those features are enabled!).

5.6 FIFO LATENCY

FIFO latency is defined as the amount of time necessary for a word of data to flow through a given FIFO from source to destination. Since the EPC supports fully asynchronous operation of the PCI and local interfaces, exact latency is impossible to specify and is not deterministic.

The only way to specify best case latencies is to assume both interfaces run at the same frequency with little or no skew between clocks. In such a situation the latency for data through the bridge will range from 2-4 clock cycles. An exact time cannot be given since there is no internal synchronization between the PCI and local bus clocks within the EPC.

Chapter 6

DMA Controller

The EPC's DMA Controller includes two channels, each capable of transferring data from Local memory to PCI, or from PCI to Local memory. The DMA Controller supports the following features:

- Block transfers up to 4 megabytes in size for each link
- On-the-fly byte order conversion
- PCI and local transfer ranges independent of data transfer apertures
- Block chaining with no limit on the number of chained blocks

The DMA controller is useful in applications that transfer large amounts of sequential data across the bridge. For example, an intelligent disk controller may need to transfer a data buffer from the local memory space to the host processors' memory space located on the PCI bus. Using DMA is ideal in this case, since the programmer simply sets starting addresses and the transfer count, then lets the EPC transfer the data without local or host processor intervention.

6.1 DMA TRANSFERS

A DMA transfer consists of the movement of data by the EPC from local memory to the PCI bus, or from PCI memory to the local bus.

6.1.1 Local Bus to PCI Bus DMA Transfers

Prior to starting local-to-PCI DMA transfers, the programmer must set the local and PCI starting address, as well as the byte order conversion, direction, and priority options. DMA transfers are initiated by setting the IPR bit within the DMA_CSRx register.

A Local-to-PCI DMA data transfer begins with the EPC requesting the local bus. Once the local bus is granted, the EPC performs a local bus read to fetch the contents of the memory location pointed to by the local DMA address register (DMA_LOCAL_ADDRx). The local bus read performed by the DMA controller does not use the read prefetch FIFOs. The data read from the local bus is then placed in the Local-to-PCI write FIFO, the address for the write phase is stored in the DMA address generator. The PCI command that is used for the write

DMA Controller

DMA Transfers

cycle can be programmed via the DMA_WTYPE bits (DMA Write to PCI Bus Command Type) in the PCI_CFG register. It defaults to a value of 3h which produces a memory write command of "0111".

After each local bus read transfer has completed, the local address is incremented and the transfer count register is decremented. The DMA controller will continue to repeat the above process until the transfer count reaches zero. Since transfer count is tracked on the source *read* cycle (as opposed to the destination *write* cycle), polling the transfer count register for a zero value is NOT a reliable way to ensure that the transfer is complete since the data may not yet be written to the destination and may be sitting in the FIFO. To determine that a DMA transfer is finished and it is safe to reprogram it, the DMA_IPR bit in the DMA_CSRx register should be polled for '0'. If DMA_IPR is not clear, the contents of the destination address should not be changed.

When the transfer count has reached zero, and the destination data has been written completely, the DMA controller will either generate a "process complete" interrupt, or fetch the next block transfer descriptor (if programmed for chaining). Block chaining is described below.

6.1.2 PCI Bus to Local Bus DMA Transfers

PCI to Local bus DMA transfers operate nearly identically to the Local to PCI DMA transfers described above. In this case, the data is read from the PCI side of the bridge (using a PCI "Read Memory" command) and then posted in the PCI-to-Local write FIFO.

A flowchart of basic DMA operation is shown in Figure 10.

6.1.2.1 A Special Note on Byte Enables (Using DMA in FIFO Applications)

When using the DMA to transfer from PCI to a FIFO on the local bus there is a special consideration that must be accounted for. If a PCI burst read of source data is disconnected without data¹ by the slave device then a "dummy" data will be written to the local bus. This "dummy" data is identified by the byte enables being all de-asserted. This does not cause a problem for memory systems that implement a write-per-byte (as most do). However, an application involving a FIFO must cause the cycle to be ignored (i.e. don't push) when $BE\#[3:0] = "1111"$.

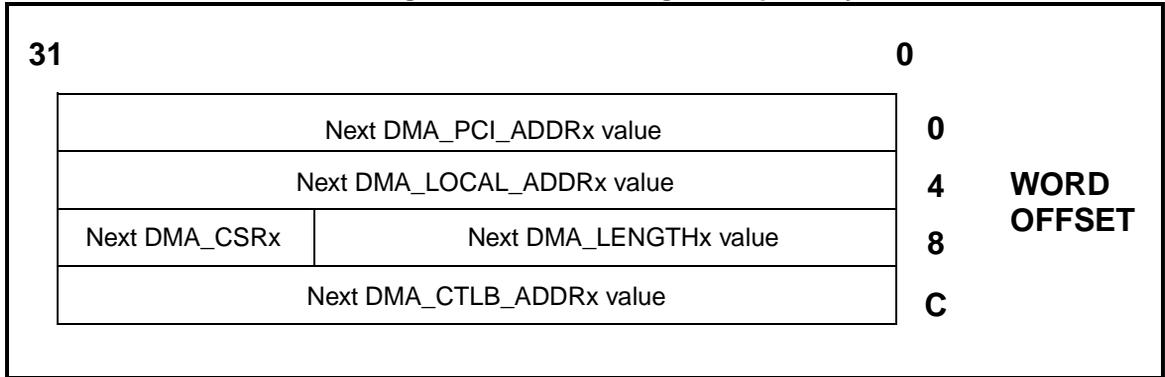
6.1.3 DMA Block Chaining

Upon completion of a single block transfer, the DMA Controller can be programmed to automatically fetch a new control block descriptor with the parameters for a subsequent block. The descriptor includes the values for the PCI start address, Local start address, transfer length, next control block descriptor address and control information. The initial

1. Disconnecting without data is inefficient and should be avoided by slave devices. For this reason the EPC (as a slave device) does a disconnect WITH data when necessary.

control block address is stored in the DMA_CTLB_ADRx register for each channel. DMA control blocks may be located in local memory only. Figure 9 shows the layout of the DMA chaining descriptor. Please note that the format for the chaining descriptor is for little-endian memory, as this is the same format used within the EPC's control registers.

Figure 9: DMA Chaining Descriptor Layout.



DMA chaining is controlled by the CHAIN bit in the DMA_CSRx register. Block chaining transfers begin the same as normal DMA transfers: the first block's start addresses, transfer count, and control parameters are set up by the programmer in the EPC's DMA registers. When the first transfer completes, the DMA controller checks the CHAIN bit, and if this bit is set, it will fetch the descriptor pointed to by the DMA_CTLB_ADRx register. Chaining terminates when the DMA controller completes a block for which the CHAIN bit is not set. There is no limit on how many blocks may be chained together.

6.1.4 Multi-processor DMA Chaining

A special bit in the DMA_CSR registers (called CLR_LEN) can be used to cause the length value of the memory based DMA descriptor to be cleared once that descriptor has been processed by the DMA controller. Therefore, looped DMA chains may be created without redundant transfers taking place. In a multi-processor environment, this can be used to create a large number of virtual DMA channels. A particular processor or process can be assigned one or more of these virtual channels. They can then initiate a transfer by setting up the memory based descriptor as desired and then updating the DMA_LENGTH portion of the descriptor last. The next time that descriptor is processed then DMA will be performed and then the DMA_LENGTH value cleared. Then when the looped descriptors are processed on the next pass, no data transfer will be initiated.

In order to avoid having the DMA engine polling to aggressively on the local memory when the descriptors are looped, the DMA_DELAY register is provided. This register controls the number of clocks of delay between when one descriptor finishes and another is loaded. A larger value will provide more time for other local bus masters to occupy the local bus.

6.1.5 Chain Descriptor Loading

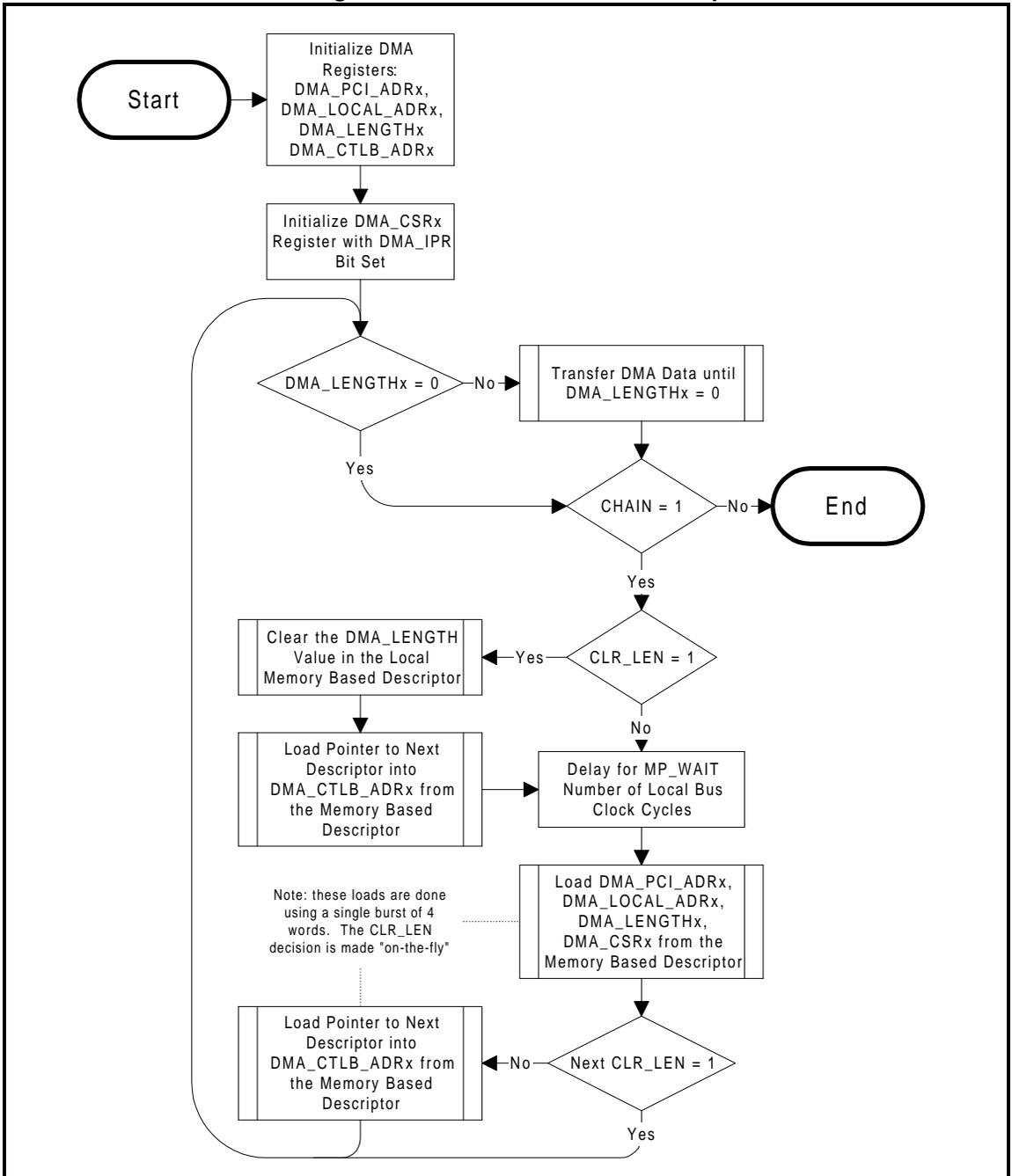
Chain descriptors are always loaded as a burst of 4 words. As the descriptor is loaded in, the EPC will check the CLR_LEN bit and decide, on the fly, if the last word of the burst (DMA_CTLB_ADR) is to be internally loaded or ignored (either way, the data transfer

DMA Controller

DMA Transfers

happens). If CLR_LEN of the incoming descriptor is set, then the new DMA_CTLB_ADR value will be ignored so that the old pointer remains intact. The old pointer is used to locate the memory based DMA_LENGTH value so that it can be cleared later. Clearing of the DMA_LENGTH value is done with a single write cycle and subsequent loading of the new DMA_CTLB_ADR is done as a single read cycle.

Figure 10: Flowchart of Basic DMA Operation



6.1.6 DMA Transfer Size

The DMA controller performs all transfers in word (32 bits) sizes.

For the V350EPC where the local bus is 16-bit wide, a 16 to 32-bit conversion will occur (the EPC will wait for a second 16-bit word before writing a 32-bit word to a PCI bus). DMA transfers begin and end on a 32-bit boundary and local bus transfers will be done 16 bits at a time (one data phase on the PCI bus will produce two data phases on the local bus). Byte and short (16-bit) boundaries are not supported. There is no performance penalty for this restriction. From a software standpoint, a programmer wishing to transfer byte data need only normalize the byte pointers to the next inclusive word boundary. This will result in transferring more *byte* data than necessary, however, since the PCI bus transfers data in 32-bit words, there is no negative performance effect.

6.1.6.1 Block Size

The largest block size that can be transferred in any single link is 4MB (minus 4 bytes). The DMA address generator provides a 25 bit count. Therefore, if a DMA transfer will cross a 32MB boundary then the address will wrap to the bottom of the same 32MB boundary instead of the next 32MB boundary. Larger block sizes or crossing of a 32MB boundary can easily be accomplished using the chaining feature of the DMA controller.

6.1.7 Relationship to the Data Transfer Apertures

The DMA Controller does not use the data transfer apertures. The write FIFOs, however, are shared between transfers initiated by a bus master through either the PCI-to-Local or Local-to-PCI apertures, and by the DMA Controller. Figure 11 shows the usage of the write FIFOs by both the DMA Controller and the data transfer apertures.

The read ahead FIFOs are *only* used during bus master read accesses through the data transfer apertures; they are not used during DMA operations at all.

6.1.8 Automatic DMA Throttling

The DMA Controller has a built in throttling mechanism to prevent it from monopolizing the write FIFO or target buses. The DMA Controller will not initiate a transfer if the target write FIFO is more than half full. Once the write FIFO is drained below one quarter full, DMA transfers involving the corresponding write FIFO will proceed.

6.1.9 Demand Mode DMA

Explicit hardware DMA request inputs can be used to throttle DMA transfers. This is provided by the DREQ_EN bits in the DMA_LENGTHx registers which use INTC and/or INTD as active low DMA request inputs. Hardware throttling works by allowing the source data of a DMA transfer to be loaded only when the external DREQx pin is asserted. The act of reading or writing local memory can be used as the DMA acknowledge.

DMA Controller

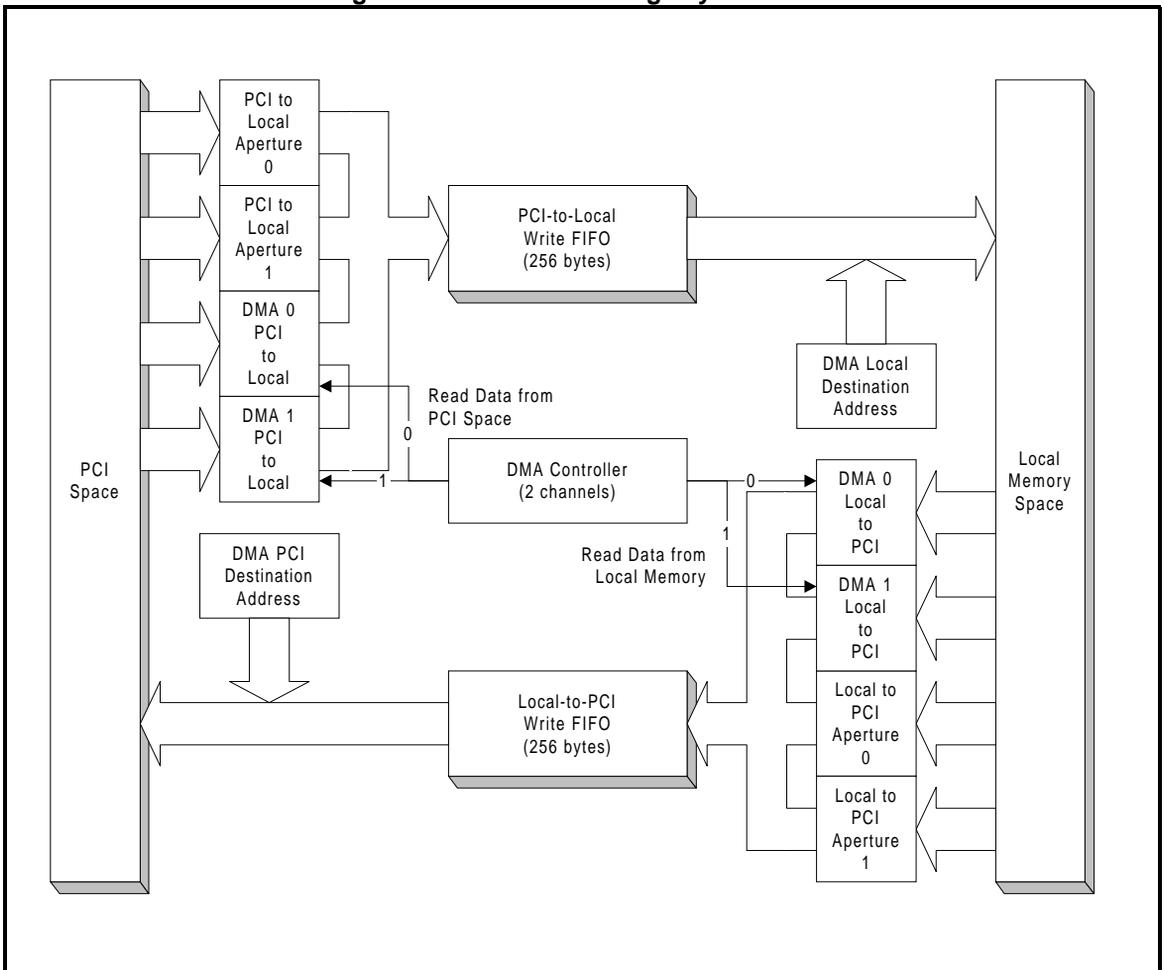
DMA Transfers

Since demand mode throttling works by gating the loading of source data, it is important that the when the local bus is the destination, it is capable of taking whatever could remain in the internal FIFO of the EPC if wait states via READY are to be avoided. In this scinerio, up to 32 words of data could be sitting in the

6.1.10 DMA Interrupts

Each DMA Controller channel will generate an interrupt whenever the transfer count reaches zero and the CHAIN bit in the corresponding DMA_CSRx register is zero (no further chains to complete). There are separate requests from each channel. The DMA interrupt requests are routed to both the Local interrupt control logic, and the PCI interrupt control logic. The DMA interrupt requests are latched by both the PCI interrupt status register and the Local interrupt status register and must be cleared independently in both status registers.

Figure 11: Write FIFO Usage by the DMA Controller



6.2 PROGRAMMING THE DMA CONTROLLER

The DMA Controller is programmed through two sets of registers, one for each of the two channels. The DMA Controller registers are accessible from both the Local and PCI sides of the EPC.

6.2.1 Setting the Starting Addresses

Two starting addresses must be initialized before DMA transfers can begin: the Local memory start address and the PCI memory start address.

The Local memory starting address is programmed through the DMA_LOCAL_ADDR0 (or DMA_LOCAL_ADDR1) register. The local memory address is word aligned. The EPC sets the 2 least significant bits of DMA_LOCAL_ADDRx to zero in order to force word alignment.

The PCI memory starting address is programmed through the DMA_PCI_ADDR0 (or DMA_PCI_ADDR1) register. The PCI memory address is word aligned. The EPC sets the 2 least significant bits of DMA_PCI_ADDRx to zero in order to force word alignment. The DMA Controller can only generate PCI Read and PCI Write commands on the PCI bus.

Both address registers are automatically incremented after each word is transferred. These registers may be read at any time to determine the current value of the pointers. The address registers may also be written to at any time, however, the programmer should check the "In Progress" bit (in the DMA_CSRx register) before modifying the pointers of a DMA operation in progress. Failure to do so could result in undefined bridge operation.

As previously mentioned, the DMA Controller is unrelated to the data transfer registers. The address translation, endian conversion, and prefetching options for the data transfer apertures will not affect DMA operation, *even if the DMA controller is performing a transfer involving memory locations that fall within a data transfer aperture.*

6.2.2 Setting the Transfer Count

The transfer count is stored in the DMA_LENGTHx register. The transfer count is in 32-bit words. The maximum value for the transfer count is 1 megaword or 4 megabytes (longer transfers may be performed using block chaining).

The transfer count is decremented after each word of data is transferred. The DMA_LENGTHx registers may be read at any time to determine the current value transfer count. The transfer count registers may also be written to at any time, however, the programmer should check the "In Progress" bit (DMA_IPR bit in the DMA_CSRx register) before modifying the transfer count for a DMA operation in progress. It is possible to halt DMA operation immediately, by setting the transfer count of a DMA process in progress to zero.

DMA Controller

Programming the DMA Controller

6.2.3 Setting the Transfer Direction

The DMA transfer direction - PCI-to-Local or Local-to-PCI - is set by the DIRECTION bit in the DMA_CSRx register. *This bit must not be changed while a DMA process is running* (as indicated by the state of the DMA_IPR bit for each channel.) Changing the DIRECTION bit while a DMA process is running will result in undefined bridge operation.

6.2.4 Byte Order Conversion

Each DMA channel can convert data byte order on-the-fly. The SWAP bits in the DMA_CSRx register control the three conversion options: 8-bit, 16-bit, and 32-bit. Byte order conversion by the DMA Controller is identical to that performed by the data transfer apertures.

6.2.5 Using DMA Block Chaining

A "linked list" of DMA block descriptors must be set up in local memory before initiating a chained DMA transfer. The individual descriptors do not need to occupy contiguous locations in local memory, since each descriptor includes an absolute pointer to the next (in the DMA_CTLB_ADDRx register). The address of the *second* descriptor in the chain is written to the DMA_CTLB_ADDRx register for the corresponding DMA channel. The parameters for the first block to be transferred are written directly into the DMA channel's address, transfer count, and control/status registers.

Each descriptor in the chain must have the CHAIN bit set in the DMA_CSRx register. The last block must have the CHAIN bit cleared, to indicate to the DMA Controller that it is to terminate the process after this block is complete. There is no limit to the number of blocks in a DMA chain, in fact "ring buffers" may be easily implemented by pointing the "last" DMA block descriptor back to the "first".

6.2.6 Starting DMA Operation

A DMA channel begins operation when the DMA Initiate Process bit (DMA_IPR) is set in the corresponding DMA_CSRx register. This bit is automatically cleared when the transfer count expires and there are no further chains to process (i.e. CHAIN = 0). Writing a zero to DMA_IPR has no affect on operation and is ignored. Once a DMA transfer has started, the corresponding DMA registers must not be written. The only exception is the "Early Termination" outlined below.

6.2.7 Early Termination of a DMA Process

A DMA process may be terminated in advance of completion by setting the ABORT bit in the corresponding DMA_CSRx register.

Using the ABORT bit will also prevent the next chain from being fetched when chaining is

enabled. The ABORT bit can also be used as a "pause" bit. Once paused, the same DMA can be finished from where it left off by simply setting the DMA_IPR bit again.

Caution: There may be a delay between when the ABORT bit is set and when the DMA_IPR bit is cleared (this is due to having source data sitting in the internal FIFO waiting to write it to the destination). Once an ABORT has been initiated, software should poll DMA_IPR and wait for it to be cleared.

6.2.8 Setting Priority Between the DMA Channels

The EPC provides a simple mechanism for setting the priority between the two DMA channels. A PRIORITY bit is provided in each channel's DMA_CSRx register. Table 9 describes the priority options that are based on the state of these bits.

Table 9: DMA Channel Priority Options

Channel 0 PRIORITY bit	Channel 1 PRIORITY bit	Result
0	0	First Come, First Served. Priority is assigned to the channel that is the first to initiate a DMA process (have its DMA_IPR bit set).
0	1	Channel 1 Priority. DMA channel 1 has the highest priority and will interrupt a DMA channel 0 transfer in process. DMA channel 0 will continue its process only after DMA channel 1 completes its transfer.
1	0	Channel 0 Priority. Same as above, except channel 0 has priority over channel 1.
1	1	Rotating Priority. The two channels rotate priority based on the least recently granted channel. When both channels are running a DMA process, they will alternate bus accesses.

DMA Controller

Programming the DMA Controller

Chapter 7

PCI Bus Interface

The EPC implements the PCI bus according to the revision 2.1 PCI Specification published by the PCI Special Interest Group. This section assumes a familiarity with the PCI bus specification and only describes performance and exception handling issues.

7.1 TARGET TRANSFERS

The EPC acts as a PCI target (slave) when it bridges a read or write access to one of the PCI-to-Local data transfer apertures. There are two basic types of target transfers: reads and writes.

7.1.1 Target Reads

The following command types fall under the category of target reads: Memory Read, Memory Read Multiple, I/O Read, Configuration Read, Memory Read Line, and Interrupt Acknowledge.

Upon receipt of a PCI-to-Local read request, the EPC will attempt to access the local bus by asserting the local bus request signal (BREQ or HOLD). The EPC supports delayed reads when the RD_POST_INH bit is clear. This causes an immediate retry when a PCI read is initiated where there is no valid data present in the prefetch buffer.

If read posting is disabled, then no retry will be performed. Instead, $\overline{\text{TRDY}}$ will be delayed until the local cycle produces data.

PCI burst reads that cross a 1k byte address boundary will be broken into two smaller bursts by the EPC. This is done by issuing a PCI disconnect to the initiator as the burst crosses the 1k byte boundary.

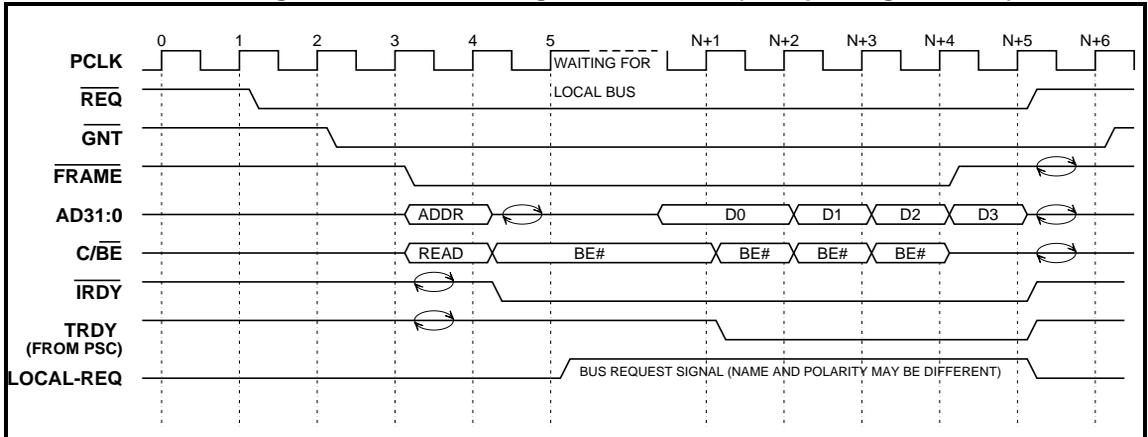
PCI-to-Local I/O reads require one additional clock of address decoding when using the fine grain I/O PCI-to-Local aperture (see “DOS Compatibility”).

Target mode reads through the PCI-to-Internal Register aperture (PC_IO_BASE) will have 3 wait-states inserted between each data cycle.

PCI Bus Interface

Target Transfers

Figure 12: EPC PCI Target Mode Reads (read posting disabled)



7.1.2 Target Writes

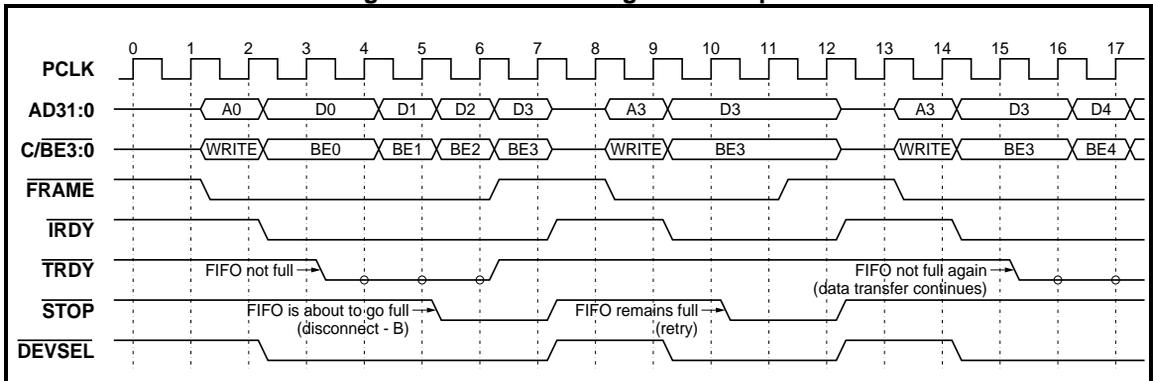
The following command types fall under the category of target writes: Memory Write, Memory Write and Invalidate, I/O Write, Configuration Write and Special Cycle.

Upon receipt of a PCI-to-Local write request, the EPC will attempt to complete the write by placing the data in the PCI-to-Local write FIFO. The EPC will complete each data phase by asserting TRDY until either the write completes (see Figure 13), or there is no room left in the write FIFO. If the PCI-to-Local write FIFO becomes full, the EPC will issue a PCI disconnect (see below).

PCI burst writes that cross a Burst boundary, as determined by PBRST_MAX in the FIFO_CFG register, will be broken into two smaller bursts by the EPC. This is done by issuing a PCI Disconnect to the initiator as the burst crosses the Burst boundary.

PCI-to-Local I/O writes require one additional clock of address decoding when using the fine grain I/O PCI-to-Local aperture (see "DOS Compatibility").

Figure 13: EPC PCI Target Mode Aperture Writes



7.1.3 PCI Exceptions During EPC Target Cycles

The EPC only generates recoverable exceptions when acting as a PCI target.

7.1.3.1 Recoverable Exception: Target Disconnect

The EPC will generate a PCI Disconnect under the following conditions:

- A burst in progress completely fills the PCI-to-Local write FIFO.
- A PCI burst is attempted to a PCI-to-Local aperture that is disabled for bursting.
- A burst in progress crosses a 1k byte burst boundary, or exceeds the programmed maximum burst length for the aperture (as determined by the PBRST_MAX field in FIFO_CFG).
- A burst in progress is pre-empted.
- The time between two $\overline{\text{TRDY}}$ assertions in a burst exceeds 8 PCI clocks.

7.1.3.2 Recoverable Exception: Target Retry

The EPC will generate a PCI Retry under the following conditions:

- A PCI write is attempted to an already full PCI-to-Local write FIFO.
- A PCI access is attempted to the Local space while the local master is attempting a PCI access through the EPC. The EPC issues a Retry to prevent a possible deadlock.
- A PCI access is attempted while the EPC is initializing from the serial EEPROM.
- Posted reads are enabled (Bit 15 in PCI_MAPx is clear) and a read to a location not previously prefetched is performed.

7.1.4 PCI Access of EPC Internal Registers

The internal registers of the EPC can be accessed in two ways from the PCI bus:

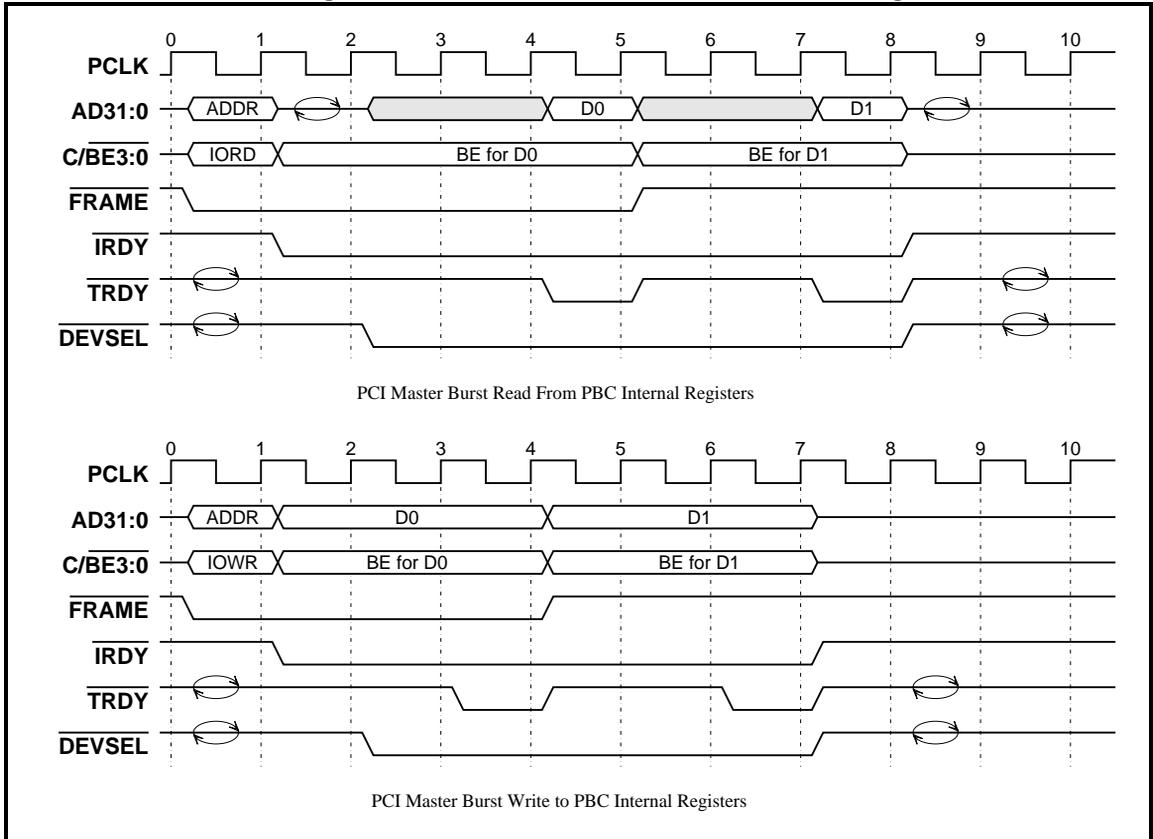
- Through PCI "Configuration Space" when the IDSEL input is high during the address phase and a configuration cycle type is identified on the C/BE[3:0] signals.
- Through the PCI_IO_BASE base register address as either I/O or memory.

The following waveforms illustrate the timing of a PCI master access to the internal registers.

PCI Bus Interface

Initiator Transfers

Figure 14: PCI Read and Write of Internal EPC Registers



7.2 INITIATOR TRANSFERS

The EPC acts as a PCI initiator (master) when it bridges a read or write access to one of the Local-to-PCI data transfer apertures. There are two basic types of initiator transfers: reads and writes.

7.2.1 Initiator Reads

The following command types fall under the category of initiator reads: Memory Read, Memory Read Multiple, I/O Read, Configuration Read, Memory Read Line, and Interrupt Acknowledge.

The EPC will attempt to perform the fastest PCI read cycle possible ($\overline{\text{IRDY}}$ wait states are not inserted by the EPC) as shown in Figure 15 through Figure 18.

PCI reads always require a lead-off wait-state to allow for bus turnaround between the address and data phases. A PCI read cycle may be extended by slower targets not returning $\overline{\text{TRDY}}$ until the target is ready to return data.

7.2.2 Initiator Writes

The following command types fall under the category of initiator reads: Memory Write, Memory Write and Invalidate, I/O Write, Configuration Write and Special Cycle.

The EPC will attempt to perform the fastest PCI write cycle possible as shown in Figure 19 through Figure 22. Unlike PCI read cycles, PCI writes do not require bus turnaround between the address and data phases. A PCI write cycle may be extended by slower targets not returning $\overline{\text{TRDY}}$ until the target is ready to receive data.

When the LB_WR_PCI bits in FIFO_CFG register are programmed to '00', the local bus write cycle will cause a PCI bus request as soon as possible. Other settings of LB_WR_PCI will cause REQ assertion to be delayed.

Table 10: FIFO control for Local Bus Write to PCI Bus Aperture 0 and 1

LB_WR_PCI	Result
00	Assert PCI bus request immediately whenever the corresponding FIFO is not empty
01	Reserved
10	Assert PCI bus request whenever the Local bus to PCI corresponding FIFO has 3 or more words of data pending
11	Assert PCI bus request whenever the Local bus to PCI corresponding FIFO has 3 or more words of data pending <i>or</i> the FIFO is not empty and the local bus master ends a burst write cycle to the FIFO

PCI Bus Interface

Initiator Transfers

Figure 15: V350EPC (i960 mode) Initiated PCI Read Cycle

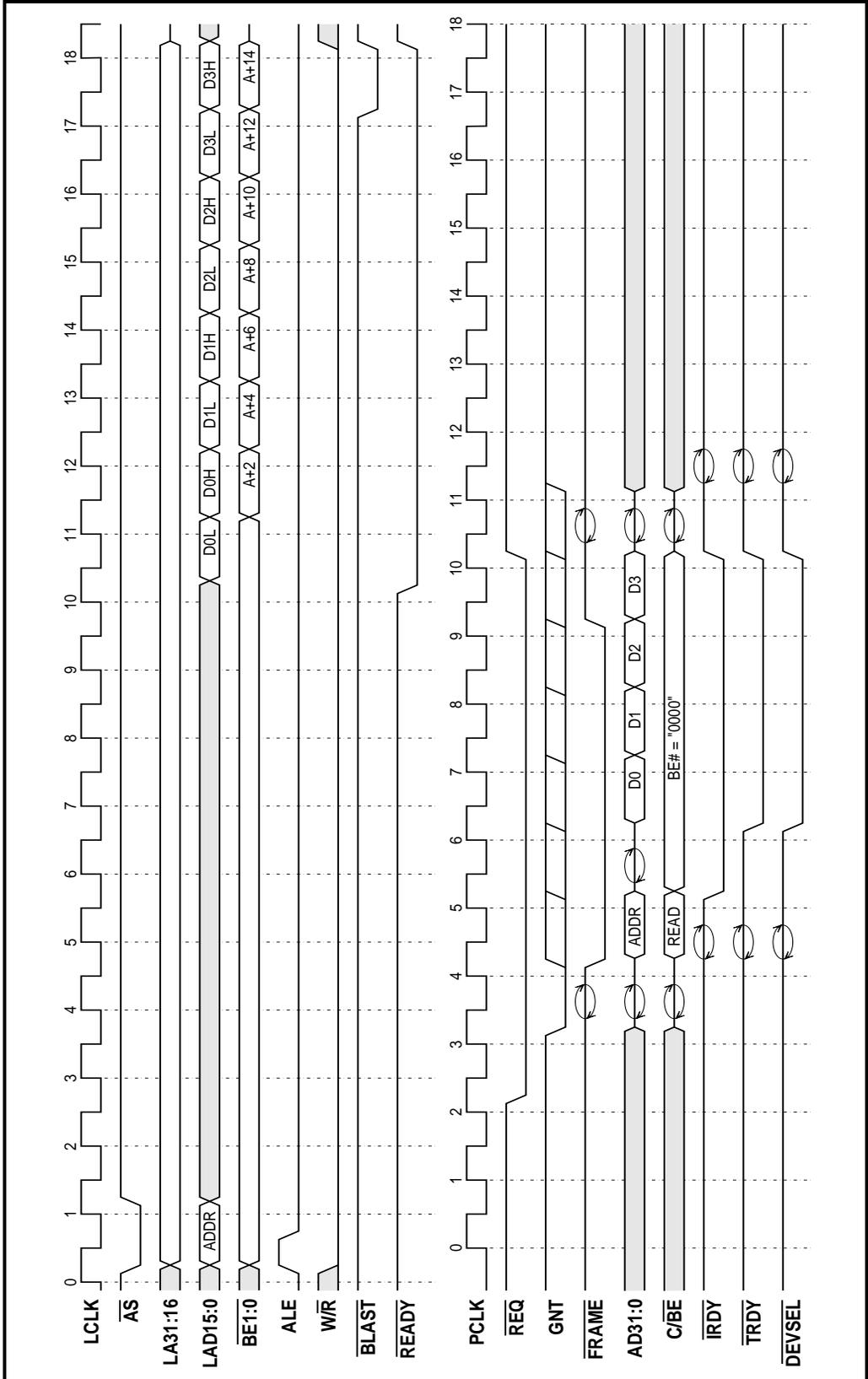
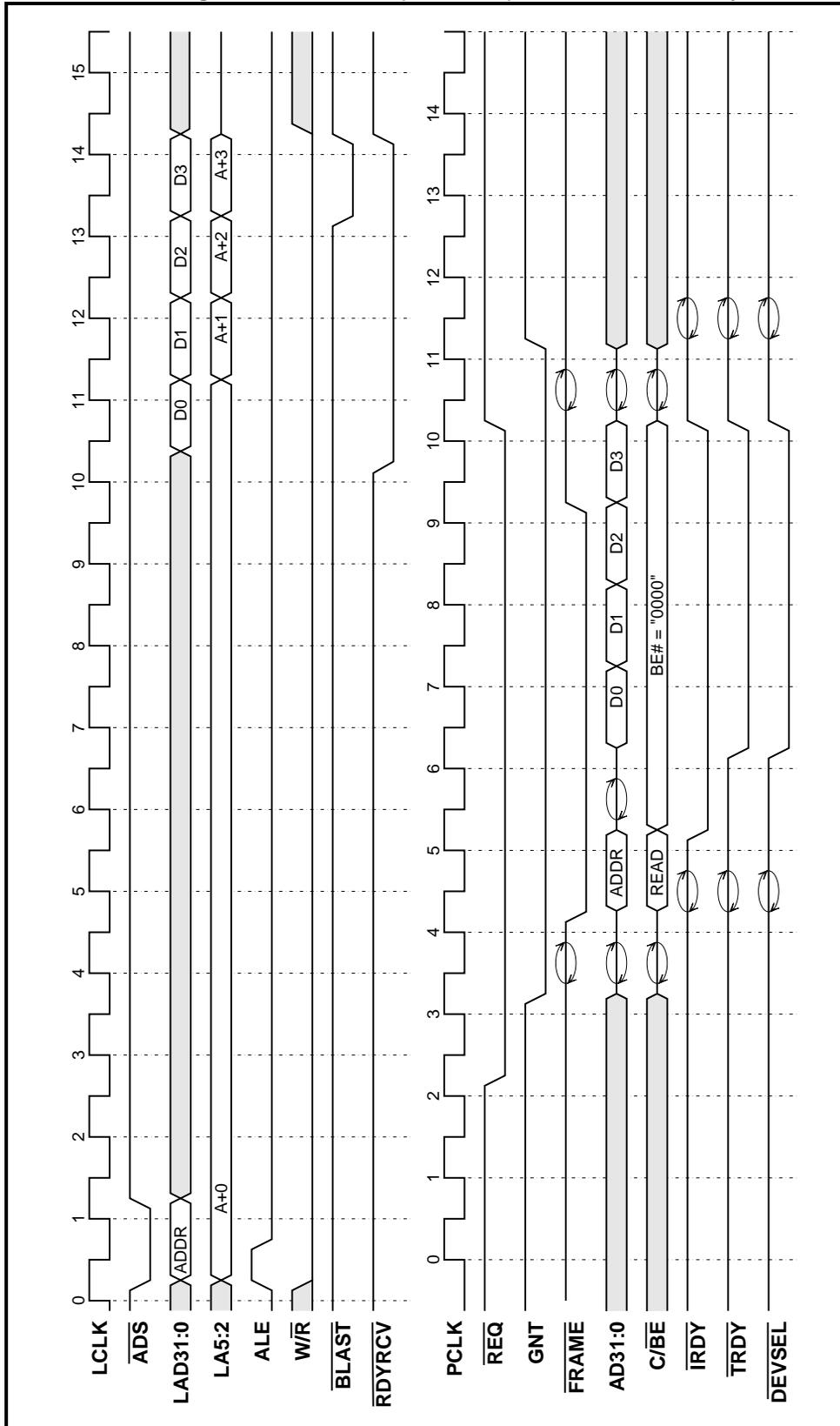


Figure 16: V350EPC (961 mode) Initiated PCI Read Cycle



PCI Bus Interface

Initiator Transfers

Figure 17: V360EPC (962 mode) Initiated PCI Read Cycle

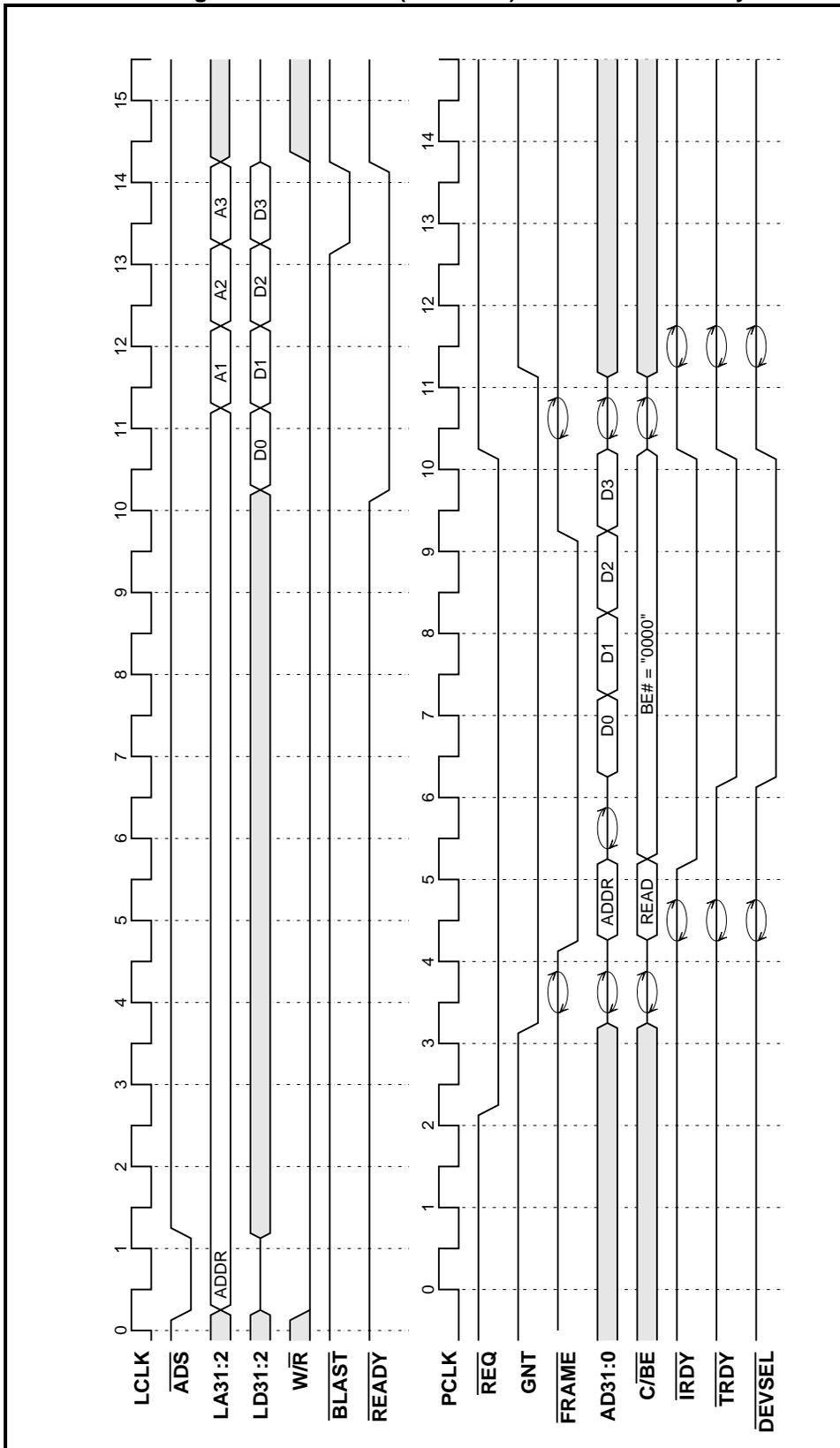
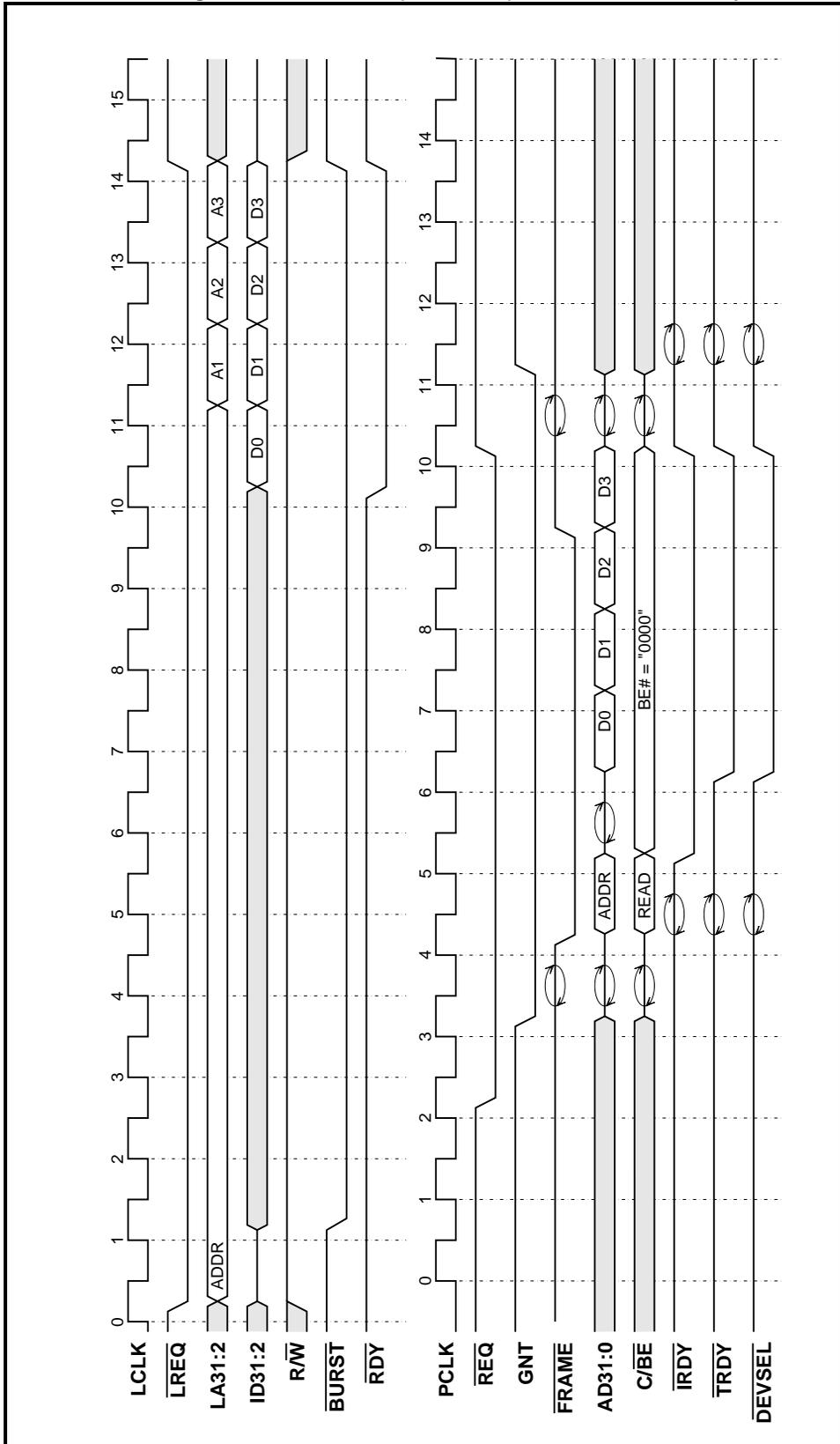


Figure 18: V360EPC (292 mode) Initiated PCI Read Cycle



PCI Bus Interface

Initiator Transfers

Figure 19: V350EPC (960 mode) Initiated PCI Write Cycle

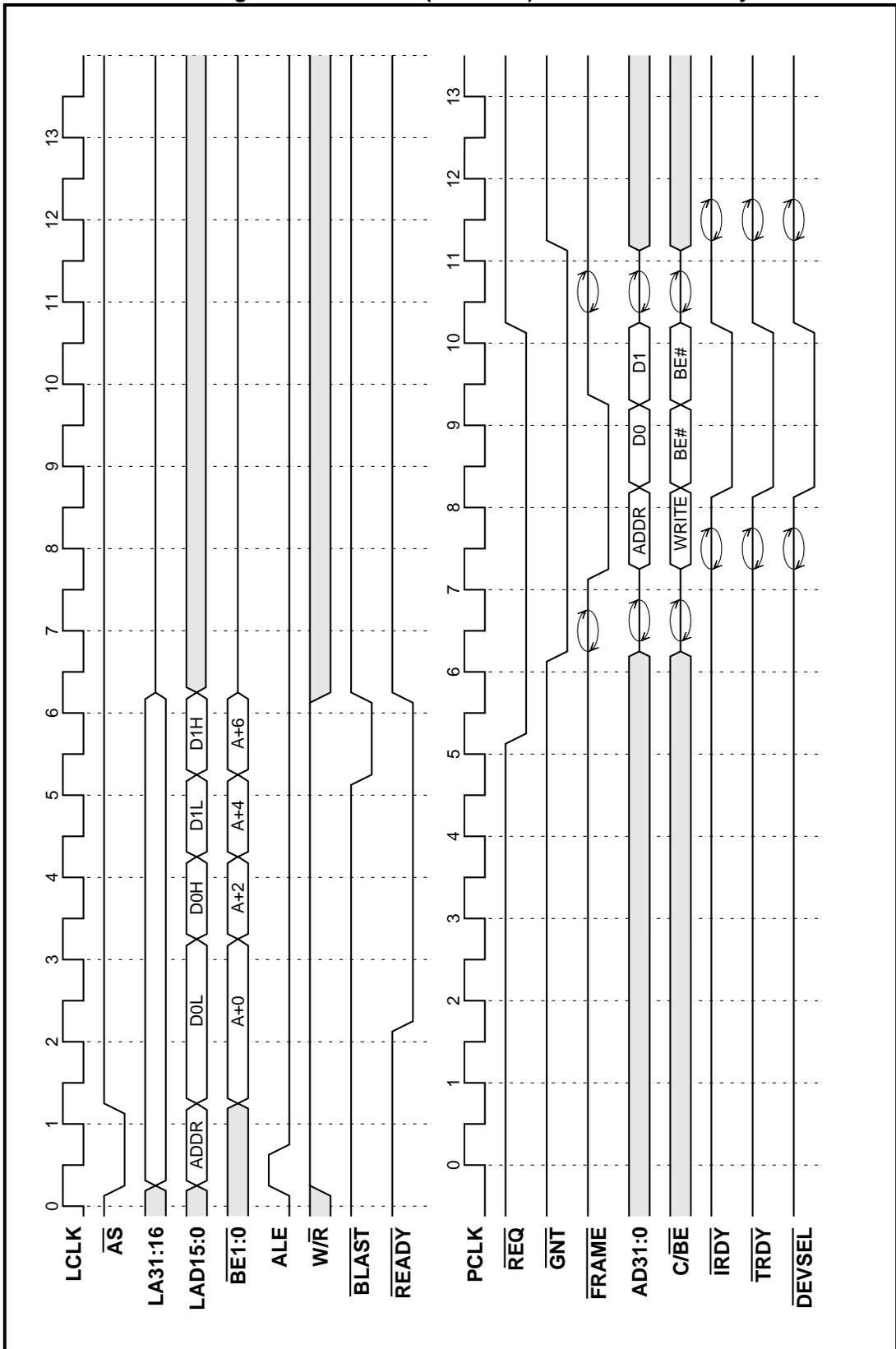
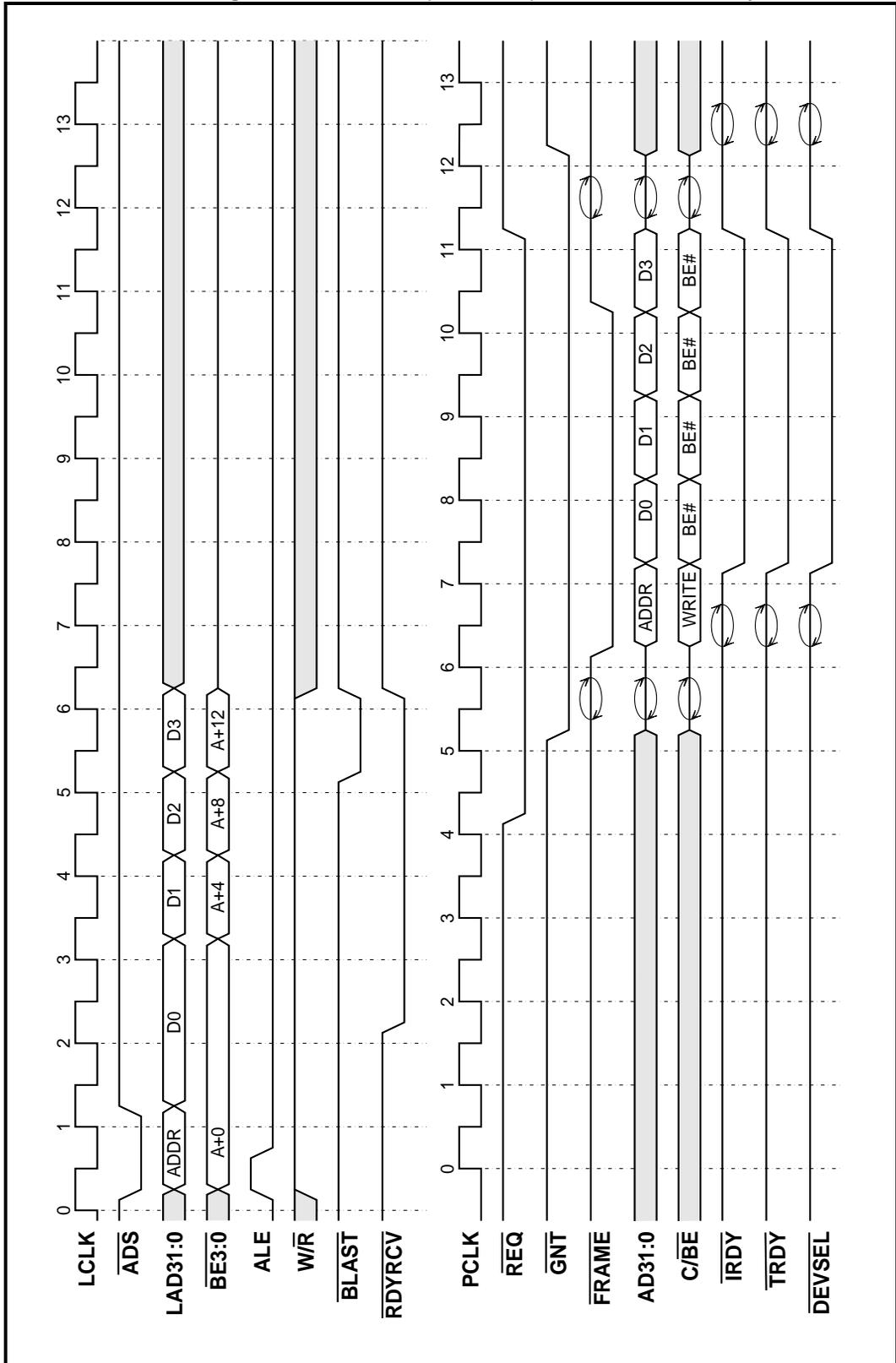


Figure 20: V350EPC (961 mode) Initiated PCI Write Cycle



PCI Bus Interface

Initiator Transfers

Figure 21: V360EPC (962 mode) Initiated PCI Write Cycle

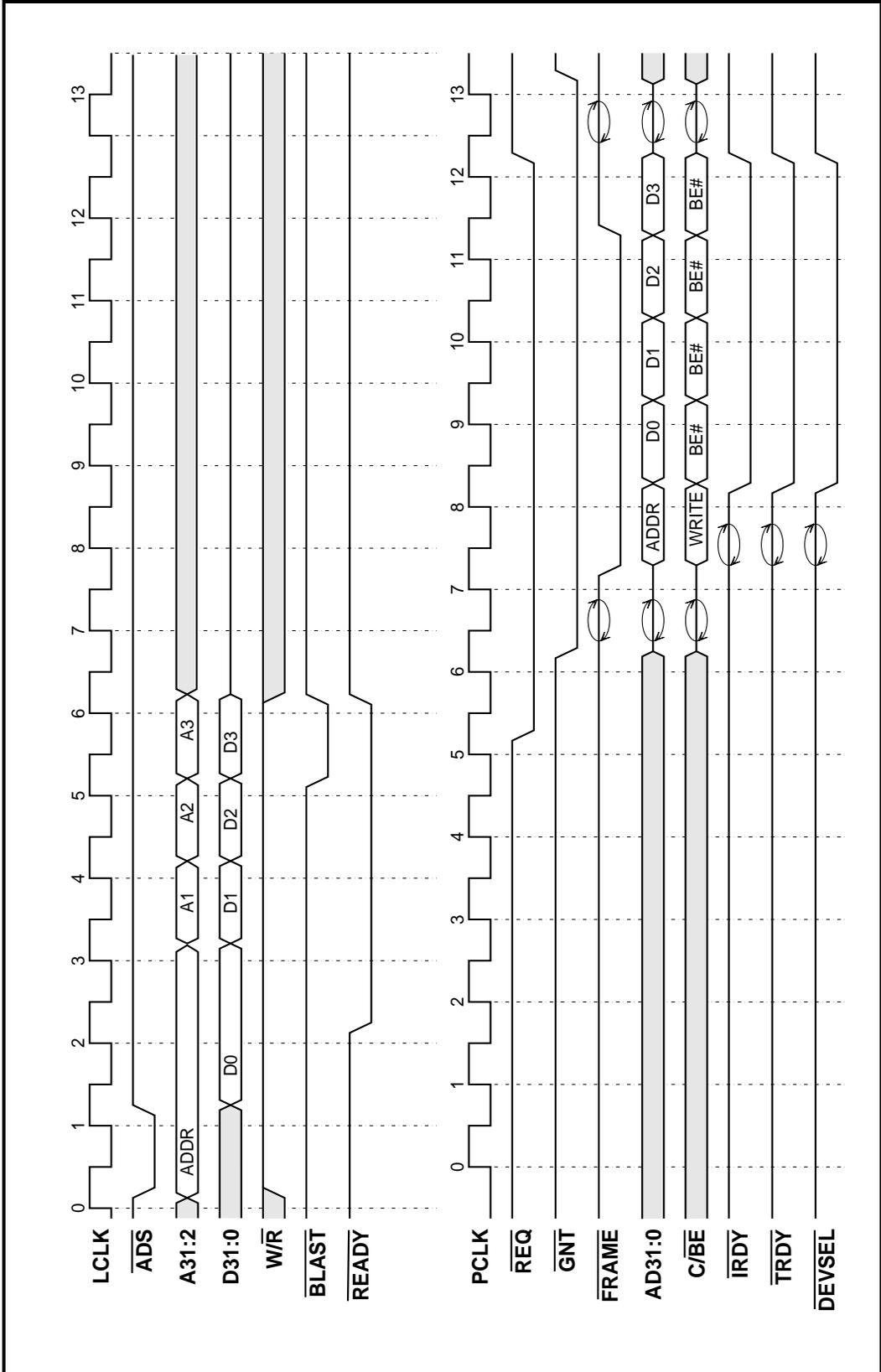
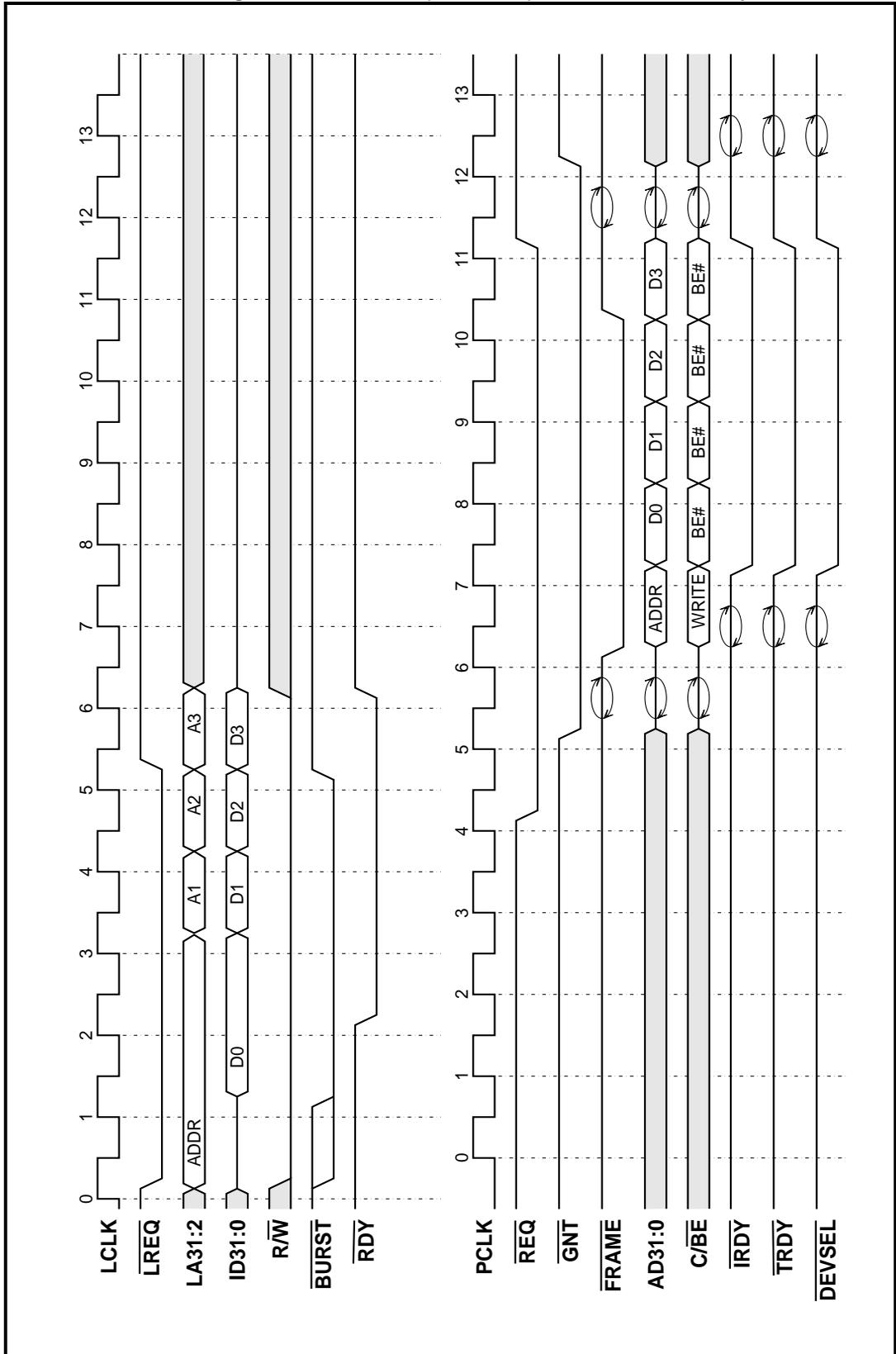


Figure 22: V360EPC (292 mode) Initiated PCI Write Cycle



7.2.3 PCI Exceptions During EPC Initiated Cycles

There are two categories of exceptions that can be generated on the PCI bus:

Fatal Exceptions: These are conditions from which recovery is not possible such as Master Abort (attempted access to non-existent device) and Target Abort (unrecoverable error within a target).

Recoverable Exceptions: These are exceptions from which recovery is guaranteed such as Disconnect and Retry. A target disconnect, for example, is used by a target to discontinue a burst that is longer than it can accommodate in a single transfer. This exception is recovered by simply starting another PCI cycle to complete the transaction.

Fatal exceptions are reported to the local processor by a processor interrupt. Recoverable exceptions are handled transparently by the EPC.

7.2.3.1 Fatal Exception: Master Abort (Reads)

A Master Abort occurs when the EPC does not receive a DEVSEL in response to a read or write attempt. This typically results from an attempt to access a non-existent device.

The EPC responds to Master Abort conditions during a read as follows:

- READY is returned to the local processor to “unlock” the local bus. The data returned is FFFFFFFFH. Optionally, a PCI read exception interrupt can be generated.
- The M_ABORT bit in the PCI_STAT register is set.

7.2.3.2 Fatal Exception: Master Abort (Writes)

Since PCI writes are posted by the EPC, it is conceivable that a Master Abort exception can occur long after the local bus write completes. In this case it makes no sense to generate a bus error on the local bus.

The EPC responds to Master Abort conditions during a write as follows:

- A PCI write exception interrupt is generated (maskable).
- The M_ABORT bit in the PCI_STAT register is set.

7.2.3.3 Fatal Exception: Target Abort (Reads)

A Target Abort occurs when the EPC receives a target abort indication from the currently addressed target. Target Abort is only used in the most extreme cases since it implies that the target is in the system (i.e. a DEVSEL is received), however it is permanently incapable of responding to the request.

The EPC responds to Target Abort conditions during a read as follows:

- READY is returned to the local processor to “unlock” the local bus. The data returned is indeterminate. Optionally, a PCI read exception interrupt can be generated.
- The T_ABORT bit in the PCI_STAT register is set.

As a target, the PCI will never generate a target abort.

7.2.3.4 Fatal Exception: Target Abort (Writes)

Since PCI writes are posted by the EPC, it is conceivable that a Target Abort exception can occur long after the local bus write completes. In this case it makes no sense to generate a bus error on the local bus.

The EPC responds to Target Abort conditions during a write as follows:

- A PCI write exception interrupt is generated (maskable).
- The T_ABORT bit in the PCI_STAT register is set.

7.2.3.5 Recoverable Exception: Target Disconnect

A target responds with a Target Disconnect when it is no longer capable of receiving data during a transaction. For example, if a target's write buffer became full during a long burst it can issue a Target Disconnect to break the burst into smaller “pieces”. Target Disconnect informs the initiator that the burst can be restarted at the point at which the disconnect occurred (i.e. the initiator does not need to repeat the entire burst).

The EPC handles Target Disconnect transparently by simply restarting the transaction at the point at which it was “disconnected”. No errors are reported since no data is lost.

7.2.3.6 Recoverable Exception: Target Retry

A target responds with a Target Retry when it is not capable of receiving data during a transaction. For example, if a target is currently too busy to respond to a PCI request, it can issue a Retry to tell the initiator “come back later”. Target Retry informs the initiator that the burst must be restarted from the beginning (i.e. the initiator needs to repeat the entire burst).

The EPC handles Target Retry transparently by simply restarting the transaction at the point at which it was “retried”. No errors are reported since no data is lost.

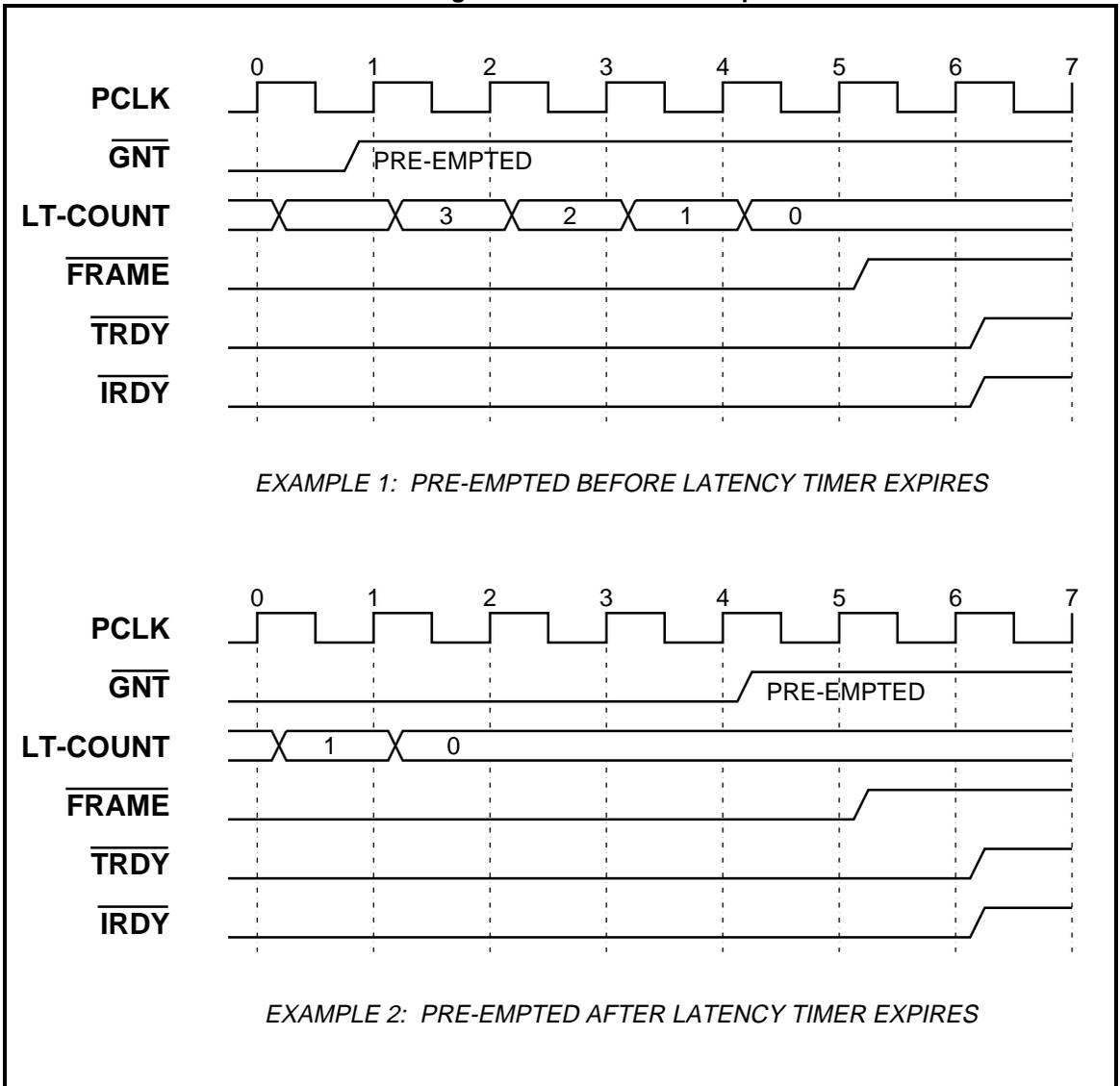
7.2.4 Initiator Pre-Emption

A PCI initiator is said to be pre-empted whenever its $\overline{\text{GNT}}$ line is deasserted during an active transfer. When the $\overline{\text{GNT}}$ is deasserted, the EPC checks its latency timer. If the latency timer has not expired, the EPC will maintain ownership of the bus until the timer reaches zero. If the latency timer has reached zero, the EPC will relinquish the PCI bus following the next data transfer. Both cases are shown in Figure 23.

PCI Bus Interface

Initiator Transfers

Figure 23: Initiator Pre-Emption



Chapter 8

Local Bus Interface

The EPC family devices are designed to be directly connected to i960®(Am29K™) family processors without the use of any "glue logic". The local bus protocol used by the EPC devices duplicate the bus protocol of the corresponding i960(Am29K) processor. For example, the EPC uses the same bus protocol as the i960(Am29K).

The EPC acts as both a local bus target and as a local bus master. This section of the manual describes the local bus interface for all versions of the EPC family.

8.1 TARGET MODE

The local bus interface is said to be in *target mode* when the EPC is responding to read and write requests from the local bus master (normally an i960(Am29K) family processor). There are currently three i960 processor buses supported: i960Sx (V350EPC in V960 mode), i960Jx (V350EPC in V961 mode), i960Cx/Hx (V360EPC in 962 mode) and two AMD processor buses:Am29030/40(V360EPC in 292 mode).

8.1.1 Local Bus CPU Configuration

Some local bus CPU devices, such as the i960Cx, have programmable bus parameters such as wait states and bus size. The EPC devices work at the largest bus width of the processor which is 32 bits for all EPC members except the V350EPC (960 mode) which is 16 bits to match the i960Sx 16 bit data width. Also, the EPC devices control the number of wait states for each access and do not rely on the processors programmable wait state generators. Therefor, for the regions that access the EPC, the CPU should be set up for 32 bit bus width (16 for the V350EPC), external ready enabled and minimum wait state settings.

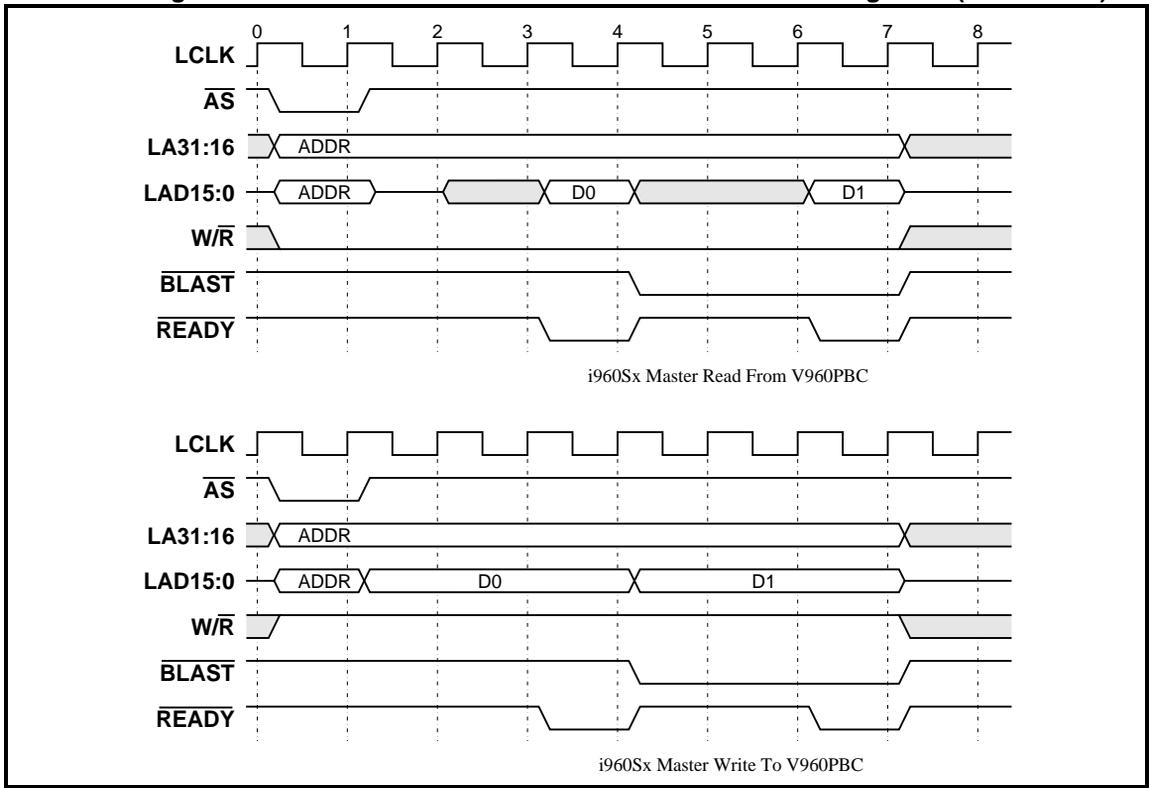
8.1.2 Local Reads and Writes to Internal Registers

Reads and writes to the EPC's internal registers occur through the Local-to-Internal Register aperture defined by the LB_IO_BASE register. Internal register read/writes always take 2 wait-states in burst mode. Since the internal registers can be accessed from both PCI and local buses, the number of initial wait states for local access is dependent on the interaction of internal arbitration with PCI. To accomodate this, control of the local bus READY signal is provided by the EPC. Burst accesses are permitted to internal registers. Figure 24 through Figure 27 show a local-to-internal register read/write access for each of the EPC components.

Local Bus Interface

Target Mode

Figure 24: Local Master Read/Write to Internal V350EPC Registers (i960Sx Bus)¹



1. ALE signal required but not shown

Figure 25: Local Master Read/Write to Internal V350EPC Registers (i960Jx Bus)

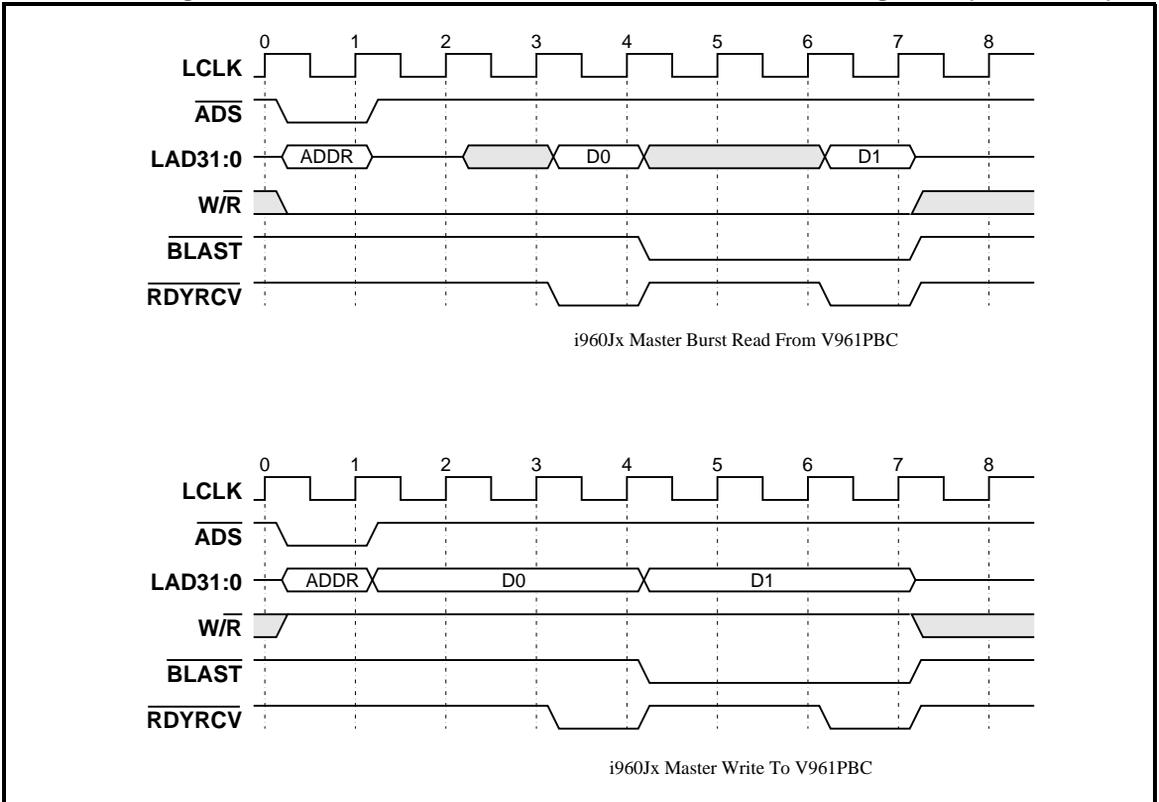
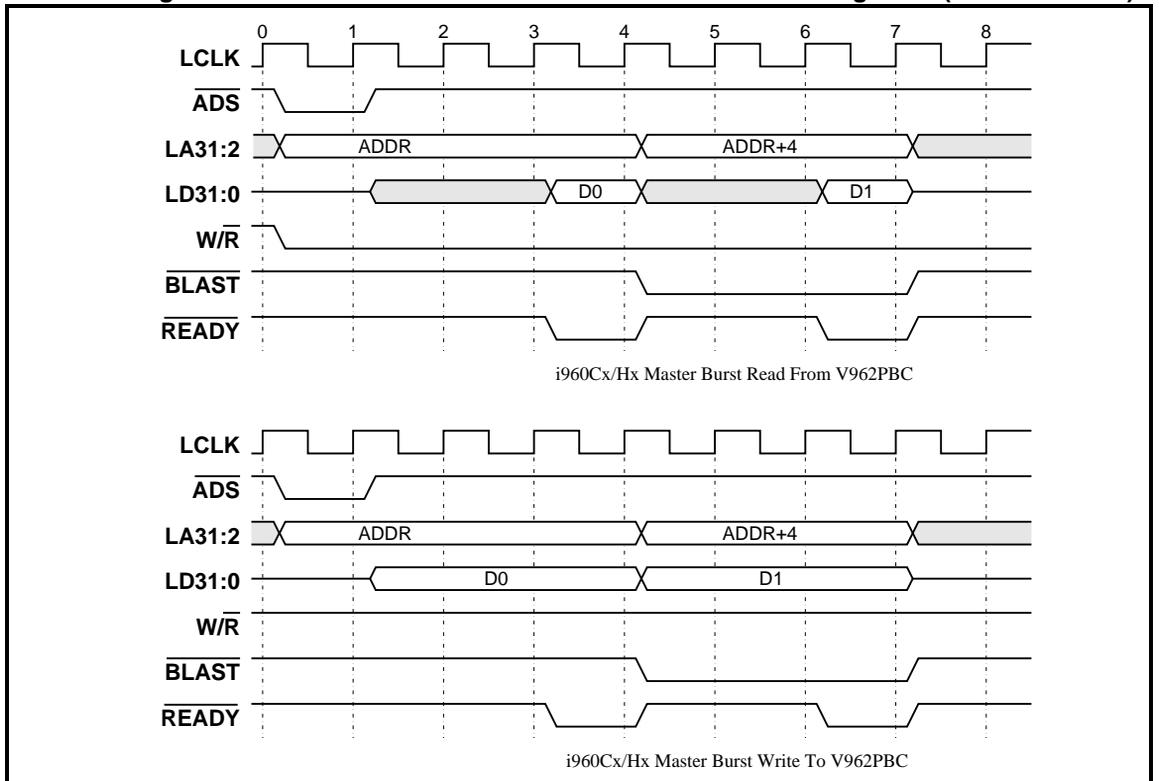


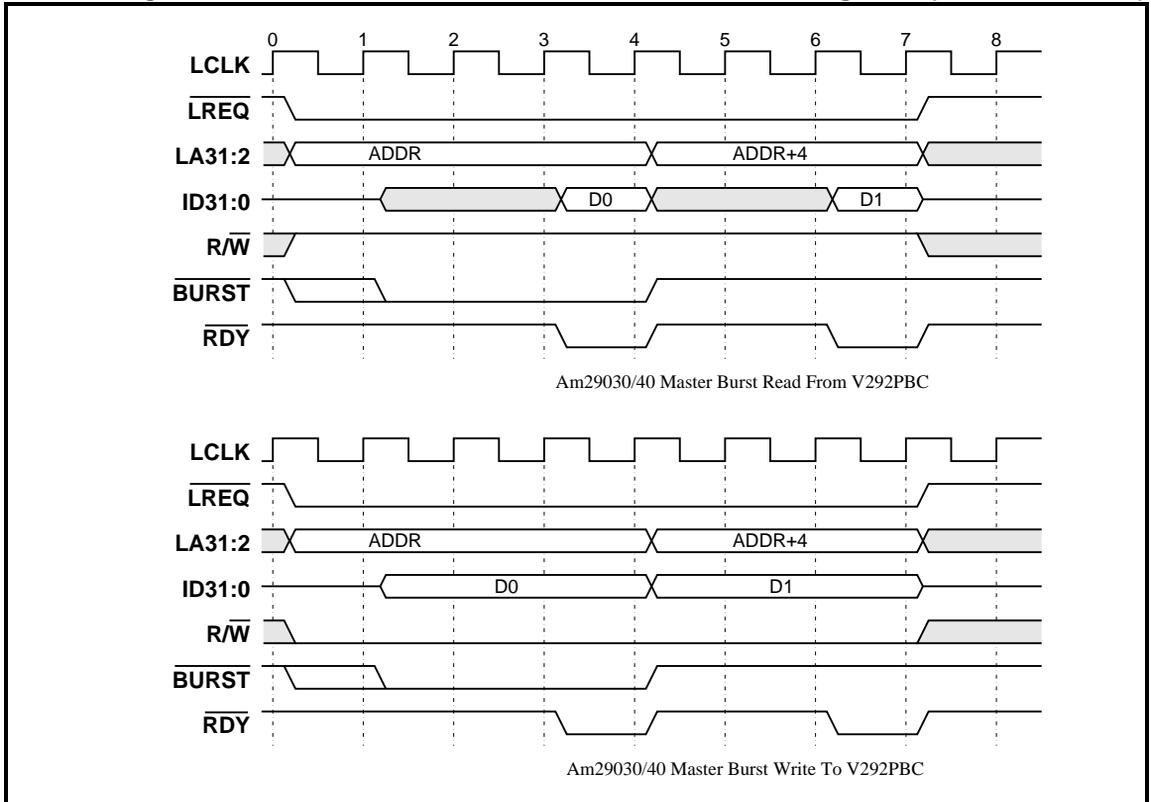
Figure 26: Local Master Read/Write to Internal V360EPC Registers (i960Cx/Hx Bus)



Local Bus Interface

Target Mode

Figure 27: Local Master Read/Write to Internal V360EPC Registers (Am29030/40 Bus)



8.1.3 Local Read from Local-to-PCI Apertures

Local reads from PCI space take place through the Local-to-PCI apertures. The initial read to any aperture will incur the full penalty of waiting for the corresponding read to complete on the PCI bus. The local bus is held NOT READY until the first datum is available from the PCI bus. Subsequent reads will complete as fast as is possible based on the following:

- How many words, if any, are buffered in the read FIFO.
- Whether or not prefetching is turned on for the aperture.
- How large a delta there is in the operating frequencies of the PCI and local bus.

The fastest local bus reads will complete is one wait-state for address-to-data and then zero wait-states for data to data. Zero wait-state reads will only occur when the read in progress is "draining" prefetched data from a Local-to-PCI read FIFO.

The EPC will attempt to perform the fastest local read cycle possible as shown in Figure 15 through Figure 22.

8.1.4 Local Write to Local-to-PCI Apertures

Local writes to PCI space take place through the Local-to-PCI apertures. Writes are posted

at zero wait-state until no room is left in the write FIFO. Attempts to write to a full write FIFO will result in the local bus being held NOT READY until room becomes available. Figure 32 through Figure 35 show the fastest local bus writes to the Local-to-PCI write FIFO.

Local Bus Interface

Target Mode

Figure 28: V350EPC (960 mode) PCI Posted Read from Local Bus

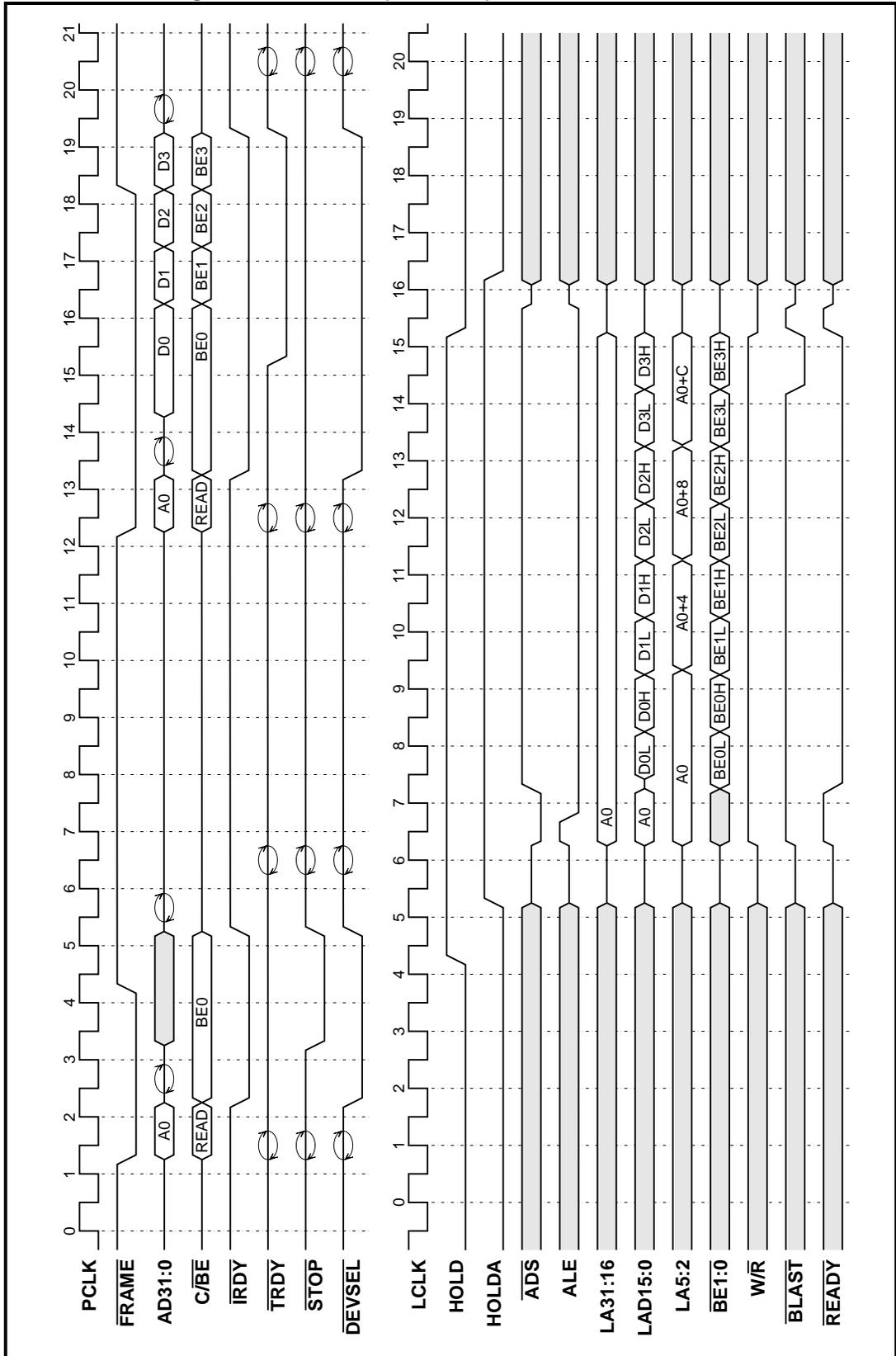
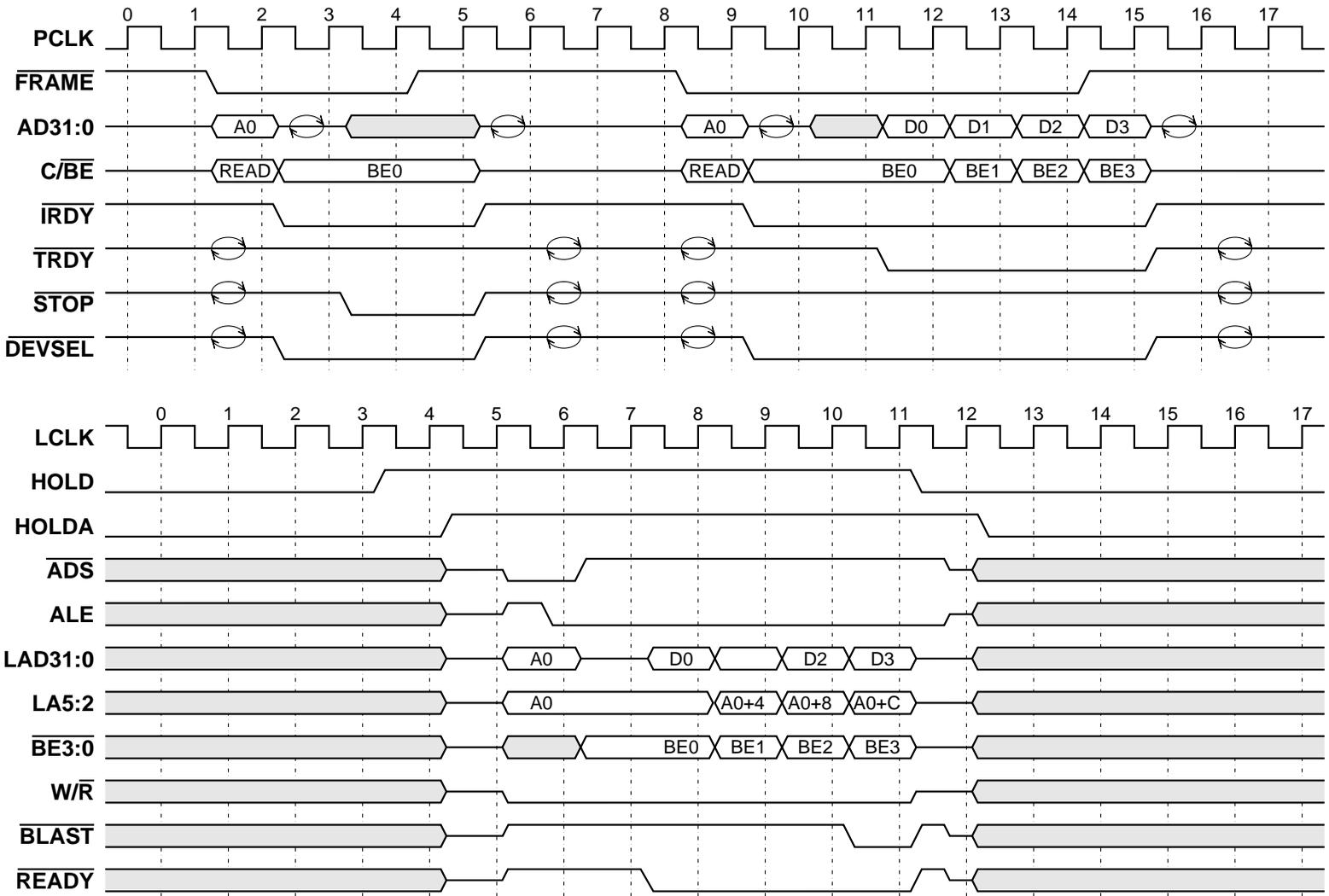


Figure 29: V350EPC (961 mode) PCI Posted Read from Local Bus



Local Bus Interface

Target Mode

Figure 30: V360EPC (962 mode) PCI Posted Read from Local Bus

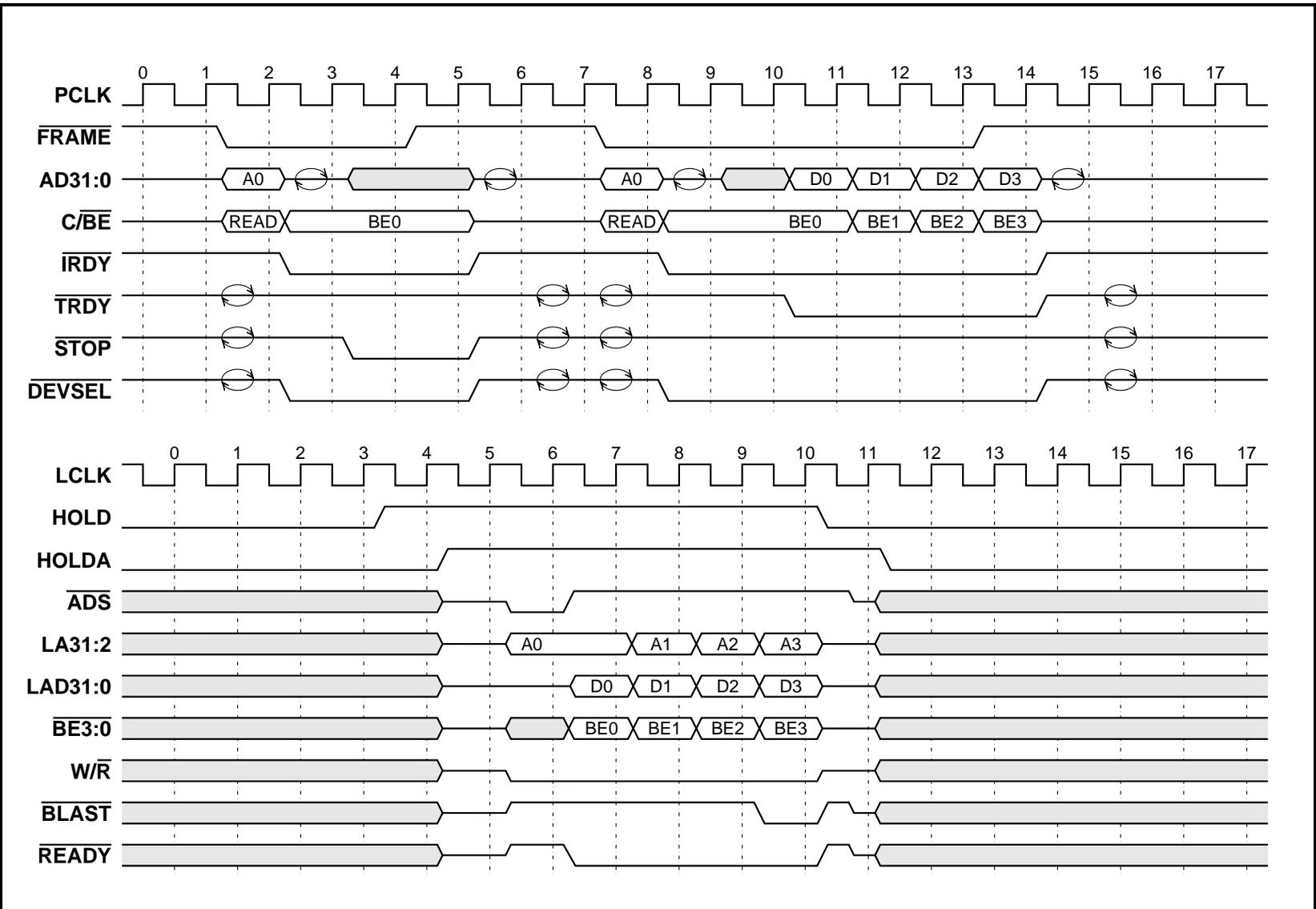
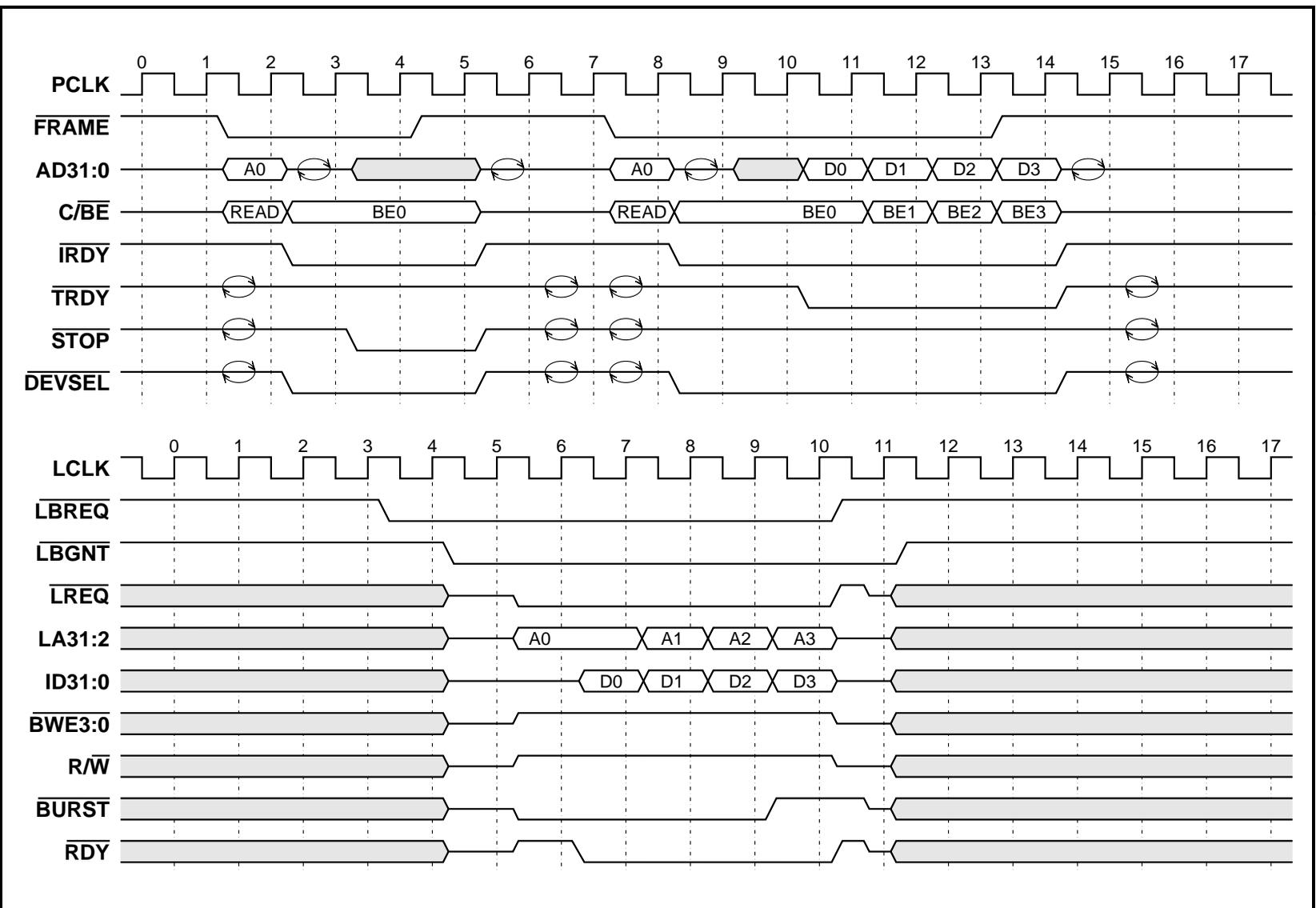


Figure 31: V360EPC (292 mode) PCI Posted Read from Local Bus



Local Bus Interface

Target Mode

Figure 32: V350EPC (960 mode) Initiated Local Write Cycle

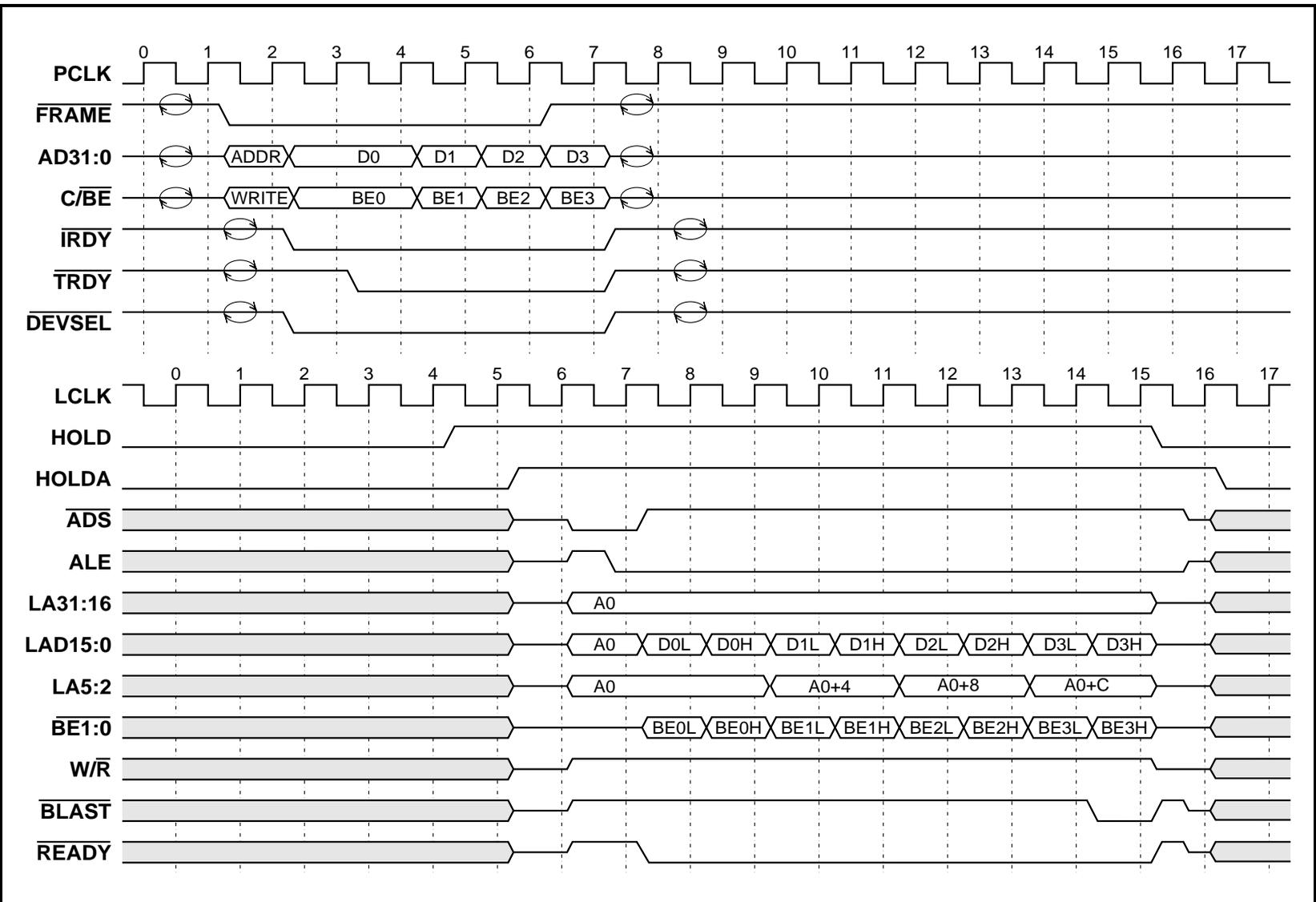
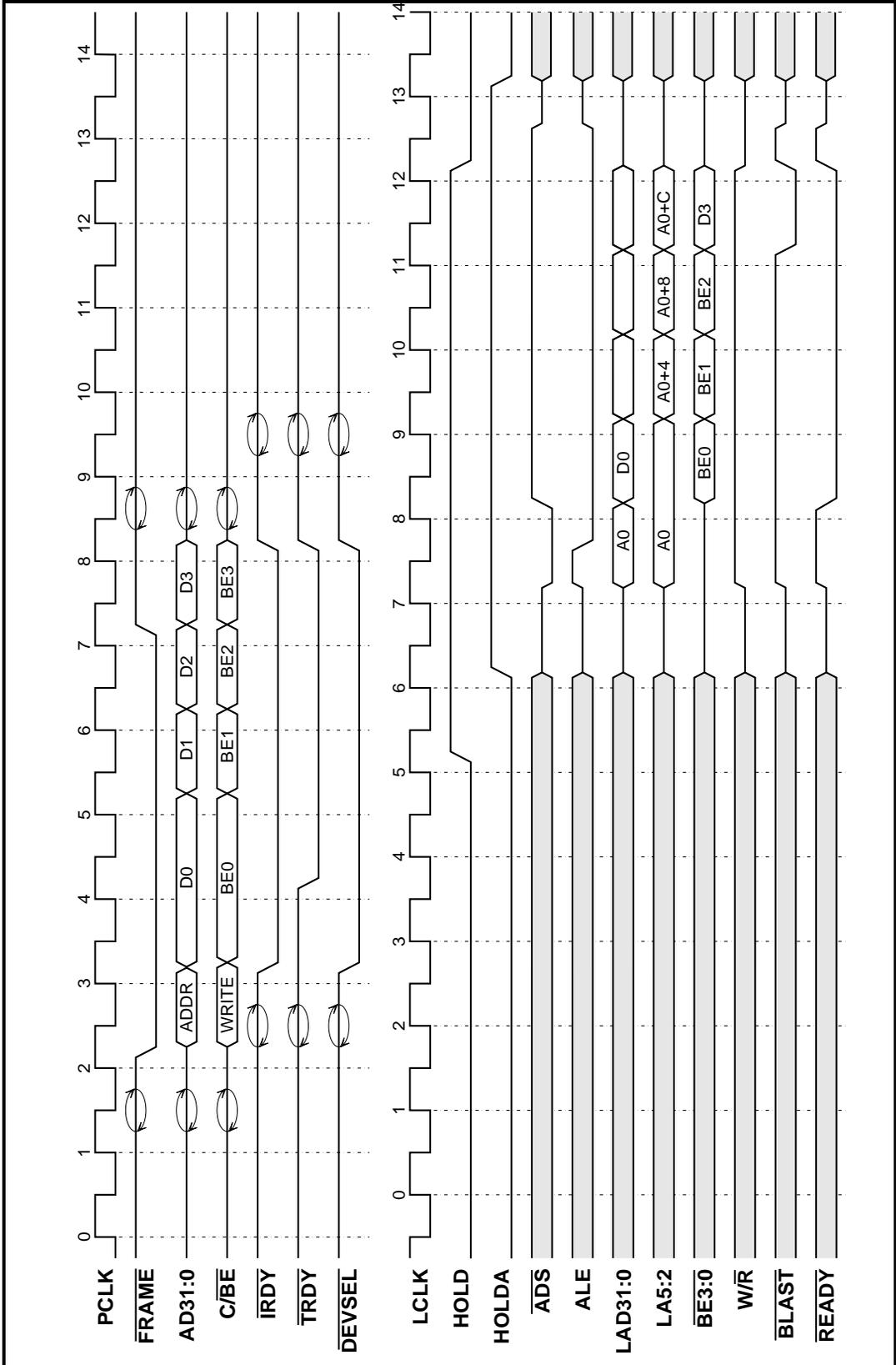


Figure 33: V350EPC (961 mode) Initiated Local Write Cycle



Local Bus Interface

Target Mode

Figure 34: V360EPC (962 mode) Initiated Local Write Cycle

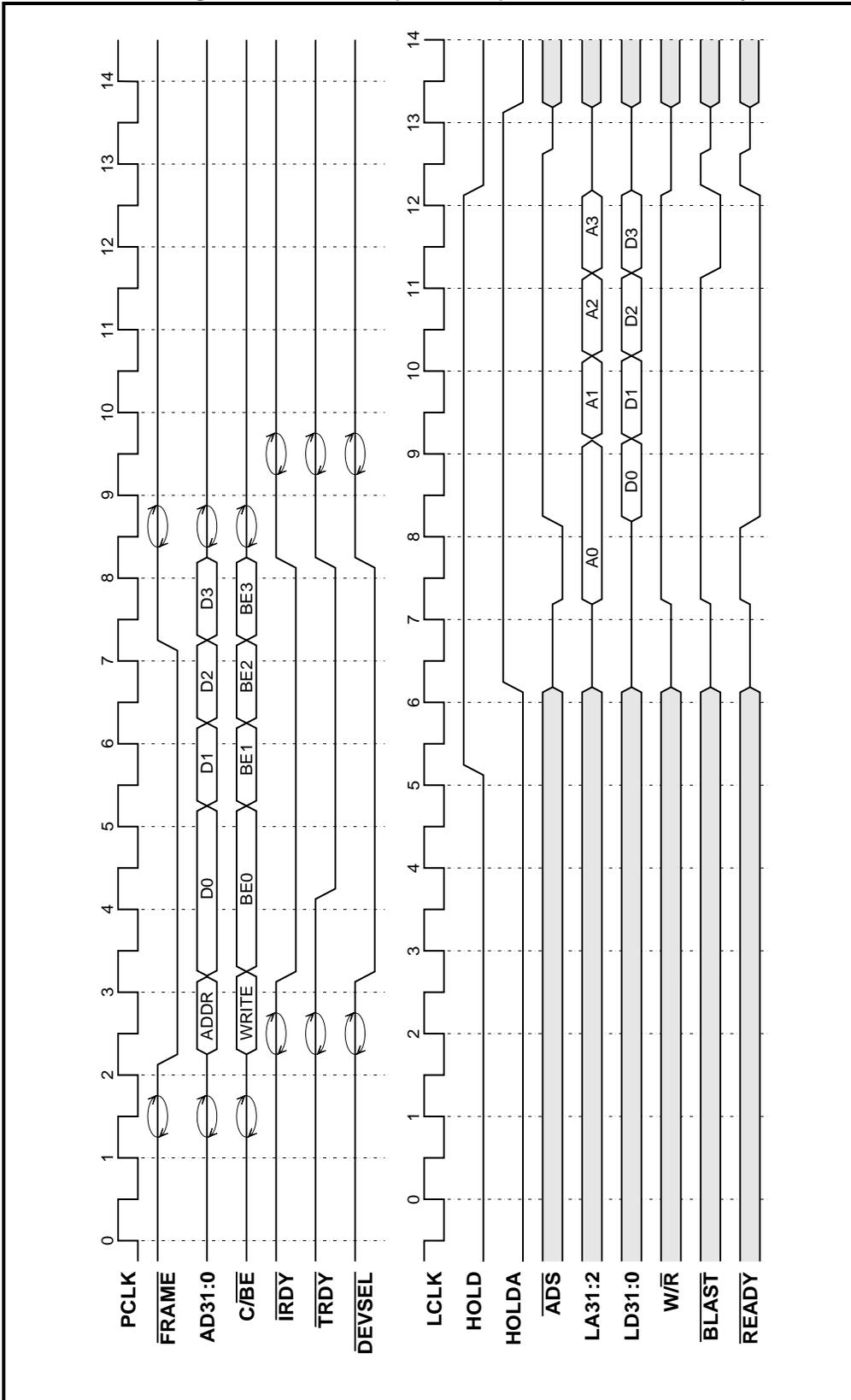
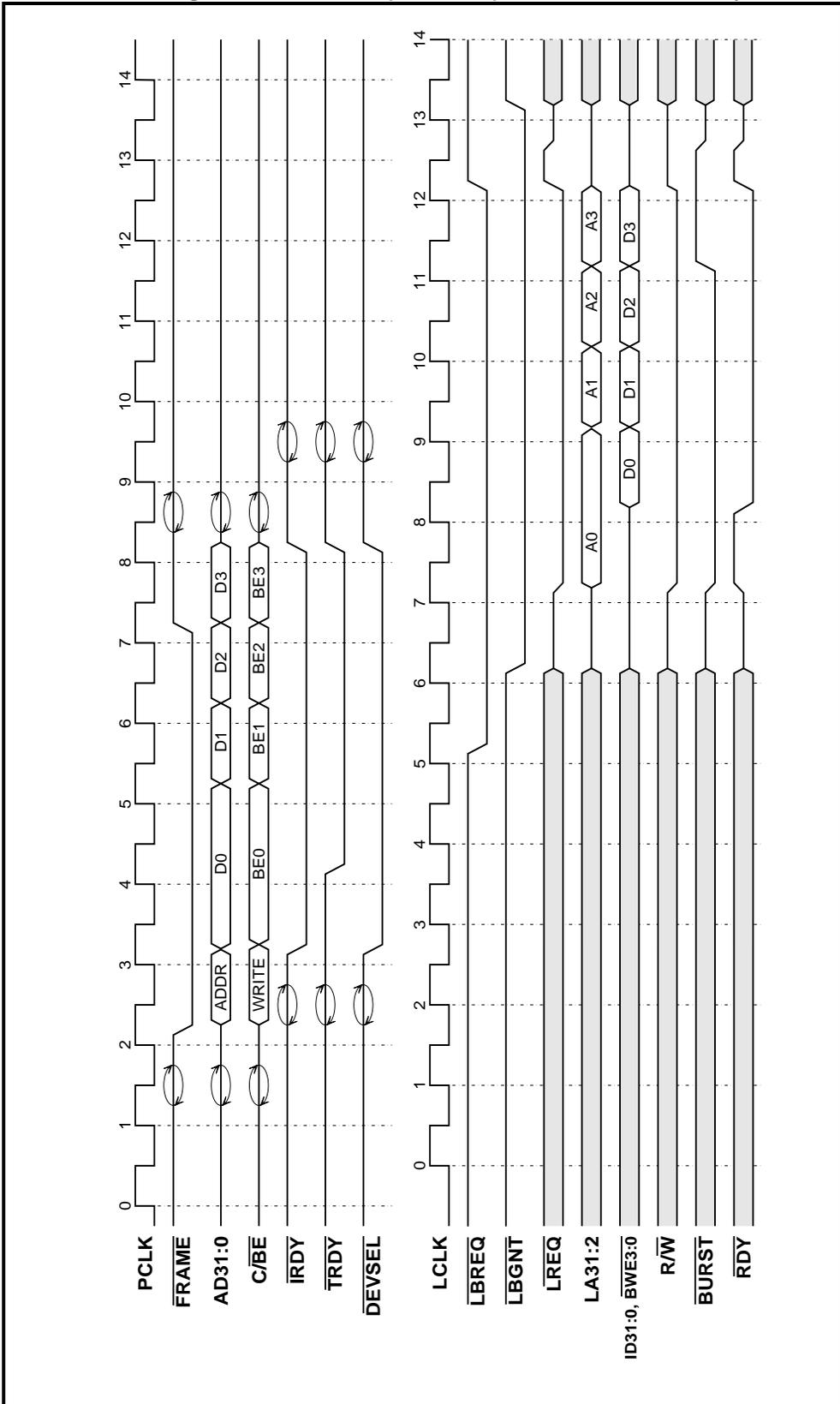


Figure 35: V360EPC (292 mode) Initiated Local Write Cycle



Local Bus Interface

Target Mode

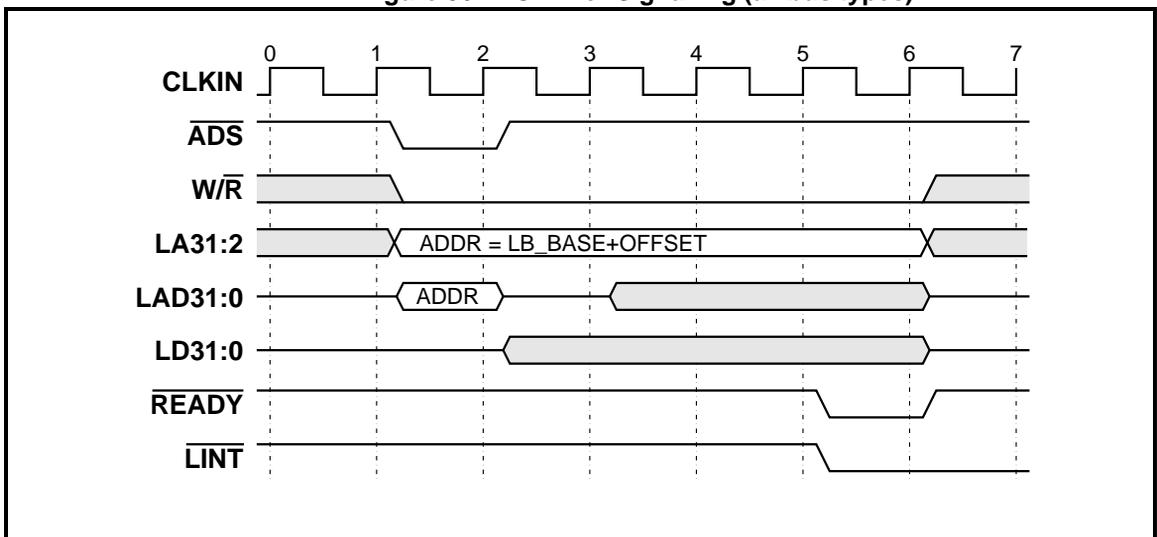
8.1.5 Target Mode PCI Error Signalling

Target mode reads and writes to/from PCI space may cause a number of PCI errors (see “PCI Interface”). The EPC reports these errors for reads by:

- Asserting $\overline{\text{RDY}}$ to unlock the local bus.
- Generating a maskable interrupt to both the local and PCI interrupt controllers.

Figure 36 shows the case of a PCI read error caused by a Master Abort (attempt to access a non-existent PCI device). Write errors are handled differently since they will complete on the PCI bus some time after the local master has posted them in the write FIFO. Write errors are reported only through the PCI error interrupt. See “PCI Bus Interface” for more details.

Figure 36: PCI Error Signalling (all bus types)



8.1.6 Deadlock Conditions and Resolution

There is a potential deadlock condition that exists whenever two or more local processor/bridge combinations are used in a system. This condition manifests itself as follows:

- One local master attempts to read the local memory of another processor on the PCI bus; simultaneously the other processor is attempting to read the first processor’s local memory.
- Per the PCI spec, the master in control of the PCI bus receives a “retry”, however, since that master is performing a read it cannot return data to its processor, and therefore maintains a NOT READY indication to that processor. This situation effectively keeps the other processor involved off the local bus.

Since neither of these reads can proceed, the PCI bus is at a *deadlock*. If the write FIFO becomes full then there is also a similar deadlock issue involving writes.

The EPC resolves this problem automatically by detecting “overdue” unresolved reads or writes on the local bus. A timer can be enabled (LB_CFG register) that will time-out on any access that takes longer than 64-1024 clocks to complete will be deemed overdue, and can be terminated by asserting RDY and/or LINT (to indicate a PCI access error). A software handler must be used to recover from the error. For processors that provide a bus backoff or retry mechanism (such as the i960Cx/Hx processors and the Am29030/40 processors) fully transparent deadlock avoidance is possible as described in section “Deadlock Avoidance using the BTERM as an Output” on page 84.

8.2 MASTER MODE

The local bus interface is said to be in *master mode* when the EPC is performing local read and write requests in response to PCI or DMA accesses.

8.2.1 Requesting the Local Bus

The local bus is requested by the EPC by asserting the HOLD(LBREQ) signal. Local bus access is granted by the local bus arbiter by returning the HOLDA(LBGRT) signal.

8.2.2 Local Bus Size

The local bus master interface on the EPC can support 8, 16 and 32 bit target devices. This is controlled through the LB_SIZE register. The local bus address space is sub-divided into 16 regions of 256MB. For each of these regions, the bus width can be determined separately. For 16 bit bus modes, there are 2 options:

- 16 bit packed: a PCI to local read/write that only involves bytes in one half of the 32 bit word (15:0 or 31:16) will result in only a single local bus transfer. The result is that a burst sequence can “skip” cycles where no data is to be transferred.
- 16 bit un-packed: a PCI to local read/write that only involves bytes in one half of the 32 bit word (15:0 or 31:16) will result in two consecutive transfers done as a burst where the BE1:0 will be de-asserted for the cycle that doesn’t involve any data transfer. Consequently, burst cycles will always be linear.

Table 11: PCI to 16 Bit Local Packed Mode Transfers (little endian)

PCI C/BE3:0	Local		PCI C/BE3:0	Local	
	1st cycle	2nd cycle		1st cycle	2nd cycle
0000	D15:0	D31:16	1000	D15:0	D23:16
0001	D15:8	D31:16	1001	D15:8	D23:16
0010	D7:0	D31:16	1010	D7:0	D23:16
0011	D31:16		1011	D23:16	
0100	D15:0	D31:16	1100	D15:0	
0101	D15:8	D31:16	1101	D15:8	

Local Bus Interface

Master Mode

Table 11: PCI to 16 Bit Local Packed Mode Transfers (little endian)

PCI		Local		PCI		Local	
C/BE3:0		1st cycle	2nd cycle	C/BE3:0		1st cycle	2nd cycle
0110		D7:0	D31:16	1110		D7:0	
0111		D31:24		1111			

Figure 37: Connection of 8 and 16 bit Peripherals to the V350EPC

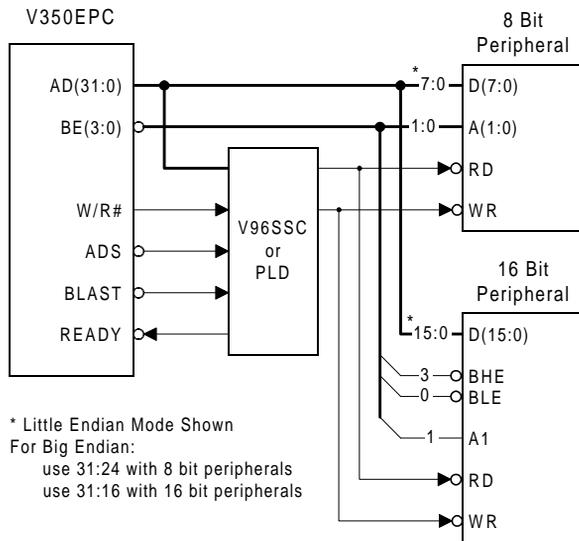
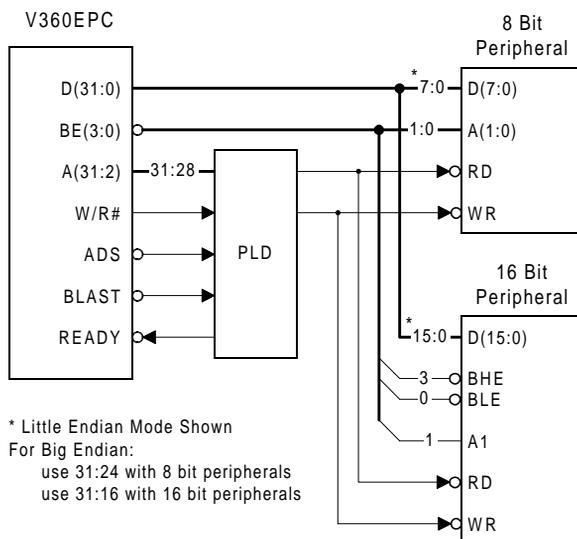


Figure 38: Connection of 8 and 16 bit Peripherals to the V360EPC



8.2.2.1 Local Bus Endian Mode

The endian mode of the local bus is set with the ENDIAN bit in the LB_CFG register. This bit controls how data is sequenced in 8 and 16 bit regions on the local bus when the EPC is a local bus master. It does NOT effect the access to internal registers or access to 32 bit local bus regions. It also has not direct relationship to swapping.

Table 12: Translation of 32 Bit PCI Data into an 8 Bit Local Bus Region

Access	Little Endian (ENDIAN=0)		Big Endian (ENDIAN=1)	
	A1:0	Data (Via D7:0)	A1:0	Data (Via D31:24)
1st	00	D7:0	11	D31:24
2nd	01	D15:8	10	D23:16
3rd	10	D23:16	01	D15:8
4th	11	D31:24	00	D7:0

Table 13: Translation of 32 Bit PCI Data into an 16 Bit Local Bus Region

Access	Little Endian (ENDIAN=0)		Big Endian (ENDIAN=1)	
	A1	Data (Via D7:0)	A1:0	Data (Via D31:24)
1st	0	D15:0	1	D31:16
2nd	1	D31:16	0	D15:0

8.2.3 Data Swapping

Data swapping options are available for each of the PCI-to-local and local-to-PCI address translation units. In the local-to-PCI direction, swapping is controlled by the LB_BASEx registers. In the other direction PCI_MAPx is used. Each DMA channel also has individual control over this option. These options are used to translate between big endian local bus processors and little endian PCI space. Unfortunately, there is no way to provide a universal endian converter because processors that cache data can burst data of differing size. However, if 8/16 bit loads/stores are always done as 8/16 bit operations, then the "Auto Swap®" mode can be used. It works by examining the size of the transfer based on how many byte lanes are enabled. When $\overline{BE}[3:0] = "1100"$ or $"0011"$ then a 16 bit swap is done. When $\overline{BE}[3:0] = "1110"$, $"1101"$, $"1011"$ or $"0111"$ then an 8-bit swap is done. Any other combination results in non-swapped data.

Table 14: Swap Mux Options

Description	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]
Auto Swap	11	Auto Swap (not available for DMA transfers)			

Local Bus Interface

Master Mode

8.2.4 i960 Local Bus Reads and Writes

The local bus protocol used by the EPC family components is identical to that of the target processor: V960EPC duplicates the i960Sx protocol, the V961EPC duplicates the i960Jx protocol, and the V962EPC duplicates the i960Cx/Hx protocol.

All EPC's support bursts longer in length than the i960 processor limit of 4 words (8 transaction on the i960Sx devices). The maximum burst length supported is 256 words. The end of the burst is indicated by the $\overline{\text{BLAST}}$ signal, as is the case with "standard" i960 buses. The burst length on the local bus is programmable via the LBRST_MAX field in the FIFO_CFG register. Bursts may also be terminated at any time by asserting the $\overline{\text{BTERM}}$ signal. The burst will re-start with another $\overline{\text{ADS}}$ assertion at the point it was terminated.

V3 has chosen not to duplicate the wait-state control logic found in the i960Cx (MCON registers) and i960Hx (PMCON registers).¹

8.2.5 Am29K Local Bus Reads and Writes

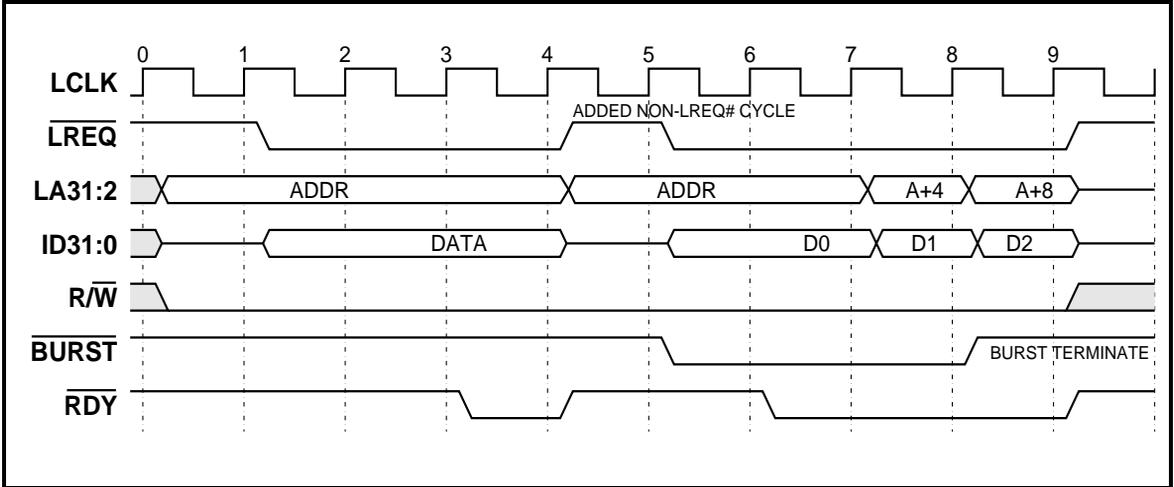
The V292EPC provides two different local bus protocols for Am29030/40 style busses: high-performance mode and strict compatibility mode. The two modes differ in the BURST signal assertion timing and in the length of bursts supported.

8.2.5.1 Strict Compatibility Mode

In *strict compatibility* mode the $\overline{\text{BURST}}$ signal is asserted coincident with the assertion of $\overline{\text{LREQ}}$ (see Figure 39). This mode duplicates the bus protocol shown in the Am29030/40 documentation. When a series of access cycles are done by the V360EPC (292 mode) on the local bus (such as a series of burst writes) the $\overline{\text{LREQ}}$ signal will be de-asserted for one cycle between bursts. This is done so that the $\overline{\text{BURST}}$ and $\overline{\text{LREQ}}$ signals can occur simultaneously at the beginning of a burst or non-burst cycle.

1. This logic is of no use in systems using DRAM as main memory, as the wait-state profile for DRAMs is indeterminate (due to refresh cycles.)

Figure 39: V360EPC (292 mode) Local Bus Master Access (Strict Compatibility Mode)

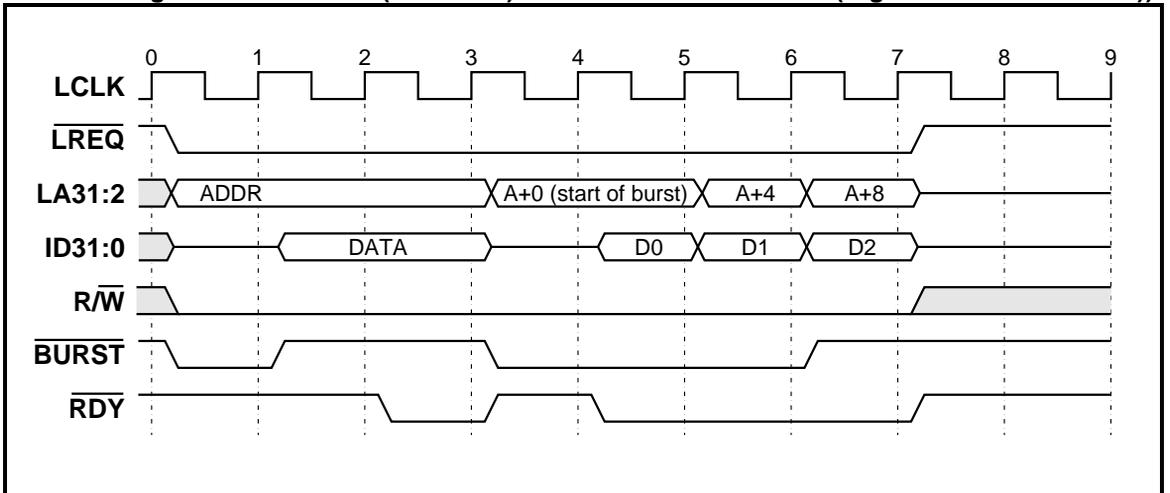


8.2.5.2 High-Performance Mode

In systems that choose to use high-performance mode, both $\overline{\text{BURST}}$ and $\overline{\text{LREQ}}$ are asserted as soon as the V360EPC (292 mode) begins a cycle (LCLK 1 and LCLK 3 in Figure 40). If the cycle is a non-burst cycle then $\overline{\text{BURST}}$ is de-asserted on the next cycle (LCLK2). If a burst is really going to happen then $\overline{\text{BURST}}$ remains asserted until before the last data transfer of the burst (LCLK 6) as it normally would. If the local bus slave devices don't care about the state of $\overline{\text{BURST}}$ on the first cycle then the high performance mode will save one cycle and should be used by setting the FAST_REQ bit in the SYSTEM register.

The V360EPC (292 mode) defaults to strict compatibility mode following reset. The local bus mode is changed by writing to the FAST_REQ bit in the SYSTEM register. System using the V292BMC/CMC Burst DRAM Controller can take advantage of the high performance mode and will also work when strict compatibility mode is selected.

Figure 40: V360EPC (292 mode) Local Bus Master Access (High-Performance Mode)



8.3 BURST SUPPORT

The EPC is designed to accommodate bursting of up to 1K bytes on both the PCI and local buses. Long bursting is accomplished by programming the PBRST_MAX and LBRST_MAX bits in the FIFO_CFG register to select the 1K byte (256 word) burst size. Simply selecting a 1K burst size will not guarantee a full 1K uninterrupted burst - other parameters must be considered that will limit the burst size:

- For aperture writes (local bus master writes PCI bus *or* PCI bus master writes local bus) the burst length on the source bus will determine the length on the destination bus if it is shorter than the value of the corresponding PBRST_MAX or LBRST_MAX.
- For the case of local to PCI access: burst length will be limited by PCI target devices by asserting STOP (bus retry) at its burst limit.
- In the case of read prefetch, bursts will terminate if the prefetch FIFO becomes full.
- Burst length on the local bus will be limited by the memory systems ability to support long bursts and not by the burst size of the processor on the local bus. Controllers such as the V96SSC, V96BMC, V96CMC, V292BMC and V292CMC will all support 1K bursts. Attempting to do long bursts to memory systems incapable of supporting them will result in lost or corrupt data.
- When using the V96BMC Rev D with the V350EPC or V360EPC please note that the BMC Rev D device strictly follows the i960 bus protocol which does not allow byte enables to change on a burst. However, the EPC has been allowed to be more flexible by allowing the byte enables to change in a burst. Therefore, you need to take this fact in consideration when you are using the V96BMC Rev D with the EPC.

```
-----  
-----  
-- V3 Modification (VHDL CODE) for a Small PLD:  
-- Explanation:  BTERM# must be driven only when a write  
--               cycle is being initiated by the EPC when not all  
--               the byte enables are driven.  
--  
--               This fix creates a better cohesion between the BMC  
--               and the EPC since the BMC adheres strictly  
--               to the i960 protocol and the EPC has more flexibility  
--               to allow byte transfers to occur. Please note that  
--               this fix is not required when an SSC is used as the  
--               memory controller. (This occurred because the  
--               BMC was designed before the EPC was designed,  
--               and the SSC came after the EPC).  
--  
--  
  
process  
begin  
wait until(clkin'event and (clkin = '1'));
```

```
-- identify the troublesome cycle by EPC
fix_qads <= (not l_ads_n) and pci_hlda and l_wr_n;

end process;

-- disconnect (block) the troublesome cycle from EPC
l_bterm_n = '0' when( not((l_be_n3 & l_be_n(1 downto 0))
    = "000") and fix_qads='1') else
    '1' when(pci_hlda = '1') else 'Z';

-----
-----
```

Bursts by the EPC are always terminated at a burst modulo page boundary. For example: if a 1K burst length is selected then the EPC will never cross a 1K page boundary (it will never burst from address XXXXX3FCH to XXXXX400H). Similarly, if a 16 byte (4 word) burst is selected it will never burst from XXXXX0CH to XXXXX10H. This makes the EPC comparable to processors and page mode memory devices.

8.4 **BTERM OPERATION (961 AND 962 MODE ONLY)**

This section is applicable only to the EPC in 961 and 962 modes.

The $\overline{\text{BTERM}}$ signal is used in two ways. When the EPC is a local bus master, the $\overline{\text{BTERM}}$ input acts as a "Burst Termination" input with the same timing as the $\overline{\text{READY}}$ input. When the EPC is a local bus slave for local aperture to PCI read/write operations, the $\overline{\text{BTERM}}$ signal can be programmed to drive as an output for time-out conditions.

Note that $\overline{\text{BTERM}}$ must have a pull up resistor on the pin even if it is not being used in as an input or output as described below.

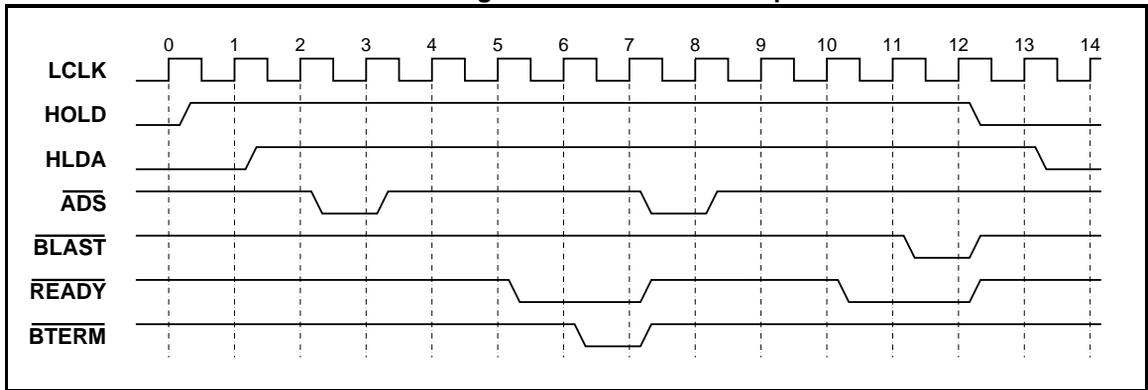
8.4.1 **$\overline{\text{BTERM}}$ as an Input**

The EPC will respond to $\overline{\text{BTERM}}$ together with $\overline{\text{READY}}$ as if $\overline{\text{BLAST}}$ had been asserted also with $\overline{\text{READY}}$. This will cause the $\overline{\text{ADS}}$ signal to be re-driven to start a new cycle where the original burst left off. The sequence is depicted below: It begins as a normal EPC local bus master cycle with $\overline{\text{HOLD}}/\overline{\text{HLDA}}/\overline{\text{ADS}}$ being driven in sequence. At LCLK 6 the first data is $\overline{\text{READY}}$ and the cycle looks like a normal burst. However, at LCLK 7 $\overline{\text{BTERM}}$ is asserted with $\overline{\text{READY}}$ and the cycle terminates as if $\overline{\text{BLAST}}$ had been asserted at that time. At LCLK 7 the EPC begins to drive $\overline{\text{ADS}}$ again and with the address for data item 3 left over from the previous unfinished burst. This burst cycle terminates normally with $\overline{\text{BLAST}}$ and $\overline{\text{READY}}$ at LCLK 12.

Local Bus Interface

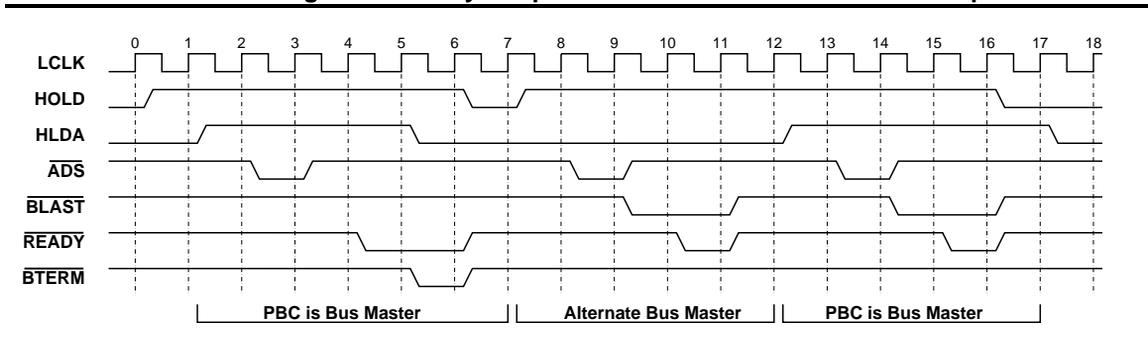
BTERM Operation (961 and 962 mode Only)

Figure 41: \overline{BTERM} as in Input



Additionally, the assertion of \overline{BTERM} can be used in conjunction with the de-assertion of HLDA to give another local bus master ownership of the bus so that the EPC won't "hog" the bus when long bursts are performed. When a burst is terminated (with or without \overline{BTERM} asserted) and more data transfer is pending, the EPC will drop its' HOLD request for one cycle at the end of the burst if the HLDA signal is taken away before the end of the cycle thus allowing another higher priority master to gain control of the bus. This is NOT the way that i960 processor usually drives its' HLDA output. Thus HLDA back to the EPC must be driven by a gated version of HLDA from the processor - typically from an arbiter device. The following diagram (Figure 42) illustrates the sequence.

Figure 42: Early Suspension of EPC Local Bus Ownership



In summary, the \overline{BTERM} used as an input will terminate bursts similar to a PCI burst disconnect. When the HLDA input is used with \overline{BTERM} , a high priority local bus device can gain access to the local bus by kicking off the EPC early.

8.4.2 Deadlock Avoidance using the \overline{BTERM} as an Output

The \overline{BTERM} signal can be used to indicate a time-out condition when a local bus master is trying to access the PCI bus through one of the EPC apertures. This operation is enabled through the LB_CFG register bit 9. When enabled, a time-out condition (as defined in the description given in the LB_CFG register) will cause the normally high-Z \overline{BTERM} signal to be driven low for one LCLK and then high for one clock before being returned to its' high-Z state. A time-out condition is usually caused by a deadlock situation, where the local processor is attempting to access resources on another similar subsystem while the local processor of that other subsystem is attempting to access resources on the local bus of the

first processor. On processors (such as the i960CA) that have a $\overline{\text{BOFF}}$ input, it is possible to break the deadlock by generating a $\overline{\text{BOFF}}$ from the $\overline{\text{BTERM}}$ output of the EPC. This is accomplished with a simple modification to the local bus arbiter device (typically a Programmable Logic Device). An example implementation is given below.

Figure 43: Circuit for Generation of Processor $\overline{\text{BOFF}}$ from V962EPC $\overline{\text{BTERM}}$ Output

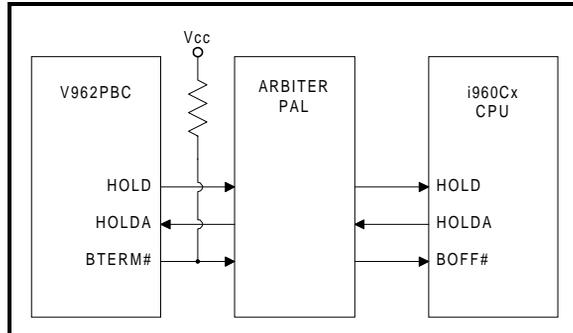
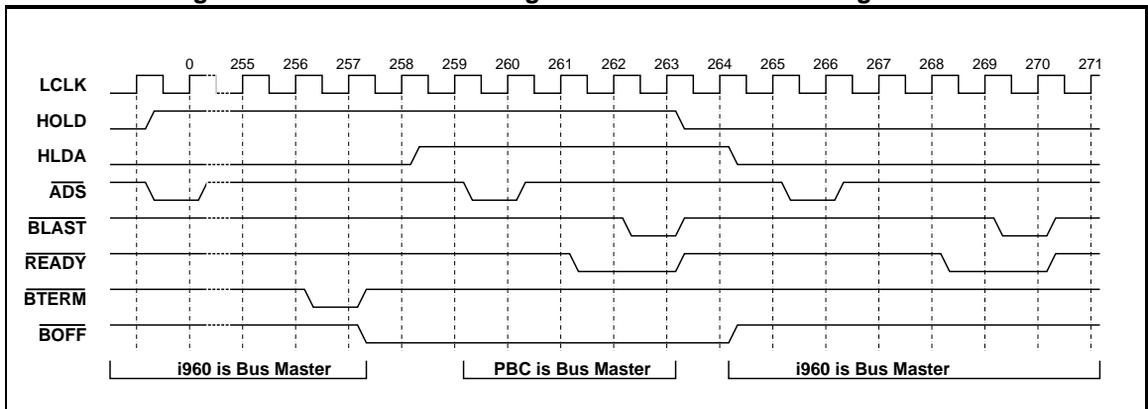


Figure 44: Waveform Showing Deadlock Avoidance Using $\overline{\text{BTERM}}$ and $\overline{\text{BOFF}}$



$\overline{\text{BTERM}}$ at LCLK 257 is detected by the Arbiter and used to generate $\overline{\text{BOFF}}$. When $\overline{\text{BOFF}}$ is asserted to the processor, it will cause it to float most of its signals as if it had lost mastership of the bus. Internally the processor is held as if in wait state waiting to see a $\overline{\text{READY}}$ assertion. This gives the EPC a chance to take mastership of the bus when it receives HLDA (hold acknowledge). With its' data transfer complete at LCLK 263, the EPC de-asserts its hold request. $\overline{\text{BOFF}}$ is then de-asserted to the processor causing it to reassert its' $\overline{\text{ADS}}$ strobe. This restarts the previously timed out access via the EPC to PCI address space. With the deadlock broken, this access now completes normally. This provides an automatic hardware solution to the deadlock problem: no deadlock handling software is required.

8.5 LOCAL BUS PARITY

The EPC is capable of generating and checking data parity on the local bus. When acting as a local bus master, the EPC generates one bit of parity information for each byte of data during write cycles. During read cycles by the EPC or any other local bus master, the EPC can perform parity checking.

Local Bus Interface

Local Bus Parity

Parity generation and checking is intended for operation with an external memory system or peripheral device. It is commonly used with DRAM arrays where single bit errors can occur due to the nature of these devices. Consequently, valid parity is not asserted when the EPC is a slave device on the local bus. In this case the value of the parity signals should be treated as "unknown".

8.5.1 Relationship between Local Parity and PCI Parity

There is no direct relationship between PCI bus parity and local bus parity since they both operate in a different manner and for a different purpose. PCI parity is provided as a means to ensure the integrity of point-to-point, master-to-slave connections. Local bus parity, on the other hand, is designed to check the integrity of a local bus memory system typically implemented with DRAM. This requires that the local bus parity information travel on the bus with the same timing as the data. This is not the case with PCI parity which lags the data by one clock cycle.

Table 15: Comparison of PCI and Local Parity

	PCI Parity	Local Parity
Number of parity signals	1	4
Parity calculation	1 bit for: AD[31:0], C/BE[3:0]	1 parity bit per data byte
Parity generation	generated by receiver of data	generated by bus master
Data to parity relationship	parity lags data one clock	parity and data together
Data to parity error relationship	error lags data two clocks	error lags data one clock

8.5.2 Local Bus Parity Generation

Valid byte parity is driven out on to the LPARx pins whenever a master write cycle is performed and with the same timing as write data is driven onto the data bus (or LAD bus in the case of the V350EPC). This allows the local bus memory system to store the parity information as if it is extra data. This differs from PCI parity in that PCI parity lags the data by one clock cycle making it more difficult to store along with the data in a memory array (this is not the intention of PCI parity).

The EPC generates parity ONLY for itself when acting as a local master (PCI-to-local aperture writes and PCI-to-local DMA transfers). Other masters (such as the CPU) must generate their own parity when they perform write cycles. Parity is also NOT generated by the EPC when it is a local bus slave (either for access to the internal registers or for Local-to-PCI aperture access).

The four parity bits are generated according to Table 16:

Table 16: Relationship between Parity Output Signals and Output Data

LPAR3	$\text{xor}(\text{LD}[31:24], \text{POE}^{\text{a}})$
LPAR2	$\text{xor}(\text{LD}[23:16], \text{POE})$
LPAR1	$\text{xor}(\text{LD}[15:8], \text{POE})$
LPAR0	$\text{xor}(\text{LD}[7:0], \text{POE})$

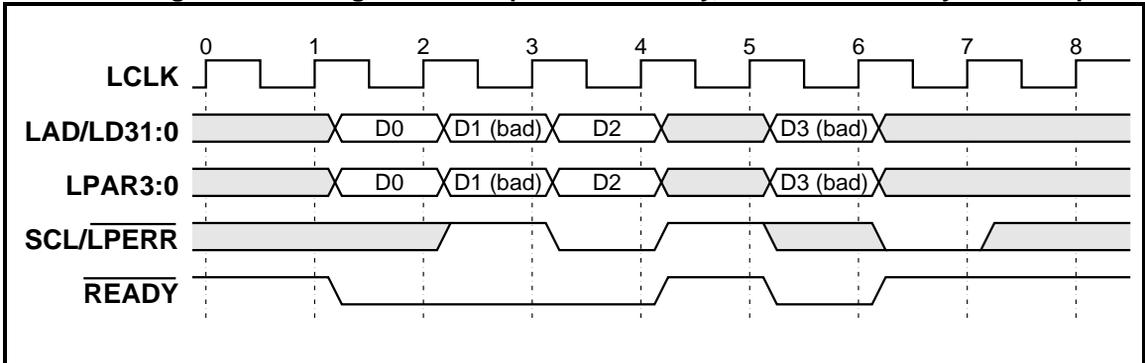
a. POE bit of the SYSTEM register (Parity Odd/Even)

8.5.3 Local Bus Parity Checking

Each LCLK cycle, the EPC checks parity on the local bus and drives the $\overline{\text{SCL/LPERR}}$ signal accordingly on the next cycle. Consequently, the $\overline{\text{SCL/LPERR}}$ signal must be qualified to allow only valid regions of address space to be checked. Figure 45 shows the relationship between data, parity and the parity error output. Note that the parity error output $\overline{\text{LPERR}}$ lags the parity and data by one clock.

The local bus parity feature is intended for use only with local master cycles. That is, LPAR[3:0] are driven with valid parity information when the EPC is performing a write cycle as the local bus master. While parity is always being checked by the internal circuitry, the only relevant cycles where parity checking is valid are EPC local bus master read cycles where the slave device is something like 36 bit DRAM. Parity is not generated for access to the EPC by an external master.

Figure 45: Timing Relationship Between Parity, Data and the Parity Error Output

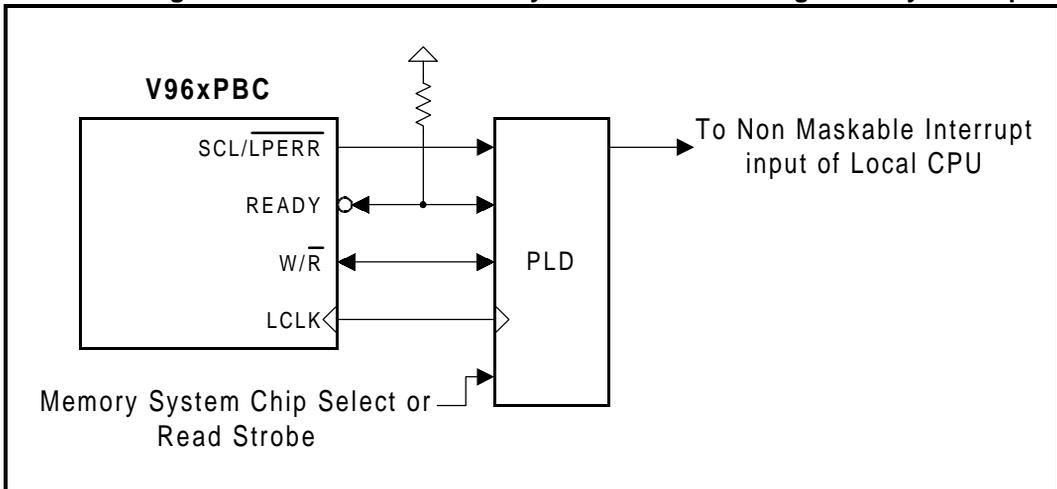


The circuit in Figure 46 is an example of how the $\overline{\text{LPERR}}$ signal should be qualified to generate a high priority interrupt to signal a parity error to the local processor.

Local Bus Interface

Local Bus Parity

Figure 46: Qualification of Parity Error to Generate High Priority Interrupt



PLD equations

```
qual.d = ready & !wnr & mem_cs; /* intermediate qualifier signal */  
nmi.d = lperr & qual /* Non Maskable Interrupt to CPU
```

Chapter 9

PCI Configuration

The EPC may be used as both a host or target bridge device. As such, the EPC can both generate configuration cycles and respond to them. This chapter describes both types of PCI configuration.

9.1 CONFIGURATION AS A SYSTEM HOST BRIDGE

The EPC acts as a host bridge when it is used to configure other PCI devices in the system. For example, a laser printer that uses an i960[®] as the main CPU and uses PCI as the mezzanine bus, would run configuration cycles to initialize PCI peripherals such as ethernet chips and SCSI controllers.

9.1.1 *EPC Host Configuration Mechanism*

PCI configuration cycles consist of setting a specified target's IDSEL line active, then performing reads and writes to the configuration space of the selected peripheral. The IDSEL line is deasserted after accesses to the target peripheral's configuration space are complete.

9.1.2 *Controlling Target IDSEL Lines*

The EPC does not provide a direct control of the state of individual targets' IDSEL lines. The system hardware must provide a mechanism for activating the IDSEL line for each target (if the EPC is to be used as a host bridge). IDSEL is not like other PCI signals in that it need not be synchronous with each configuration cycle. IDSEL must be active during the address phase to select the EPC.

An external register is the simplest method for controlling IDSEL. V3's Am29K[™] PCI motherboard (the Lion-29K), for example, uses a PAL device that sets and clears specific IDSEL lines based on accesses to specific memory locations in local space.

PCI Configuration

Configuration as a System Host Bridge

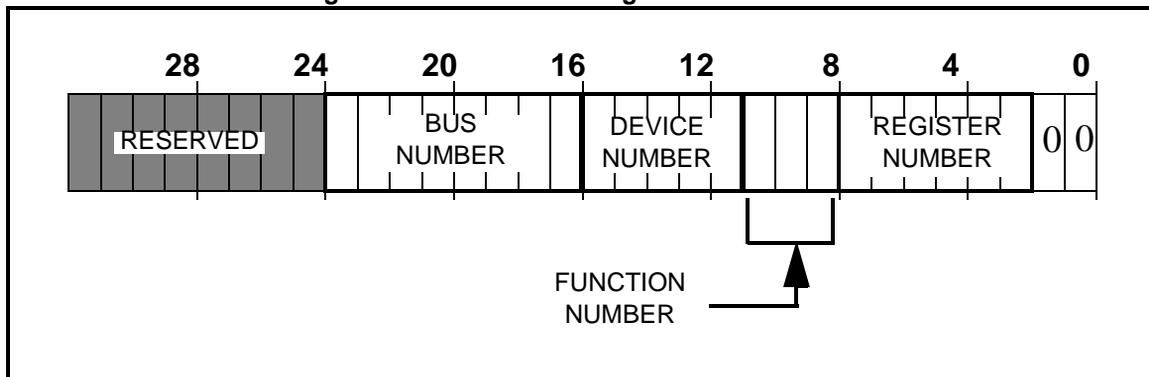
9.1.3 Generating Configuration Reads and Writes

The address placed on the bus during configuration cycles is actually encoded information as shown in Figure 47. The EPC does not automatically generate this encoding, it is completely under software control. The configuration address encoding is programmed as follows:

- Setup one of the Local-to-PCI apertures as 16 megabytes in length. This forces the address translation logic to pass the lower 24 bits of the address through to the PCI bus *without* translation. The lower 24 bits of the address are the encoded configuration information. The base address in Local memory for this aperture may be placed on any 16Mbyte boundary.
- Setup the address translation for the above aperture to translate accesses to 0000.0000H. Since the aperture is setup for 16Mbytes, this has the effect of setting A[31:24] to zeros. Set the TYPE field for the Local-to-PCI aperture to “Configuration Reads/Writes”.
- Create a 24-bit value that encodes the configuration address information. Add the base address of the Local-to-PCI aperture to this encoded information.
- Perform a read (or write) to the above address in local space. This will be translated to a configuration read (or write) in PCI space.

In the above example, it is assumed that the IDSEL lines are set appropriately by the user’s system hardware and software.

Figure 47: Encoded Configuration Address Information



9.1.4 Using Configuration Information

The following information is usually determined during configuration:

- What PCI devices are present in the system and what type of devices they are
- What resources are necessary for each device in terms of memory, I/O, and interrupts
- What capabilities each of the devices has (i.e. Fast Back-to-Back capable)

The PCI Specification describes in detail the information available during PCI configuration and how to retrieve that information. The usage of that information is highly application dependent and is beyond the scope of this manual.

9.1.5 **Determining the Presence of Target Devices During Configuration**

A Master Abort will occur if the system host attempts to access the configuration space of a device that is not present. This occurs, for example, when the IDSEL line for an empty PCI slot is activated and a configuration cycle is run. When this occurs, the EPC sets the Master Abort bit in the PCI_STATUS register and returns FFFF.FFFFH to the host. This combination of events is used to determine the presence of PCI devices in a system following a reset.

Typically, the following mechanism is used during the boot code:

- Each IDSEL line is activated in turn and a configuration read is attempted from the Vendor ID register
- If a vendor ID of FFFFH is detected it means that the current IDSEL line is not connected to a valid device (FFFFH is a reserved Vendor ID for just this purpose)
- If a valid vendor ID (i.e. not FFFFH) is detected, further configuration of the device is performed

Most systems will elect to disable PCI error interrupts during configuration.

9.2 CONFIGURATION AS A TARGET BRIDGE

The EPC responds to type 0 configuration cycles when acting as a target bridge. Type 1 (PCI-to-PCI bridge) configuration cycles are ignored.

The following information is usually retrieved from a target device during configuration:

- The number and size of I/O and memory address regions required (read from the PCI base registers)
- The vendor ID, device ID, and device type information from the PCI header
- Any supplemental information provided for in the PCI header

9.2.1 **EPC Base Register Response to Configuration Inquiries**

Most information is read by the system host directly from the EPC's configuration space registers. For example, Vendor ID is retrieved simply by reading the Vendor ID register.

PCI Configuration

Configuration as a Target Bridge

The following are exceptions to the above:

- Size requested for PCI-to-Local data transfer apertures (PCI_BASEx registers).
- Size requested for expansion ROM transfer apertures (PCI_ROM register).

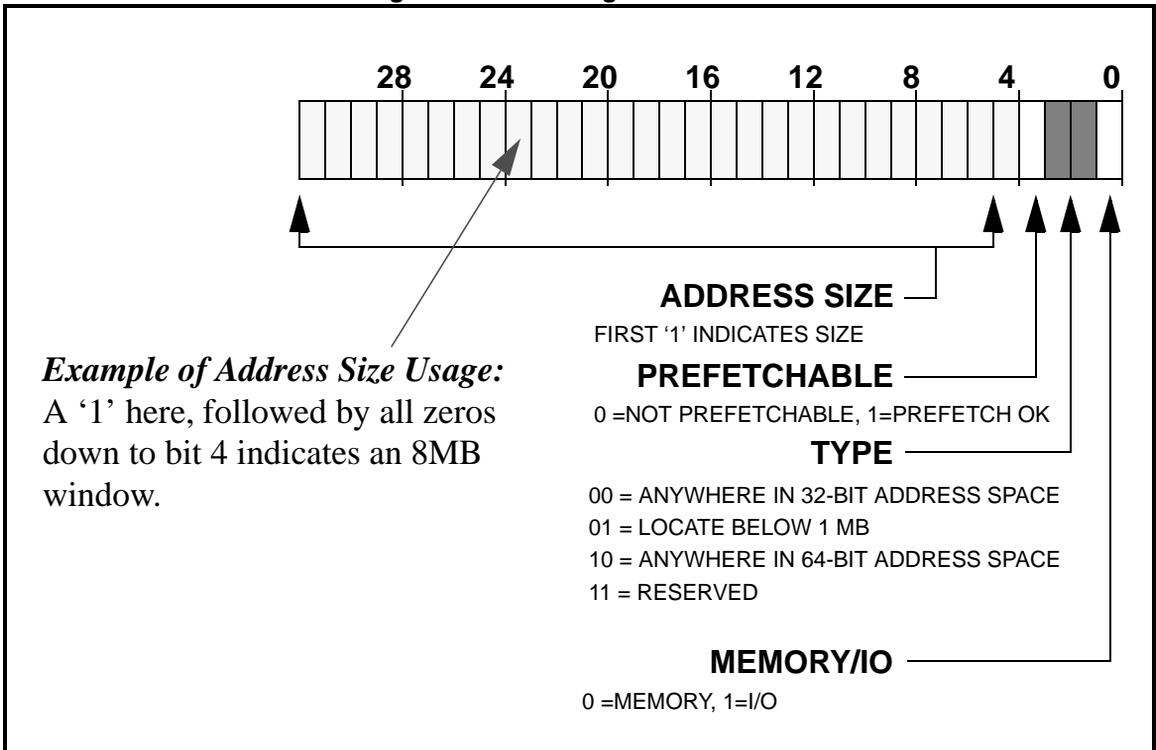
In both of these cases the PCI Specification outlines a two step interrogation process. First, each base address register is written with FFFF.FFFFH. Then the base register is read back. The value read back indicates the size and capabilities of the aperture (see Figure 48).

As an example, let's assume that PCI-to-Local bus aperture 0 is programmed for 4 megabytes in memory space with prefetching enabled. When the system host interrogates this register it will read back FFC0.0004H which is interpreted as:

- The aperture is four megabytes in size, since the first "1" seen in the address field (scanning up from bit 4) is at the 4M level
- The aperture can be located anywhere in the 32-bit address space on a 4MB boundary
- The aperture is memory mapped
- The aperture is "prefetchable"

The EPC sets the size information according to the size of the aperture specified in the PCI_MAPx register. Unused base registers in the EPC return all zeros.

Figure 48: Base Register Return Information



9.2.2 EPC Expansion ROM Base Register Response to Configuration Inquiries

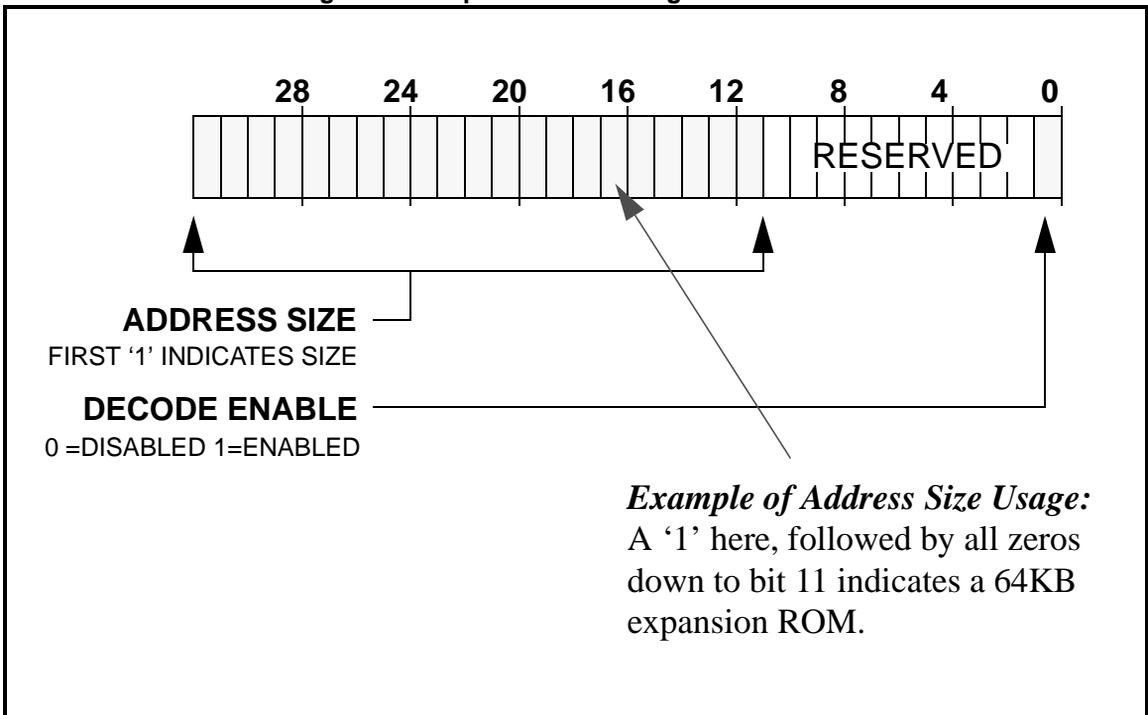
The expansion ROM base register is interrogated using the same method outlined above:

- All ones are written to the register (with the exception of the enable bit).
- The requested size is read back from the same register.

The format for the expansion ROM base return information is shown in Figure 49. The size information is determined from the expansion ROM size field set in the EPROM_MAP register.

Additional inquiries may be necessary depending on the host platform architecture. PCI based PCs, for example, will map the expansion ROM into memory and then scan for the ROM present signature of 55AAH. Please refer to the PCI BIOS spec for your particular target architecture for more information.

Figure 49: Expansion ROM Register Return Information



PCI Configuration

Configuration as a Target Bridge

Chapter 10

PC Compatibility

The EPC includes several features to improve compatibility when used as a target bridge in real mode PC/DOS systems. Real mode DOS systems have several limitations:

- Memory accesses are limited to below 1 megabyte. Add-in cards designed for real mode DOS typically use small blocks of memory located between A0000H and F0000H
- I/O devices use small blocks of I/O space, often with pre-defined addresses

The EPC provides a real mode DOS aperture that can be used to emulate DOS memory and I/O apertures for backward compatibility. It is important to note that these features fall outside of the PCI specification and can be disabled for strict compliance.

10.1 REAL MODE DOS COMPATIBILITY APERTURE

When the ADR_SIZE field of the PCI_MAP1 register (not available for Aperture 0) is configured for DOS compatibility mode then the PCI_BASE1 register is interpreted differently than the standard PCI definition. The need for this is a consequence of the fact that the original ISA bus based PC only decoded A9 down to A0 for I/O space address mapping. DOS mode addressing allows up to 3 holes in the PCI address space to be decoded that are useful for emulating DOS peripherals. There are 2 options depending on how the IO bit is set in the corresponding PCI_BASE register:

- IO = '0' allows for up to 2 decoders for PCI I/O cycles in addition to a single memory decoder for PCI memory cycles in the region between 512KB and 1MB
- IO = '1' allows for up to 3 decoders for PCI I/O cycles

Whenever a PCI cycle is detected by the DOS mode decoder (I/O or memory) then a corresponding bus cycle will be seen in a 1MB aperture on the local bus. The base address of that aperture is determined by the MAP_ADR field of the PCI_MAPx register. Within the local bus aperture, the first 64K will typically be used as the local bus remapping of PCI I/O cycles. For I/O cycles all relevant I/O addresses (up to A15) are mirrored on the local bus so that address aliasing information is carried across as DOS mode peripherals expect.

DOS memory region decode doesn't remap the address space within the 1MB aperture. Therefore, if a region is decoded starting at PCI address 0xA0000 then it will appear at

PC Compatibility

Real Mode DOS Compatibility Aperture

0xA0000 + the 1MB region selected by MAP_ADR in the PCI_MAP register.

The operation of DOS mode decode can be more easily understood with the following diagrams.

Figure 50: DOS Compatibility Mode Memory Decoding and Address Translation

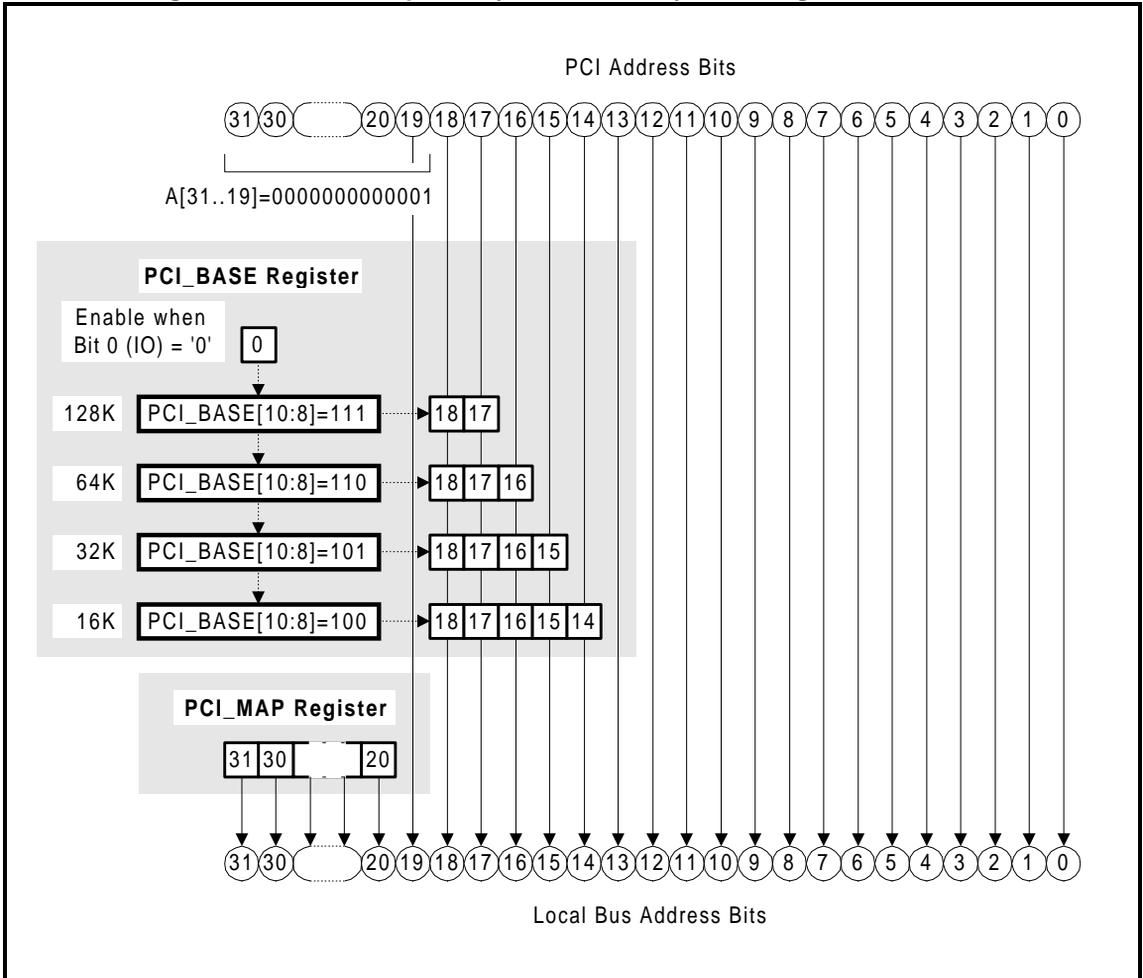
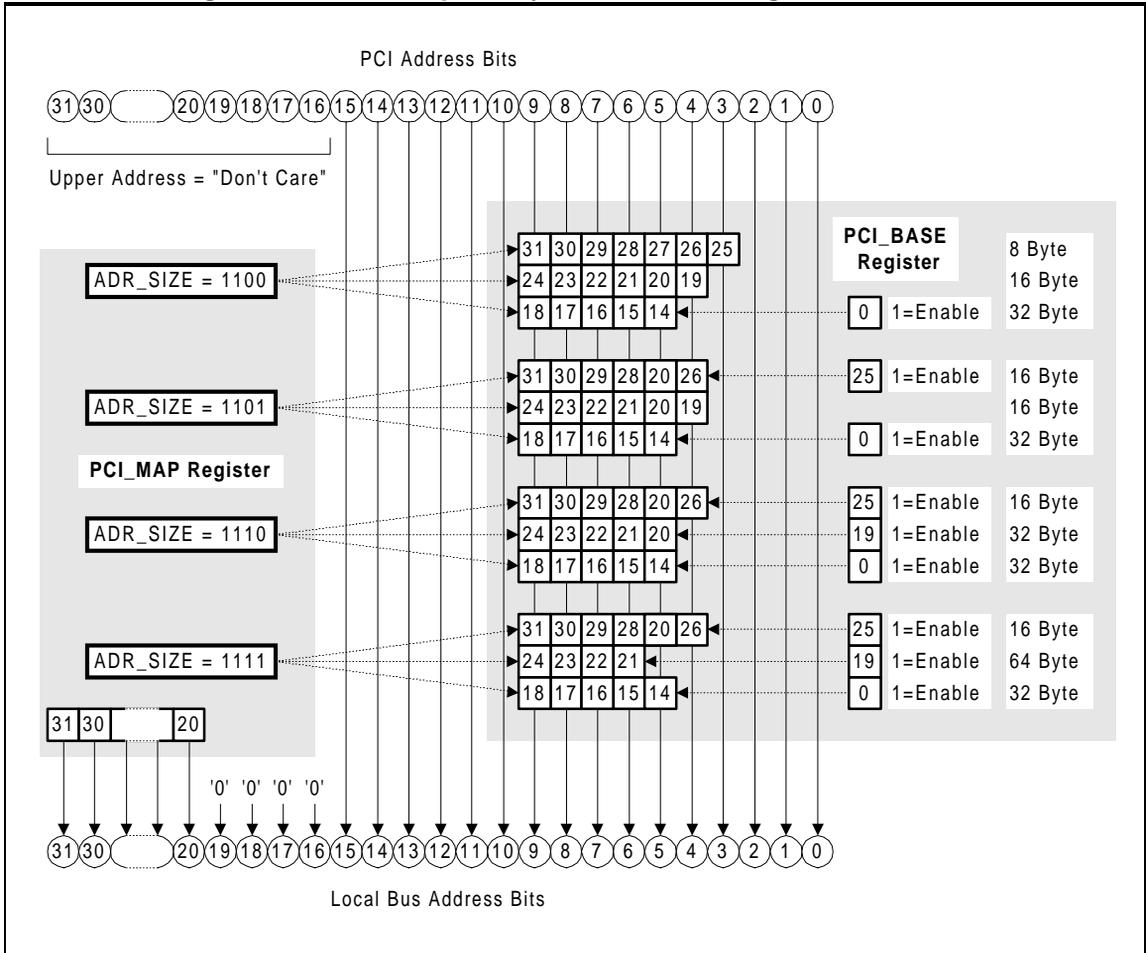


Figure 51: DOS Compatibility Mode I/O Decoding and Address Translation



10.2 EXAMPLE: VGA PERIPHERAL

Standard VGA addressing requires 3 address decodes:

- VGA Memory accesses - 0xA0000 - 0xBFFFF (128K bytes)
- VGA I/O accesses - AD[9:0] = 0x3B0 - 0x3BF (16 bytes) in addition to all ISA aliases where AD[15:10] are not decoded
- VGA I/O accesses - AD[9:0] = 0x3C0 - 0x3DF (32 bytes) in addition to all ISA aliases where AD[15:10] are not decoded

Such an address map can be decoded through a single aperture by programming one of the sets of PCI_BASE and PCI_MAP registers in the following way:

- PCI_MAP: ADR_SIZE = 1110 to select a 16 and 32 byte I/O decode.

PC Compatibility

Example: VGA Peripheral

- $\text{PCI_BASE}[10:8] = 111$ to select a 128K memory space.
- PCI_BASE : IO bit = 0 to enable the 3rd decode to be in memory space instead of I/O space.
- $\text{PCI_BASE}[31:25] = 1110111$ to enable a decode of $\text{AD}[9:0] = 0x3B0 - 0x3BF$ for PCI I/O cycles.
- $\text{PCI_BASE}[24:19] = 111101$ to enable a decode of $\text{AD}[9:0] = 0x3C0 - 0x3DF$ for PCI I/O cycles.
- $\text{PCI_BASE}[18:17] = 01$ to decode PCI memory cycles in the address range $0xA0000 - 0xBFFFF$.

Chapter 11

Mailbox Registers

Occasionally it is necessary to pass small amounts of data or commands between the PCI bus and the Local bus. For example, an intelligent EPC based PCI disk controller may need to receive “get sector” commands from the x86 system host. Such commands could be sent to the local CPU by PCI-to-Local memory transfers into a command buffer, followed by a PCI interrupt request to indicate transfer completion. Using this method, however, is very inefficient, especially when only a handful of bytes need to be transferred.

To solve this problem, the EPC provides 16 mailbox registers which may be used to transmit and receive small amounts of data between the local CPU and the PCI bus. In addition, each mailbox register can request an interrupt to signal the receipt, or the demand, for more data.

Mailbox registers are also commonly used to emulate hardware registers in systems requiring backward register compatibility.

11.1 OVERVIEW

The EPC provides 16 8-bit mailbox registers arranged as contiguous bytes in both the Local and PCI internal register apertures. Each mailbox register is a dual ported memory, capable of generating an interrupt request whenever it is read or written from either side. Figure 52 shows a block diagram of the mailbox registers.

11.1.1 Accessing the Mailbox Registers

The mailbox registers are accessed from the Local bus through the Local-to-Internal Register (LB_IO_BASE) aperture. Typically, LB_IO_BASE is programmed during system initialization (see “Initialization”).

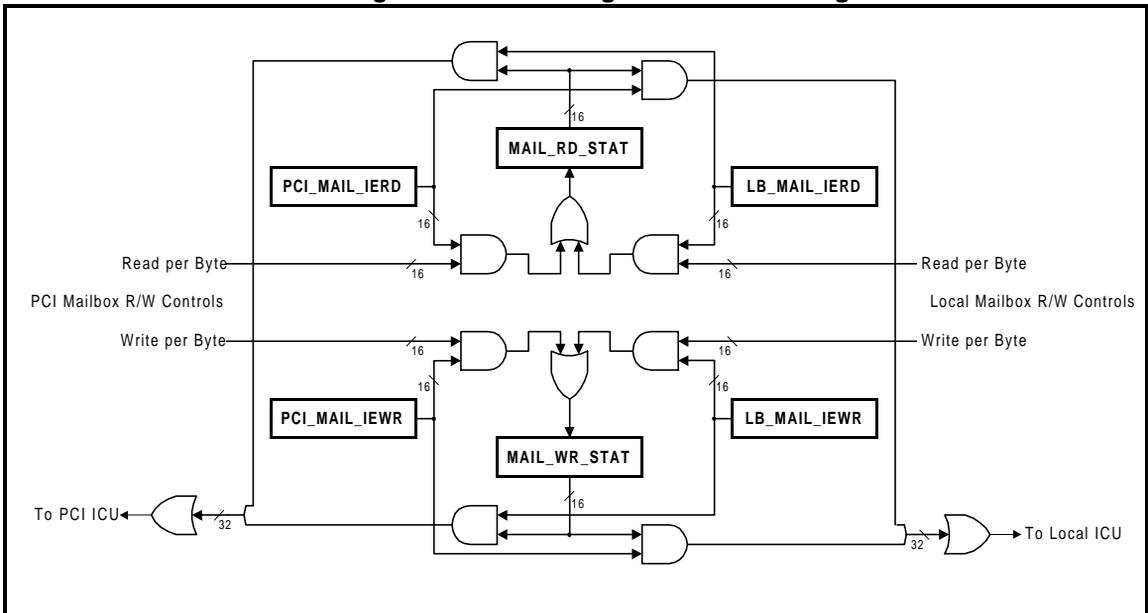
The mailbox registers are accessed from the PCI bus through the PCI-to-Internal Register (PCI_IO_BASE) aperture. Alternatively, the mailbox registers can be accessed through PCI configuration space for the EPC.

The mailbox registers may be accessed as byte, short, word, or multiple word quantities.

Mailbox Registers

Overview

Figure 52: Block Diagram of Mailbox Registers



11.1.2 Doorbell Interrupts

Each of the 16 mailbox registers can generate four different interrupt requests called *doorbell interrupts*. Each of these requests can be independently masked for each mailbox register. The four doorbell interrupt types are:

- Local interrupt request on read from PCI side
- Local interrupt request on write from PCI side
- PCI interrupt request on read from Local side
- PCI interrupt request on write from Local side

The PCI read and Local read interrupts are OR'd together and latched in the mailbox read interrupt status register (MAIL_RD_STAT). Similarly, the PCI write and Local write interrupts are OR'd together and latched in the mailbox write interrupt status register (MAIL_WR_STAT). All of the interrupt request outputs from the status registers are OR'd together to form a single mailbox unit interrupt request and routed to both the Local and PCI Interrupt Control Units.

When a block of mailbox registers are accessed simultaneously, for example when 4 mailbox registers are read as a word quantity, then each register affected will request a separate interrupt if programmed to do so.

11.2 PROGRAMMING THE MAILBOX REGISTERS

After RESET, interrupt requests for all mailbox registers are disabled. The programmer must specifically enable interrupt requests for each mailbox register and for each access type.

11.2.1 Enabling Doorbell Interrupt Requests

The four types of doorbell interrupts described above are enabled in four 16-bit registers as shown in Table 17.

Table 17: Doorbell Interrupt Types and Corresponding Enable/Mask Registers

ACTION CAUSING INTERRUPT	REGISTER
PCI side read	PCI_MAIL_IERD
PCI side write	PCI_MAIL_IEWR
Local side read	LB_MAIL_IERD
Local side write	LB_MAIL_IEWR

11.2.2 Clearing Doorbell Interrupt Requests

All of the interrupt requests from the 16 mailbox registers are logically OR'd together and then forwarded to the Local Interrupt Control Unit (LICU) and the PCI Interrupt Control Unit (see Figure 52). For an interrupt handler to clear the local mailbox/doorbell interrupt request in the LICU, it must clear *all* of the enabled local mailbox interrupt requests from the individual mailbox registers. This is done by clearing the corresponding bit(s) in the MAIL_RD_STAT and MAIL_WR_STAT registers (by writing a '1'), where the mailbox interrupt requests are latched. Similarly, to clear the PCI mailbox/doorbell interrupt request in the PICU, it must clear *all* of the enabled PCI mailbox interrupt requests from the individual mailbox registers. This is done by clearing the corresponding bit(s) in the MAIL_RD_STAT and MAIL_WR_STAT registers, where the mailbox interrupt requests are latched.

Note that MAIL_RD_STAT and MAIL_WR_STAT registers are cleared by writing a '1' into the bits to be cleared. Writing a '0' will have no effect. Interrupt status bits in the Local Interrupt Control Unit (LB_ISTAT) and the PCI Interrupt Control Unit (PCI_INT_STAT) are cleared by clearing the corresponding MAIL_RD_STAT and MAIL_WR_STAT registers.

Mailbox Registers

Programming the Mailbox Registers

12.1 OVERVIEW

As an I₂O compatible device, the EPC provides the following required features:

- An I₂O compatible “Address Translation Unit” (ATU). This is provided through a PCI configuration register at offset 0x10 and is called PCI_I2O_BASE.
- At offset 0x40 within the address range of the ATU aperture, an I₂O compatible Inbound FIFO port is provided.
- At offset 0x44 within the address range of the ATU aperture, an I₂O compatible Outbound FIFO port is provided.
- When enabled, a PCI interrupt may be generated whenever an outbound message has been posted. It is cleared when all outstanding outbound messages have been read. (This feature is not currently part of the I₂O standard 0.96 although it is implied and will undoubtedly become part of the standard in the future)

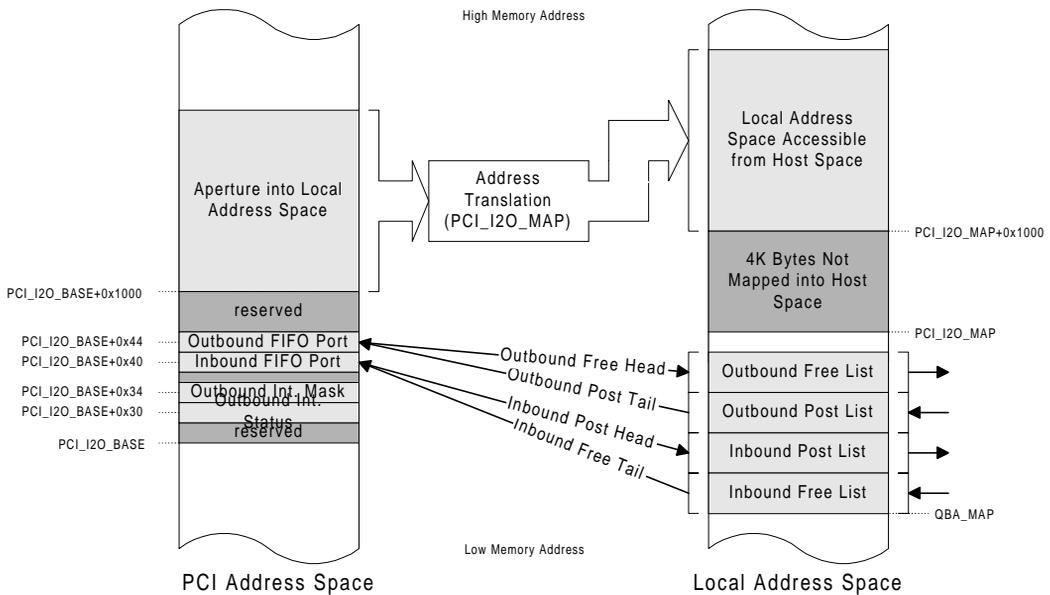
12.2 I₂O COMPATIBLE ADDRESS TRANSLATION UNIT

The I₂O compatible Address Translation Unit (ATU) provides a “window” or “aperture” for an external PCI agent to access the local bus address space. In this regard it is exactly the same as the 2 non-I₂O compatible PCI-to-Local apertures found on the VxxxPBC. However, there is one important difference: the I₂O ATU has two predefined special registers in its address range. These are the Inbound/Outbound FIFO registers at offset 0x40 and 0x44 in the ATU address space. These locations don't directly map to local address space. Instead, reads and writes to these locations are managed as curricular queues in local memory. As far as the ATU is concerned, the relationship of PCI and local address spaces is depicted in the following diagram:

I2O Interface

I2O Compatible Address Translation Unit

Figure 53: Operation of the I2O Address Translation Function and In/Outbound Free/Post List Mapping



12.2.1 ATU Setup and Configuration

The I₂O compatible ATU is configured through 2 registers:

- PCI_I2O_BASE determines where in PCI system address space the ATU will be located
- PCI_I2O_MAP determines the size of the ATU aperture and the location of the aperture within local address space.

12.2.1.1 PCI_I2O_BASE Operation

The PCI_I2O_BASE register provides a standard PCI - I₂O compatible Address Translation Unit. It provides two main functions:

- Access to the I₂O inbound/outbound free list and post list FIFOs.
- An aperture into Local Bus address space that translates a PCI address into a local bus address.

The bottom 4K bytes of the aperture are not translated into local bus address space. In this 4K region only 16 bytes are used (0x30-0x37, 0x40-0x47) for the inbound/outbound registers. The I2O specification requires no other ports in this 4K space and the remainder of that space is reserved.

12.2.2 PCI_I2O_MAP Operation

Local bus memory can be accessed by a PCI master through the PCI_I2O_BASE register. The base address on the local bus is translated into a local bus address according to the value in the PCI_I2O_MAP register. This forms the local bus side of the I₂O compatible “Address Translation Unit”. The PCI address is translated into a local bus address in the following manner:

$$\text{Local Bus Address} = \text{PCI Address} - \text{PCI_I2O_BASE Base Address} + \text{PCI_I2O_MAP Address}$$

Where:

PCI Address is within the aperture size defined by the PCI_I2O_MAP register

PCI_I2O_BASE Base Address is on an aperture sized boundary

PCI_I2O_MAP Address is on an aperture sized boundary

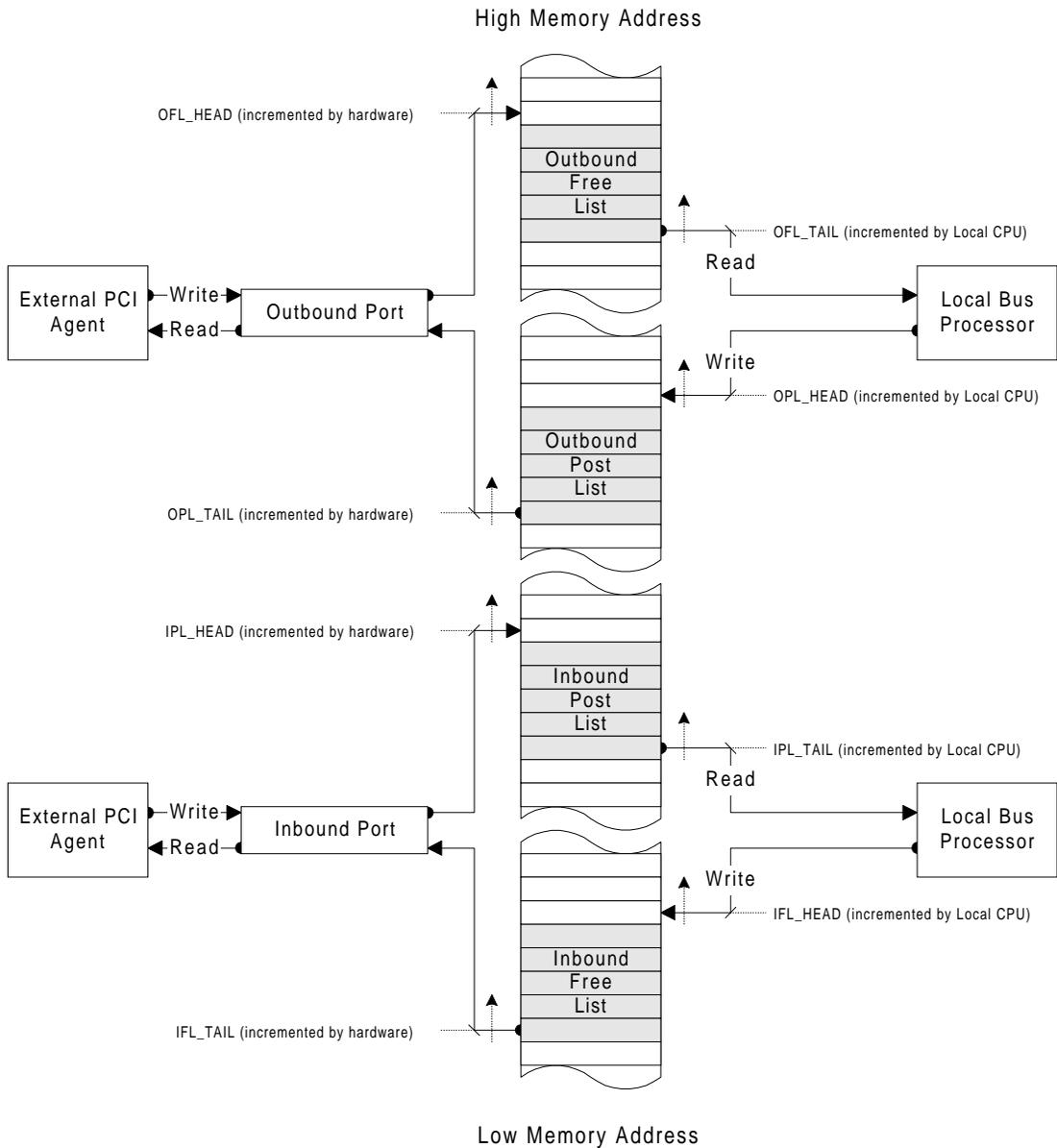
12.3 INBOUND/OUTBOUND QUEUE MANAGEMENT

An I2O driver operating on the host or external PCI agent will exchange Message Frame Addresses (MFA) with the local processor (acting as the Intelligent I/O processor). The exchange of these MFAs is managed as a circular queue or FIFO (first in first out) storage. The physical storage of the MFAs is in external memory on the local bus (the I2O specification itself doesn't make any assumptions about where the storage is located). There are 2 pairs of queues provided:

- The inbound free list and post list queues
- The outbound free list and post list queues

For each of the 4 queues in memory data flows in only a single direction: either PCI-to-Local or Local-to-PCI. The following diagram illustrates the flow of data in and out of these queues located in local memory.

Figure 54: In/Outbound Free/Post List Queue Operation



12.3.1 Queue Pointers

A set of pointer registers provide the mechanism for writing and reading local bus memory so that circular queue operation can be achieved. These registers are now described.

The management of a queue is accomplished with a tail and head pointer. Data is “pushed” into the FIFO queue at the location pointed to by the Head Pointer. Data is “popped” out of the FIFO queue at the location pointed to by the Tail Pointer. Pushing or popping causes the

respective head or tail pointer to be incremented. If pushing or popping is accomplished by the external PCI agent then the respective pointer is automatically incremented by the EPC. Pointers that are managed by the local processor are also updated by it under software control.

Table 18: I2O Queue Pointers

Reg. Name	Offset	Description	Pointer Maintenance
OFL_HEAD	0xBC	Outbound Free List Head Pointer	PCI write of Outbound Port auto-increments
OFL_TAIL	0xB8	Outbound Free List Tail Pointer	Updated by local processor
OPL_HEAD	0xB4	Outbound Post List Head Pointer	Updated by local processor
OPL_TAIL	0xB0	Outbound Post List Tail Pointer	PCI read of Outbound Port auto-increments
IPL_HEAD	0xAC	Inbound Post List Head Pointer	PCI write of Inbound Port auto-increments
IPL_TAIL	0xA8	Inbound Post List Tail Pointer	Updated by local processor
IFL_HEAD	0xA4	Inbound Free List Head Pointer	Updated by local processor
IFL_TAIL	0xA0	Inbound Free List Tail Pointer	PCI read of Inbound Port auto-increments

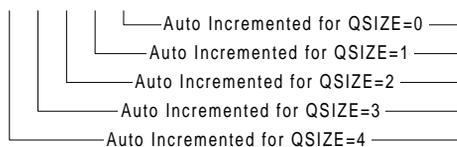
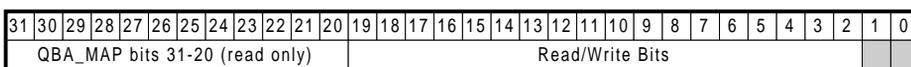
It is important to follow an orderly initialization process for the pointers. Typically all the pointers would be initialized by the local processor. Once the queues have been initialized and the ONLINE bit in the QBA_MAP register has been set, the local processor will be unable to modify the contents of any of the pointers controlled by the host driver namely OFL_HEAD, OPL_TAIL, IPL_HEAD and IFL_TAIL. These are maintained automatically when the Inbound/Outbound ports are read by the PCI agent as the above table indicates.

11

12.3.1.1 Pointer Format

Each pointer is a 32 bit address into local memory. Since pointers are always aligned to a 32 bit boundary, the low 2 bits (bits 1-0) are read only ('0'). The upper 12 bits of the address (bits 31-20) of all 8 pointers are derived from a single register - the Queue Base Address register (QBA_MAP) described in the next section. The format of each of the 8 pointer registers is described in the next diagram:

Figure 55 Figure 3 Pointer Register Bit Format



Only 4 of the pointers are automatically incremented by hardware. These are OFL_HEAD, OPL_TAIL, IPL_HEAD and IFL_TAIL. Incrementing is always done on a modulo boundary of queue size (queue size is determined by the QBA_MAP register described in the next section). The pointer format diagram above shows where those modulo boundaries occur for each of the queue size options. As an example, consider a queue size of 4K entries

I2O Interface

PCI I2O Interrupt Registers

(QSIZE=0) which translates to 16K bytes. Pointers are incremented in the following way:

$$\text{New Pointer} = (\text{Old Pointer} + 4 \text{ bytes}) \& 0x3FFC$$

Where the “&” symbol represents the bitwise AND operation

The bitwise AND with 0x3FFC (16K-4) represents the wrapping of the pointer at the modulo boundary of queue size.

12.3.2 QBA_MAP Register: Locating the Queue in Local Memory

All 8 of the head/tail pointers are located within the same 1M byte region of local memory. This region is selected by programming the QBA_MAP register. The QBA_MAP register also determines the sizes of the 4 queues (all 4 queues must be of the same size).

12.3.3 PCI Inbound/Outbound Port Read/Write Cycles

When the Inbound/Outbound queue ports are accessed by a PCI agent, the PCI cycle looks like any normal access to an aperture (or ATU) with the following exceptions:

- Access to the Inbound/Outbound queue ports at offset 0x40 and 0x44 cannot be bursted. If a burst is attempted to these locations then the EPC will do a “Disconnect - B” (see PCI Specification) where STOP# is asserted with TRDY# to turn the burst into a single cycle.
- Reads of the Inbound/Outbound queue ports are never prefetched regardless of the state of the PREFETCH bit in the PCI_I2O_BASE register.
- Each PCI read/write operation to the Inbound/Outbound queue ports is translated into an equivalent read/write cycle on the local bus (as an access to the ATU would do) except that the address is derived from the head/tail pointers instead of the PCI address in combination with the PCI_I2O_MAP value.
- if the value of IFL_TAIL is equal to IFL_HEAD then a read of the inbound free list port will return 0xFFFFFFFF and no read cycle is performed on the local bus.
- if the value of OPL_TAIL is equal to OPL_HEAD then a read of the outbound post list port will return 0xFFFFFFFF and no read cycle is performed on the local bus.
- if the I2O “ONLINE” bit in QBA_MAP isn’t enabled then no local bus cycles will be performed in response to an access to the inbound/outbound ports. Read operations will return 0xFFFFFFFF. The outbound post interrupt is also disabled.

12.4 PCI I2O INTERRUPT REGISTERS

I2O requires that a maskable PCI interrupt be generated to the selected interrupt pin

whenever the outbound post list is not empty. The status register is mapped into offset 0x30 in the ATU PCI address space (as determined by PCI_I2O_BASE) and the mask register adjacent to it at offset 0x34. These are 32 bit registers when accessed from the I2O ATU with only a single bit defined in the lowest byte. The same register can also be accessed from the standard PCI and local bus internal register apertures in the same manner as any other internal register. However, the registers are defined as 8 bit in these cases.

12.4.1 I₂O Ready Interrupt

The I2O specification requires that if interrupts are to be used then an interrupt mask and status register are to be provided at offsets 0x34 and 0x30 in the base address region allocated to BAR0 (configuration space offset 0x10: PCI_I2O_BASE). These registers are provided by aliasing the OUT_POST bits of

PCI_I2O_IMASK (offset 0x34)				
Bits	Mnemonic	Type	Reset	Description
31-4	-	R	0H	reserved
3	OUT_POST	FRW	0H	Outbound Post Mask: when clear (0) the interrupt pin 0 is driven whenever the outbound post FIFO is not empty. Setting this mask bit disables the physical interrupt pin from being driven but has no effect on the corresponding status bit in PCI_I2O_ISTAT. This bit is aliased to the OUT_POST bit in PCI_INT_CFG except in the opposite polarity ('1' = enabled in PCI_INT_CFG whereas '0' = enabled in PCI_I2O_MASK). Changes to PCI_I2O_MASK bit 3 will also be reflected in PCI_INT_CFG.
2-0	-	R	0H	reserved

11

PCI_I2O_ISTAT (offset 0x30)				
Bits	Mnemonic	Type	Reset	Description
31-4	-	R	0H	reserved
3	OUT_POST	FRW	0H	Outbound Post Status: set (1) whenever the outbound post FIFO is not empty. Cleared when the outbound post FIFO is empty again. The state of the corresponding mask bit has no effect on this status bit. This bit is aliased to the OUT_POST bit in PCI_INT_STAT.
2-0	-	R	0H	reserved

12.5 ENABLING I₂O OPERATION

The EPC is configured for I₂O operation by setting the I2O_EN bit in the PCI_CFG register. When configured for I₂O operation, some of the VxxxPBC internal operations change:

- PCI_IO_BASE register is re-located to offset 0x18 (0x10 is the default)
- PCI_BASE1 and PCI_MAP1 are disabled
- PCI_I2O_BASE is enabled at offset 0x10 in the PBC internal register space.

I2O Interface

Enabling I2O Operation

- PCI_I2O_MAP is enabled at offset 0x44 in the PBC internal register space.
- PCI_BASE0 must be disabled (via PCI_MAP0) as it is used to control some aspects of the in/outbound queue operation.

There are also a set of registers that are dedicated for I₂O operation. Normally they wouldn't be used if I₂O operation isn't enabled (although the registers are accessible even when I₂O is disabled).

- OFL_HEAD: Outbound Free List Head Pointer
- OFL_TAIL: Outbound Free List Tail Pointer
- OPL_HEAD: Outbound Post List Head Pointer
- OPL_TAIL: Outbound Post List Tail Pointer
- IPL_HEAD: Inbound Post List Head Pointer
- IPL_TAIL: Inbound Post List Tail Pointer
- IFL_HEAD: Inbound Free List Head Pointer
- IFL_TAIL: Inbound Free List Tail Pointer

Chapter 13

Interrupt Control

The EPC includes two Interrupt Control Units (ICUs): one to process interrupt requests bound for the local CPU, and one to process interrupt requests to and from the PCI bus. The two Interrupt Control Units are connected to allow the routing of interrupt requests from PCI to the Local bus, as well as from the Local bus to PCI.

The PCI Interrupt Control Unit (PICU) also includes a special *crosspoint interrupt routing mechanism* for the four PCI interrupt requests (INTA through INTD). Each of the four PCI interrupts can function as either an input or an output, and interrupt requests may be passed from any PCI interrupt to any other PCI interrupt. PCI interrupt acknowledge cycles can also be generated by the EPC when acting as a host bridge.

13.1 LOCAL INTERRUPT CONTROL UNIT

The Local Interrupt Control Unit (LICU) process interrupts from the following sources:

- PCI read and write exceptions
- DMA chain completion
- Mailbox register access (“doorbell” interrupt)
- PCI Interrupt pin events

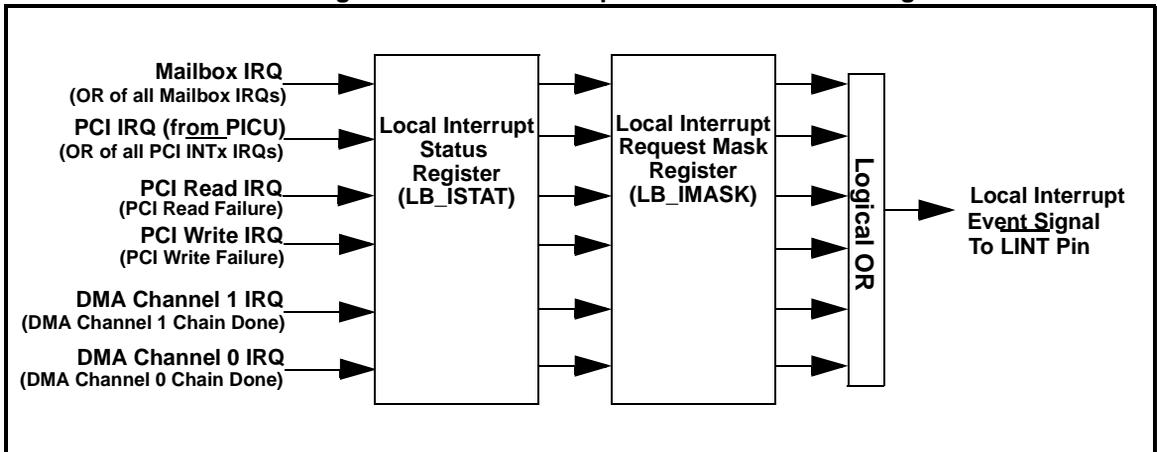
13.1.1 Overview

The LICU consists of an interrupt status register (LB_ISTAT) and an interrupt mask register (LB_IMASK) as shown in Figure 56. Interrupt requests from the individual sources are posted in the interrupt status register. The interrupt mask register controls which of the interrupt requests posted in the request register actually generate an interrupt request to the local processor, and optionally, the PCI Interrupt Control Unit.

Interrupt Control

Local Interrupt Control Unit

Figure 56: Local Interrupt Control Unit Block Diagram



13.1.2 Local Interrupt Requests

The following local interrupt requests are physically latched within the LB_ISTAT register:

- "PCI_RD", "PCI_WR" bits: PCI Read and Write Error Interrupts
- "DMA0", "DMA1" bits: DMA Channels 0 and 1 Interrupts
- "MAILBOX" bit: Mailbox interrupt requests¹

The above interrupt requests are cleared by clearing (0) the corresponding bits in the LB_ISTAT register.

The following local interrupts are not latched within the LB_ISTAT register:

- "PCI_INT" bit: PCI interrupt control unit requests

Non-latched interrupt requests are only cleared when the source of the corresponding interrupt request is cleared. For example, the PCI_INT request is only cleared when *all* enabled PCI interrupt inputs are de-asserted.

13.1.3 Masking Local Interrupt Requests

Local interrupt requests can be masked (disabled) by clearing the corresponding bit in the LB_IMASK register. The interrupt mask register controls (enables) which of the interrupt requests posted in the request register actually generate an interrupt request.

13.1.4 Local Interrupt Event Signal

The LICU has a single output signal that is the logical OR of all unmasked interrupt requests.

1. The MAILBOX bit is not latched in silicon revision B2 or later.

This signal is routed to the local processor interrupt pin ($\overline{\text{LINT}}$) and to the PCI Interrupt Control Unit. The activation of the LICU output signal in response to an interrupt is called a *local interrupt event*.

13.2 PCI INTERRUPT CONTROL UNIT (PICU)

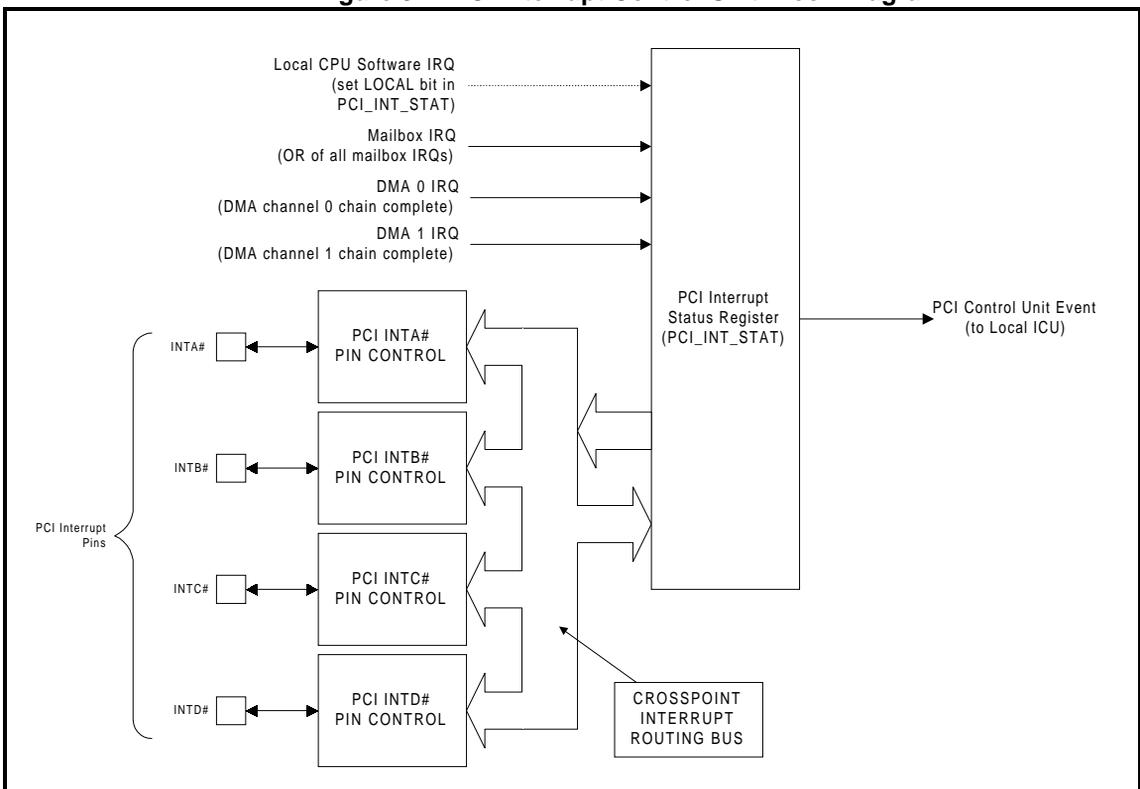
The PCI Interrupt Control Unit (PICU) process interrupts from the following sources:

- DMA Chain Completion
- Mailbox Register Access (“Doorbell” interrupt)
- Local bus direct software interrupts
- Interrupt requests from the $\overline{\text{INTA}}$, $\overline{\text{INTB}}$, $\overline{\text{INTC}}$, and $\overline{\text{INTD}}$ PCI interrupt request pins (when configured as inputs)

13.2.1 Overview

The PICU is significantly more complex than the LICU, as shown in Figure 58. The interrupt crosspoint mechanism allows for maximum system design flexibility for embedded systems using multiple PCI interrupts. In addition, control is provided for both receiving and requesting interrupts on any of the four PCI interrupt pins.

Figure 57: PCI Interrupt Control Unit Block Diagram



Interrupt Control

PCI Interrupt Control Unit (PICU)

13.2.2 PCI Interrupt Pins (\overline{INTA} through \overline{INTD})

The PCI Specification provides for four shared PCI interrupt request pins: \overline{INTA} through \overline{INTD} . Typically, these interrupts are driven by the target devices in a system and received by the host bridge device. Since the EPC can be used as either a host or target bridge, the capability is provided to either receive interrupts or drive interrupt requests on the \overline{INTA} through \overline{INTD} pins.

13.2.2.1 Configuring a PCI Interrupt Pin as an Interrupt Request Output

Any of the \overline{INTx} pins can be configured as an output pin. However, only *one* of the \overline{INTx} pins can be designated as the destination for mailbox, local direct, and DMA interrupt requests (chosen by the INT_PIN field in the PCI_BPARAM register). The remaining \overline{INTx} outputs can be programmed to reflect the state of other \overline{INTx} pins configured as *inputs* (see Crosspoint Routing Mechanism, below).

The direction of the \overline{INTx} pins is set via the MODEx fields in the PCI Interrupt Configuration register (PCI_INT_CFG). Pins configured as outputs are always active low and are software cleared. Software cleared outputs will go inactive when the corresponding bit in the PCI_INT_STAT register is cleared.

13.2.2.2 Configuring a PCI Interrupt Pin as an Interrupt Request Input

Any of the \overline{INTx} pins may be configured as inputs. The direction and detection mechanism for \overline{INTx} pins is controlled by the MODEx field in the PCI_INT_CFG register. \overline{INTx} inputs may be either edge or level sensitive. Edge sensitive pins will generate an interrupt event when a high-to-low transition is seen on the pin. Level triggered inputs will generate an interrupt event whenever the state of the \overline{INTx} pin is low. (Note that level triggered inputs will generate another interrupt event if a previous request is cleared AND the corresponding \overline{INTx} pin is still at a logic "0".)

PCI interrupt inputs will post interrupt requests in the PCI_INT_STAT register *only when they are routed to a PCI interrupt output* through the crosspoint routing mechanism (see below).

13.2.2.3 Crosspoint Interrupt Routing Mechanism

Many embedded designs will require the ability to control multiple PCI interrupt input and output events. The EPC's crosspoint interrupt routing mechanism allows for extremely flexible interrupt request routing between the four interrupt pins.

For example, consider the case of an expandable network router using PCI as the backplane. In such a system there may be no single "host processor". Each plug in board may act as a host processor from time to time, and will need the ability to both receive interrupt requests as well as post them. It would be nice if such a system could be designed to be "plug and play", so that when a new board was added, it automatically configured some of its interrupts as inputs, and some as outputs *dynamically*. The interrupt crosspoint routing mechanism allows the system designer to do just that.

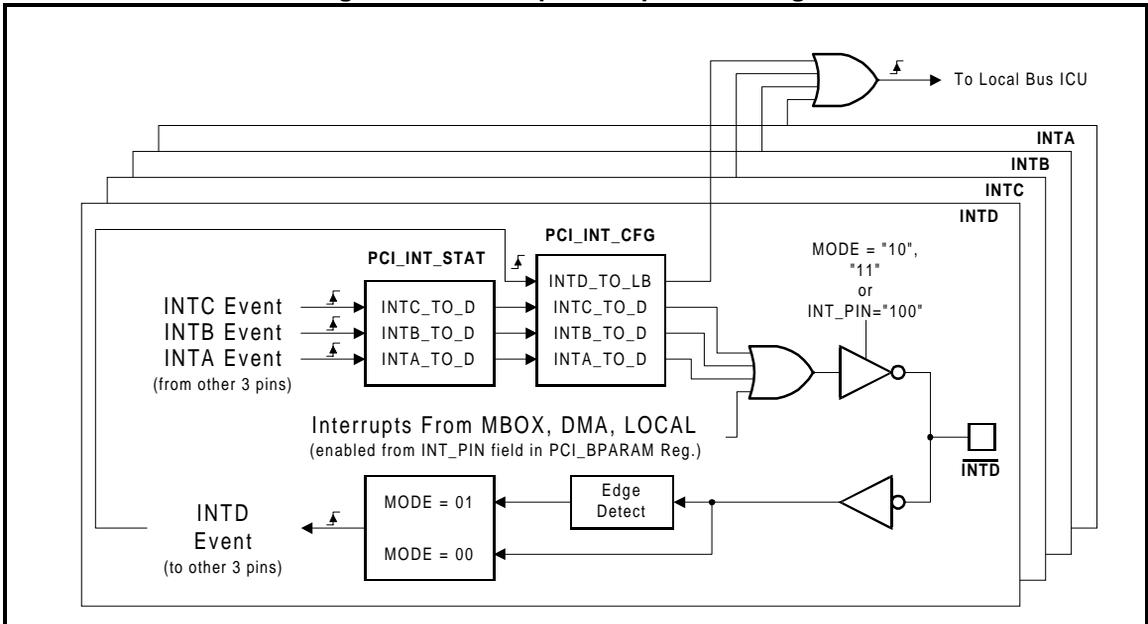
Figure 58 shows the details of the crosspoint routing mechanism. Each $\overline{\text{INTx}}$ pin is capable of generating a PCI interrupt event, which is in turn routed to all of the other $\overline{\text{INTx}}$ pin control circuits. When an $\overline{\text{INTx}}$ pin is configured as an output, it can be driven internally from either one or more of the other 3 $\overline{\text{INTx}}$ pin interrupt events, or from the combination of the mailbox, DMA, and local direct source.

The routing of interrupts is controlled through the INTx_TO_y bits in the PCI_INT_CFG register. For example, if the INTA_TO_D bit is set and INTA is configured as an input pin and INTD is configured as an output, then when INTA is active INTD will go low to request an interrupt on the PCI bus.

The individual $\overline{\text{INTx}}$ interrupt requests may also be routed to the Local Interrupt Control Unit. Each pin has an associated INTx_TO_LB bit that selects it as an input to the LICU. All PCI interrupt pin requests are OR'd before presentation to the LICU (see Figure 58).

PCI interrupt configuration is shown in more detail in the programming example below.

Figure 58: Interrupt Crosspoint Routing Mechanism



13.2.3 Internal PCI Interrupt Requests

In addition to the $\overline{\text{INTx}}$ pins, the PICU can also respond to request from the mailbox registers, the DMA controller, and the local CPU. This group of PCI interrupt request sources is called *internal PCI interrupt requests* to differentiate them from the $\overline{\text{INTx}}$ pins.

13.2.3.1 Mailbox and DMA PCI Interrupt Requests

Like the local Interrupt Control Unit, the PCI Interrupt Control Unit can receive interrupt requests from both the mailbox registers and the DMA controller. Also like the LICU, the PICU latches the DMA interrupt requests and *does not latch* the mailbox interrupt request.

Interrupt Control

PCI Interrupt Control Unit (PICU)

Clearing the DMA interrupt requests is achieved by writing "1" to the corresponding bit in the PCI_INT_STAT register. Note that clearing a DMA interrupt request in PCI_INT_STAT has no effect on the DMA interrupt request bits in the LB_STAT register.

Mailbox interrupt requests are only cleared by clearing the individual mailbox interrupt requests in the mailbox register unit.

13.2.3.2 Local Direct Interrupt Request

The Local processor may request a PCI interrupt by setting the LOCAL bit in the PCI_INT_STAT register. The LOCAL bit can be cleared from the PCI or Local side of the bridge.

13.2.3.3 Routing the Internal PCI Interrupt Requests to an $\overline{\text{INTx}}$ Pin

Only one $\overline{\text{INTx}}$ pin can be the destination for internal PCI interrupts. The specific pin is determined by programming the INT_PIN field in the PCI_BPARAM register. The pin used for this purpose may also be used as the destination for crosspoint routed $\overline{\text{INTx}}$ interrupts.

13.2.4 PICU Configuration Example

An example may be helpful in describing the operation of the PCI Interrupt Control Unit. Figure 59 shows a system block diagram for the following example. The interrupt pin usage is detailed in Table 19.

First, we'll need to mask DMA and mailbox interrupts from appearing on the $\overline{\text{INTC}}$ pin when we "turn it on" as an output (this will prevent spurious interrupts from occurring). These interrupts are disabled by clearing the MAILBOX and DMA0/1 bits in the PCI_INT_CFG register.

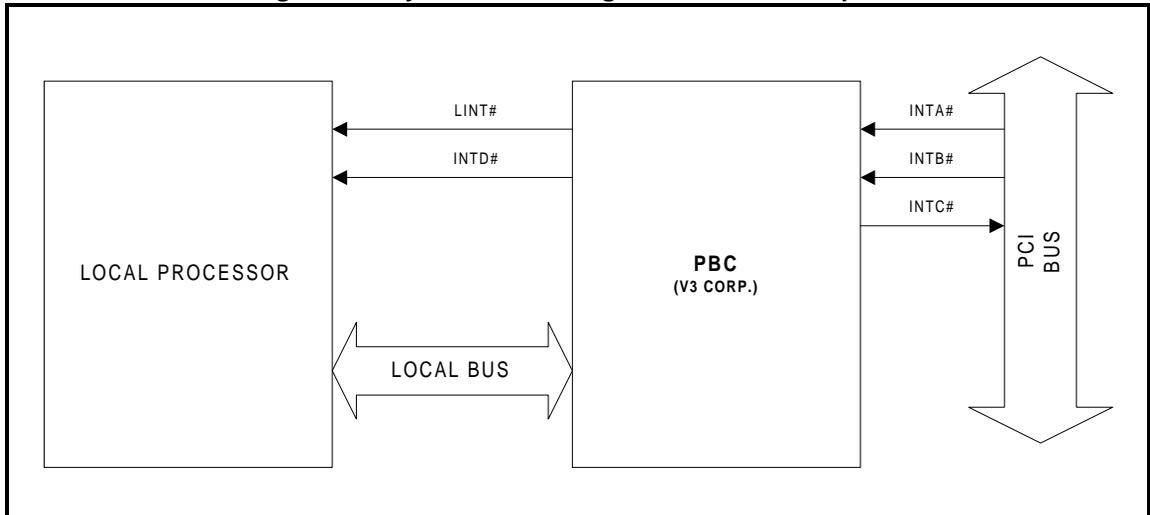
The next step is to configure $\overline{\text{INTD}}$ and $\overline{\text{INTC}}$ as outputs. This is done by setting the appropriate mode in the MODED and MODEC fields in the PCI_INT_CFG register. We'll choose "software cleared" for both (MODEx=10). Since $\overline{\text{INTC}}$ needs to be able to drive the local direct interrupt onto the PCI bus, we must also set INT_PIN field in the PCI_BPARAM register to "011" (this sets $\overline{\text{INTC}}$ as the "receiver" for internal PCI interrupts). Because all of the PCI interrupts are open drain (by PCI definition), we'll need to make sure the hardware guys put a pullup resistor on the $\overline{\text{INTD}}$ pin that's used as a local interrupt (it's not on the block diagram).

Finally, we need to configure the $\overline{\text{INTA}}$ and $\overline{\text{INTB}}$ pins as inputs to $\overline{\text{INTD}}$ (the PCI interrupt routing request pin). This is done by setting the INTA_TO_D and INTB_TO_D bits in the PCI_INT_CFG register. Configuration is now complete.

Table 19: Example PCI Interrupt Usage

Interrupt Pin	Use
$\overline{\text{LINT}}$	Tied to one of the local CPU's interrupt input pins. Signals Local interrupt requests, including PCI errors (very high priority).
$\overline{\text{INTD}}$	Also tied to one of the local CPU's interrupt input pins. This pin is configured as an output and forwards PCI interrupt requests from the $\overline{\text{INTA}}$ and $\overline{\text{INTB}}$ pins to the local CPU (medium priority).
$\overline{\text{INTC}}$	Configured as an output, $\overline{\text{INTC}}$ is used to signal local direct processor requests to other subsystems in the PCI bus.
$\overline{\text{INTA}}$ and $\overline{\text{INTB}}$	Generic PCI interrupt inputs from target subsystems.

Figure 59: System Block Diagram for PCI Interrupt Control Unit



13.3 GENERATING PCI INTERRUPT ACKNOWLEDGE CYCLES

Generating PCI interrupt acknowledge cycles is straightforward with the EPC:

- Set the TYPE field in one of the LB_MAPx registers to “Interrupt Acknowledge”.
- Turn off prefetching for the corresponding aperture by clearing the PREFETCH bit in the LB_BASEx register.
- Perform a single word read access to the aperture. The data returned will be the interrupt vector.

Interrupt Control

Generating PCI Interrupt Acknowledge Cycles



Initialization of the EPC usually occurs when the system is reset. The EPC can be reset either from the PCI bus or from the local bus. Following a reset, the EPC's internal registers can be initialized via the PCI bus, the local bus, or the serial EEPROM interface. Internal registers can be modified after reset from the PCI bus (memory, I/O, or configuration space) or from the local bus. This chapter describes reset options and register initialization procedures.

14.1 RESET DIRECTION

The EPC device is reset by driving active either the $\overline{\text{LRST}}$ pin (for resetting from the local bus) or $\overline{\text{PRST}}$ pin (for resetting from the PCI bus). Which pin is used as the EPC reset input is controlled by the RDIR pin.

The EPC is also capable of generating reset for either the local or the PCI bus. For systems using the EPC as the PCI system master bridge, PCI reset is typically driven by the EPC. Systems using the EPC as a target bridge will typically receive reset via the PCI bus, and reset the host processor in turn. Figure 60 shows examples of both reset directions.

Eight clock cycles are required for both the PCI and local bus following the rising edge of reset before attempting to access the internal registers. This eight clock delay is necessary to allow the internal operations of the EPC to achieve an idle state.

Figure 60: Reset Direction Examples

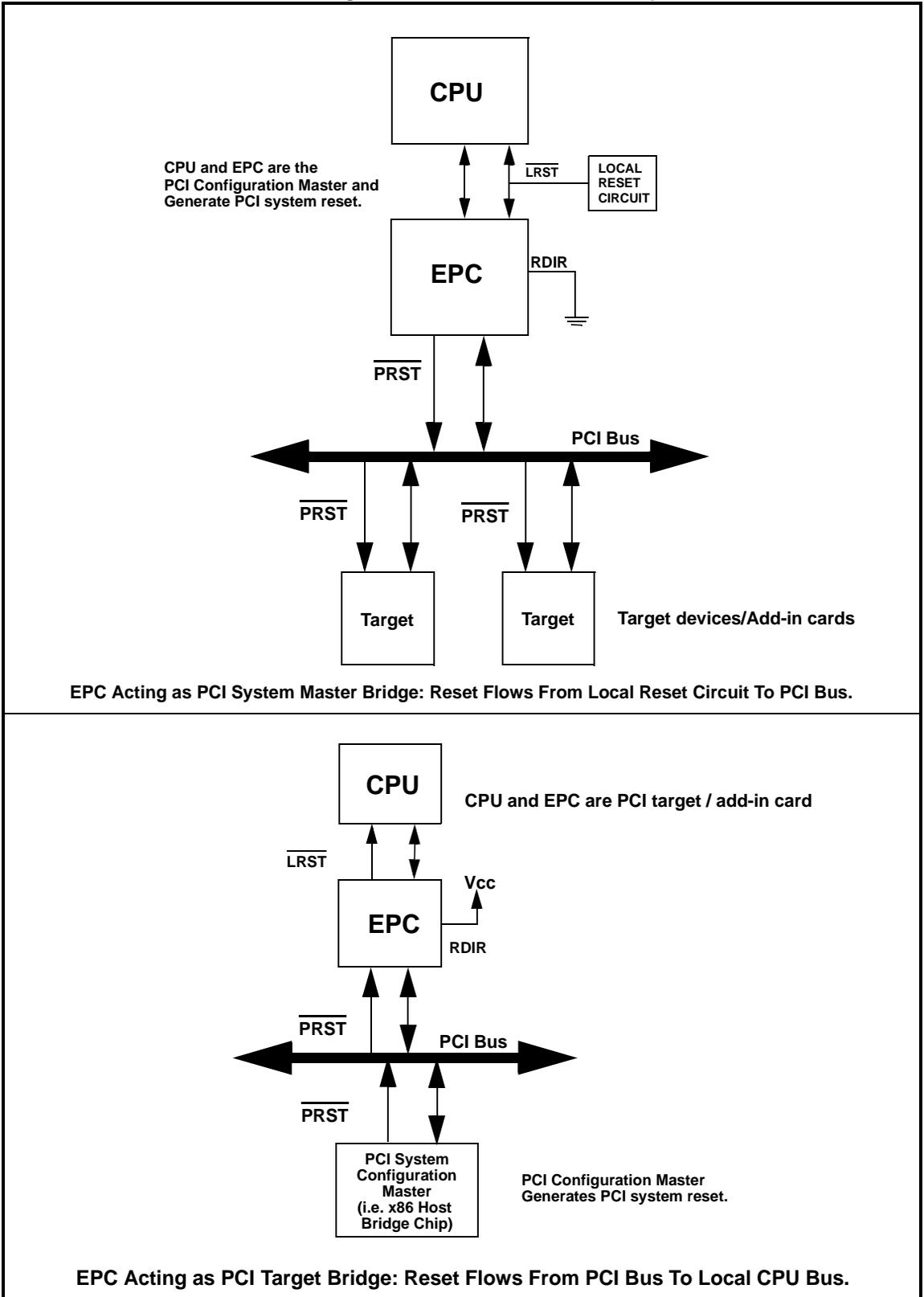


Table 20: RESET Direction Options

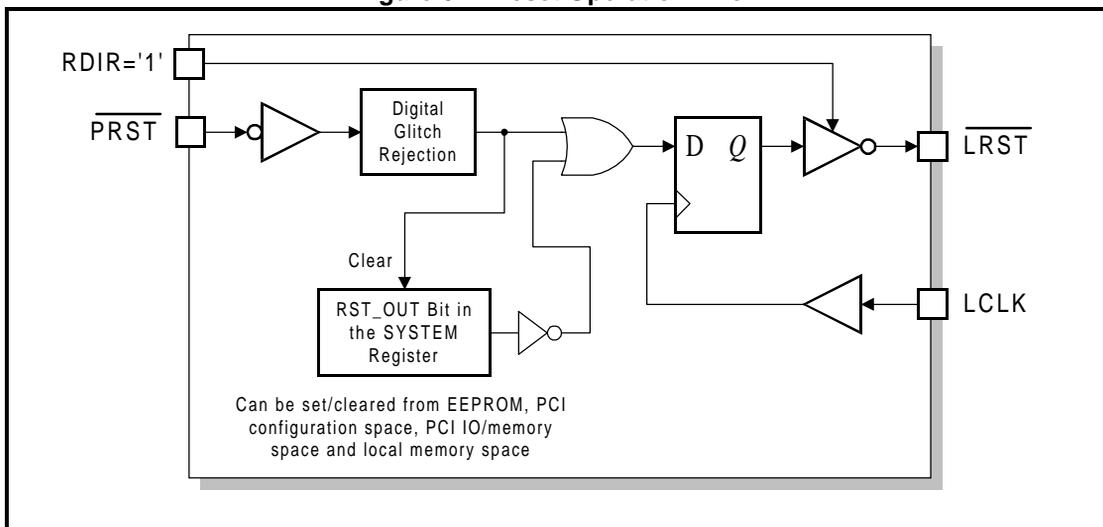
RDIR Pin State	Direction	$\overline{\text{LRESET}}$	$\overline{\text{PRST}}$	Comments
0	Local to PCI	Input	Output	$\overline{\text{PRST}}$ is deasserted by writing the RST_OUT bit in the SYSTEM register
1	PCI to Local	Output	Input	$\overline{\text{LRST}}$ is deasserted by writing the RST_OUT bit in the SYSTEM register

The reset used as output remains asserted until the input Reset has been de-asserted and the RST_OUT bit in the SYSTEM register has been written “1”. The RST_OUT bit can be written ‘0’ later to cause the output Reset to be asserted again (software controlled Reset).

The following diagram illustrates how the EPC reset operation works with the RDIR pin tied high. In this mode the PCI bus provides the reset input via $\overline{\text{PRST}}$. This $\overline{\text{PRST}}$ input is processed to produce a local Reset output $\overline{\text{LRST}}$.

In many systems, it will be desirable to un-reset the local bus after the PCI bus has been released from reset. This is accomplished automatically by programming the serial EEPROM with bit 7 set in byte 79H (this is the RST_OUT bit in the SYSTEM register).

Figure 61: Reset Operation when RDIR = 1

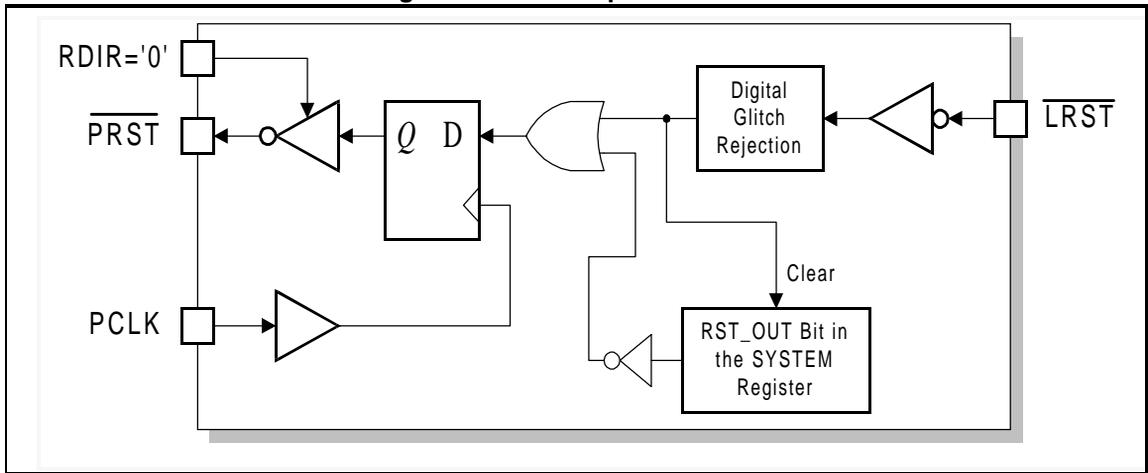


By tying the RDIR pin low, the Reset direction can be configured to drive out the PCI reset and use $\overline{\text{LRST}}$ as an input.

Initialization

Initializing the Internal Registers

Figure 62: Reset Operation when RDIR = 0



14.2 INITIALIZING THE INTERNAL REGISTERS

There are three options for initializing the EPC internal registers set after reset:

- Initialization from the local bus side (through the aperture defined by the LB_IO_BASE register)
- Initialization from the PCI bus side via configuration space
- Initialization from the serial EEPROM interface

14.2.1 Selecting Initialization Mode

There are 3 initialization modes that can be selected to control the operational of the EEPROM controller during initialization (see Table 21).

Table 21: EEPROM Initialization Options

EEPROM Port Connection	RETRY_EN	RST_OUT	Comment
SDA pulled high, No EEPROM	1	1	Typically used for initialization via local processor
SDA Tied Low, No EEPROM	0	0	Typically used for initialization via PCI
SDA ^a and SCL connected to valid EEPROM device	From EEPROM	From EEPROM	Initialization from EEPROM

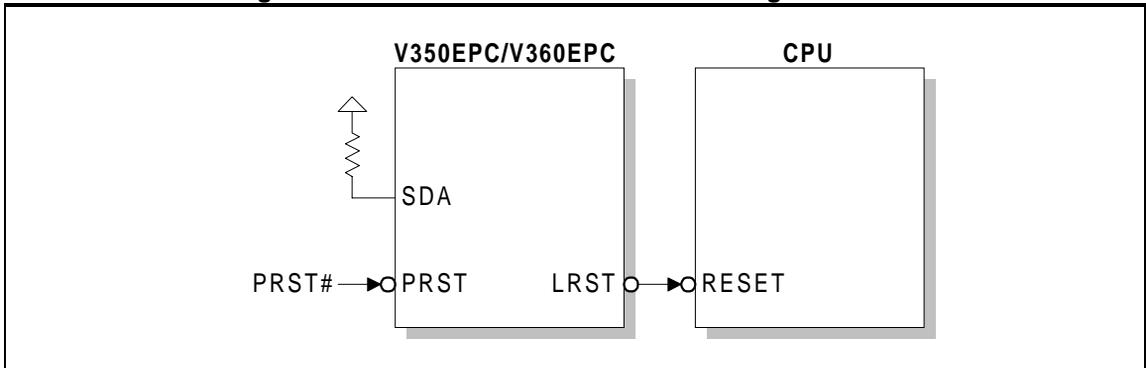
a.SDA should have a pull up resistor in order to function properly with an EEPROM device

After the input reset has been released, 10 SCL clocks are performed followed by a STOP cycle to ensure that a re-reset didn't falsely see the SDA signal low.

14.2.2 Initialization Using the Local Processor

In applications where the EPC is a secondary master it may be desirable to use the local processor to initialize the contents of the internal registers of the EPC and save the cost of the serial EEPROM device. This is best accomplished by pulling the SDA pin high on the EPC and using PRST# as an input. The processor can be connected as in Figure 63.

Figure 63: Connection for Initialization Using the Local Processor



When connected as in Figure 63, the RETRY_EN bit of the PCI_CFG register will be set when PRST# is asserted and remain that way until the local CPU clears it. RETRY_EN is used to cause a PCI configuration access to the EPC to get retried until it is cleared from the local CPU. This ensures that the local bus CPU gets a chance to properly initialize before the PCI BIOS tries to enumerate the EPC. Once the local CPU has initialized the internal registers then RETRY_EN can be cleared to allow the primary PCI master to enumerate the EPC. Since the RETRY_EN bit is type FR (locked) the sequence that software should use is: write RETRY_EN with '0' *immediately* followed with writing the LOCK bit with '1' in the SYSTEM register.

Access to the internal registers from the local bus side is through the “local bus-to-internal register” aperture defined by the LB_IO_BASE register. LB_IO_BASE is a sixteen bit register which defines a field that is compared with address bits A[31:16] for *every access* seen on the local bus. When there is a match between LB_IO_BASE and A[31:16], that access is “claimed” by the EPC and is considered to be to an access to the internal registers. The local-to-internal register aperture has a granularity of 64Kbytes (since only A[31:16] are compared) even though only the first 256 bytes are used. The remaining space is reserved.

As an example, let's assume LB_IO_BASE is programmed to the value 1234H. To read the PCI_CC_REV register (offset 08H) you would perform a word read from location 1234.0008H in local memory. Care must be taken when using big-endian processors (or big-endian regions with little-endian processors) to generate the proper addresses, as configuration space is little-endian (by PCI definition).

In order for the LB_IO_BASE decoder to respond to a local bus cycle at the desired memory location, it must be configured. There are 3 ways to do this:

Initialization

Initializing the Internal Registers

- LB_IO_BASE can be downloaded from the serial EEPROM device
- LB_IO_BASE can be configured from the PCI bus through configuration space or through the PCI_IO_BASE register
- LB_IO_BASE can be configured by the local processor by capturing the first write cycle on the local bus

After RESET, the local bus interface on the EPC will respond to any write cycle it sees on the bus until the LB_IO_BASE register is written (regardless of how it is written). Therefore, if the serial EEPROM download is enabled, then the contents of LB_IO_BASE is established in this way and the EPC will only respond to addresses in the range determined by LB_IO_BASE. The same is true of initialization from the PCI bus.

If initialization of LB_IO_BASE is to be performed by the local processor then it should be done as the first (or one of the first) write cycle. The address should be xxxx.006Ch¹ where xxxx is the base address that internal EPC registers are to be mapped in local space. The data for the write should also be the base address (this is the data that will actually be written into the register). If the first write is NOT to location xxxx.006Ch then the EPC will also claim the access and write one of the internal registers (the low address determines which register is selected). When xxxx.006Ch is eventually written then the EPC will only respond at the location determined by the value loaded into LB_IO_BASE. For the V960EPC with it's 16 bit bus, LB_IO_BASE is initialized with a single write to xxxx.006Eh which is the upper (and only significant) part of the LB_IO_BASE register. In summary, when the EPC is initialized by the local bus processor (as opposed to PCI bus or serial EEPROM):

- The EPC will capture all write cycles on the local bus until the LB_IO_BASE register (at offset 6Ch) is written by something (the Serial EEPROM, PCI bus or local bus processor)
- Writes to addresses other than xxxx.006Ch will also be captured by the EPC.
- All cycles that the EPC captures will generate a "data ready" to acknowledge the transfer

Once the LB_IO_BASE register is set, the remaining internal registers are programmed by writing to the corresponding memory mapped locations. Burst reads and writes are permitted to internal registers.

All PCI and local bus functions are disabled following a hardware reset. For example, all PCI accesses are ignored by the EPC until enabled by the programmer (with the exception of PCI configuration cycles as described below).

1. The LB_IO_BASE register is in the upper 16 bits for a 32-bit write. Address 6Eh should be used for a 16-bit write using a little endian processor.

Table 22: Local Bus Signals for LB_IO_BASE Configuration after RESET

CPU	READ/ WRITE SIGNAL	$\overline{\text{BE3}} /$ BWE3	$\overline{\text{BE2}} /$ BWE2	$\overline{\text{BE1}} /$ BWE1	$\overline{\text{BE0}} /$ BWE0	ADDRESS	DATA
i960Jx i960Cx i960Hx	WRITE	0	0	0	0	xxxx.006CH	D[31:16] = LB_IO_BASE D[15:0] = 006CH
i960SA	WRITE	NA	NA	0	0	xxxx.006EH xxxx.006CH	D[15:0] = LB_IO_BASE D[15:0] = 006CH (2 cycles)
Am2903x Am2904x	WRITE	0	0	0	0	xxxx.006CH	D[31:16] = LB_IO_BASE D[15:0] = 006CH

14.2.2.1 i960 Processor Configuration Note

In order for the V96xEPC to interact correctly with the i960 family of processors, the following processors parameters should be used:

- Burst - Enable
- External Ready - Enable
- Pipelining - Disable
- N_{RAD} - 0
- N_{RDD} - 0
- N_{WAD} - 0
- N_{WDD} - 0
- N_{XDA} - 0
- Bus Width - 32 Bits

14.2.3 Initialization Using the PCI Configuration Space

The EPC can be initialized from the PCI bus by a PCI system master via configuration space accesses. Typically, this type of configuration would be used in systems using the EPC as a target bridge for a PCI add-in card.

Access to the configuration space of the EPC from the PCI bus is achieved by asserting the EPC's IDSEL pin. With IDSEL active, PCI transfers in the address range 0-FFH will be forwarded to the internal configuration registers. The EPC ignores Type 1 configuration accesses (PCI to PCI bridge configuration cycles).

Initialization

Initializing the Internal Registers

14.2.4 Initialization Using the Serial EEPROM interface

The EPC registers can also be initialized using the serial EEPROM interface. When using this method, the EPC “downloads” the values for the first half of the 256-byte internal register block from a serial EEPROM connected to the SCL and SDA pins. All register bits of type “FRW” or “FR” for configuration registers from 00H to 7FH are written by the serial EEPROM. Only bits of type “R” cannot be written.

Serial EEPROM initialization is useful in the following situations:

- Systems without a local processor that are using the EPC as a stand alone bridge
- Systems that cannot guarantee that the EPC can be initialized in a timely fashion by the local processor
- Systems which wish to eliminate an additional variable (e.g. during software debug)

The EPC uses a I²C-like interface to download from the serial EEPROM. The interface is designed to work with 24C02 style serial EEPROMs. Table 23 shows serial EEPROMs known to be compatible with this interface.

When the reset input ($\overline{\text{LRESET}}$ or $\overline{\text{PRST}}$, as selected by RDIR) is de-asserted, the SDA pin is sampled at the rising edge of the reset input. If a serial PROM is present, the SDA pin will be pulled high by the external pull up resistor. The detection of a high signal on SDA at the rising edge of the reset input begins the serial download. Each byte is read from the EEPROM sequentially, starting from 0H and ending at 7FH.

NOTE: To prevent serial initialization for systems initializing from the PCI or local bus, YOU MUST TIE SDA TO GROUND through a 1-2.2K ohm resistor. See Figure 64 for example circuits.

Table 23: Serial EEPROMs Known Compatible with the EPC

Manufacturer	Device
Atmel	24C02
Microchip	24C02A
Xicor	X24C02

14.2.4.1 Programming the Serial EEPROM

Programming of the serial EEPROM is straightforward on most commercial EPROM programmers. Each byte in the 128-byte EEPROM is programmed with the value desired in the corresponding byte in the EPC’s register map. Unused registers should be programmed with 0H.

The EPC may be used to program the EEPROM in the target application. Direct control over the state of the SCL and SDA pins is available through the SYSTEM register. The system programmer must provide the proper signal timing for the serial EEPROM through software emulation of the I²C-like protocol. Examples of source code for programming the I²C protocol can be found on the World Wide Web by searching for I²C using one of the internet search engines.

ApNote: If the VENDOR ID and DEVICE ID fields are programmed to FFFFH (an illegal value) in the EEPROM, the EPC will ignore these values and leave the default power on values in these registers.

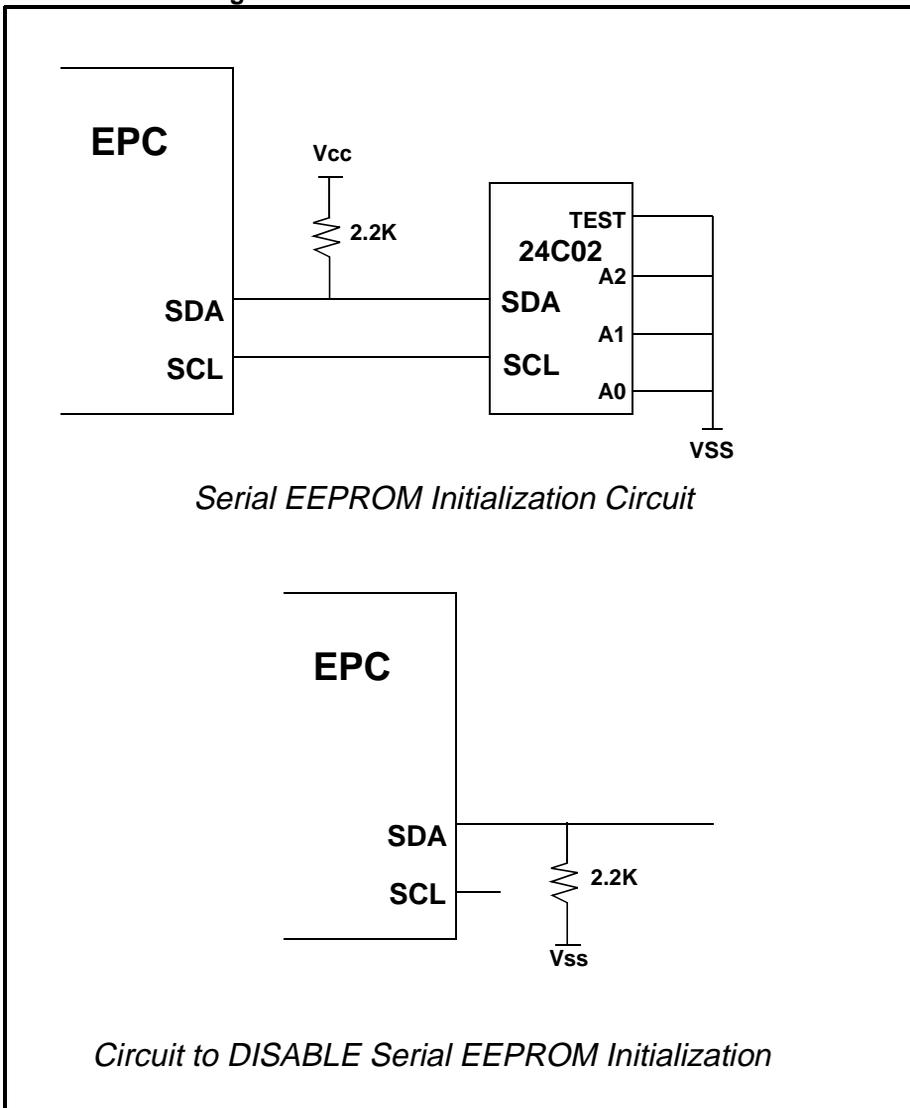
14.2.4.2 Timing Considerations when Initializing via the Serial EEPROM

The download of the serial EEPROM data takes a considerable amount of time. Each byte of data requires 9 SCL cycles at 512 PCI clocks per cycle. There is additional set-up overhead of approximately 5 data bytes (for serial EEPROM setup). The timing for a full download is:

$$(512 \text{ PCI clocks}) \times (133 \text{ bytes}) \times (9 \text{ SCL's per byte}) = 613,000 \text{ PCI clocks}$$

Access from PCI will cause Retry until the download is done.

Figure 64: Serial EEPROM Initialization Schematics



Initialization

Initializing the Internal Registers

14.2.5 Re-Initialization Using the PCI I/O or Memory Space

The EPC's internal registers may also be accessed via the PCI-to-Internal register aperture. The base address for this aperture is controlled by the PCI_IO_BASE register. This method can not be used to initialize the EPC immediately following a RESET (unless a serial EEPROM is used) because PCI memory reads and writes are ignored following RESET until one or both of the MEM_EN and IO_EN bits in the PCI_CMD register are set ('1') and the PCI_IO_BASE register base address has been established.

Chapter 15

Register Descriptions

The registers for the EPC are broken into 6 basic groups: System, PCI Configuration and Control, Local Configuration and Control, FIFO Configuration, DMA Control, and Mailbox Registers. This chapter will describe the location and attributes of the registers; programming details for each register are given in the appropriate chapters.

Each bit in the EPC registers is readable and writable according to one of the following designations. Those marked with an asterisk (*) apply to the PCI Configuration Registers only and comply with the PCI specification to provide the required PCI configuration header.

R: Read only - bits are internally driven and cannot be modified.

FR*: Firmware Initialized, Configuration Read Only - these bits are initialized after a system reset by downloading via the serial EEPROM device or by the local bus master. Once "FR" bits are loaded they may be locked from further modification by setting the LOCK bit in the SYSTEM register.

W: Write only. Typically used to issue commands.

FRW*: Firmware Initialized, Configuration Read/Write - Initialized at boot-time but can be both read or written from the PCI and Local buses.

RW: Read and Write.

All reserved register bits read back as zeros.

The PCI configuration registers required by the PCI Spec are described in the PCI Configuration Registers section below.

15.1 REGISTER MAP

Figure 65 shows the internal register map for the EPC. The offsets shown are relative to the base address of the aperture from which the registers are accessed:

- LB_IO_BASE for accesses from the local bus
- PCI_IO_BASE for memory or I/O accesses from the PCI bus
- 0H in configuration space for configuration accesses from the PCI bus

Register Descriptions

Register Map

Figure 65: Register Map

REGISTER				OFFSET
31	1615			0
PCI_DEVICE		PCI_VENDOR		00H
PCI_STAT		PCI_CMD		04H
PCI_CC_REV				08H
PCI_HDR_CFG				0CH
PCI_IO_BASE (PCI_I20_BASE when I20 operation is enabled: I20_EN bit)				10H
PCI_BASE0				14H
PCI_BASE1				18H
reserved				1C-2BH
PCI_SUB_ID		PCI_SUB_VENDOR		2CH
PCI-ROM				30H
reserved				34-38H
PCI_BPARAM				3CH
PCI_MAP0				40H
PCI_MAP1 (PCI_I20_MAP ^a when I20 operation is enabled: I20_EN bit)				44H
PCI_INT_STAT				48H
PCI_INT_CFG				4CH
reserved				50H
LB_BASE0				54H
LB_BASE1				58H
LB_MAP0		reserved		5CH
LB_MAP1		reserved		60H
LB_MAP2		LB_BASE2		64H
LB_SIZE ^b				68H
LB_IO_BASE		reserved		6CH
FIFO_PRIORITY		FIFO_CFG		70H
LB_IMASK	LB_I_STAT		FIFO_STAT	74H
LB_CFG		SYSTEM		78H
reserved		PCI_CFG		7CH
DMA_PCI_ADDR0				80H
DMA_LOCAL_ADDR0				84H
DMA_CSR0	DMA_LENGTH0			88H
DMA_CTLB_ADR0				8CH
DMA_PCI_ADDR1				90H
DMA_LOCAL_ADDR1				94H
DMA_CSR1	DMA_LENGTH1			98H
DMA_CTLB_ADR1				9CH
I2O Message Unit Pointers ^b				A0H - BCH
MAIL_DATA3	MAIL_DATA2	MAIL_DATA1	MAIL_DATA0	C0H
MAIL_DATA7	MAIL_DATA6	MAIL_DATA5	MAIL_DATA4	C4H
MAIL_DATA11	MAIL_DATA10	MAIL_DATA9	MAIL_DATA8	C8H
MAIL_DATA15	MAIL_DATA14	MAIL_DATA13	MAIL_DATA12	CCH
PCI_MAIL_IERD		PCI_MAIL_IEWR		D0H
LB_MAIL_IERD		LB_MAIL_IEWR		D4H
MAIL_RD_STAT		MAIL_WR_STAT		D8H
QBA_MAP				DCH
			DMA_DELAY	E0H

PCI_VENDOR: VENDOR ID (PCI REQUIRED)

Mnemonic: PCI_VENDOR
 Offset: 00H
 Size: 16 bits

PCI_VENDOR				
Bits	Mnemonic	Type	Reset Value	Description
15-0	VENDOR	FR	11B0H	Vendor ID. This register identifies the vendor of the device.

PCI_DEVICE: DEVICE ID (PCI REQUIRED)

Mnemonic: PCI_DEVICE
 Offset: 02H
 Size: 16 bits

PCI_DEVICE				
Bits	Mnemonic	Type	Reset Value	Description
15-0	DEVICE	FR	see text	Device ID. This register identifies the ID of the device. At reset it reads back a value dependent on the type of EPC device: 960mode=01h, 961mode=02h, 962mode=04h, 292mode=10h

PCI_CMD: COMMAND REGISTER (PCI REQUIRED)

Mnemonic: PCI_CMD
 Offset: 04H
 Size: 16 bits

PCI_CMD				
Bits	Mnemonic	Type	Reset Value	Description
15-10	-	R	0H	reserved
9	FBB_EN	FRW	0H	Fast Back-to-Back Enable 1 = EPC will perform fast back-to-back transfers when bus master 0 = EPC will not perform fast back-to-back transfers
8	SERR_EN	FRW	0H	System Error Enable 1 = System error enabled: If PAR_EN (bit 6) is also enabled then SERR is driven in response to an address parity error. 0 = System error disabled: SERR is not driven.
7	-	R	0H	reserved
6	PAR_EN	FRW	0H	Parity Error Enable: 1 = EPC will report PCI parity errors 0 = EPC will ignore PCI parity errors
5	-	R	0H	reserved
4	-	R	0H	reserved

Register Descriptions

Register Map

PCI_CMD (cont'd)				
Bits	Mnemonic	Type	Reset Value	Description
3	-	R	0H	reserved
2	MASTER_EN	FRW	0H	PCI Master Enable: 1 = EPC will act as PCI bus master (i.e. assert $\overline{\text{REQ}}$) 0 = EPC will not act as PCI bus master ^a
1	MEM_EN	FRW	0H	Memory Access Enable 1 = EPC will respond to memory accesses on the PCI bus 0 = EPC will ignore ALL memory accesses on the PCI bus
0	IO_EN	FRW	0H	I/O Access Enable 1 = EPC will respond to IO accesses on the PCI bus 0 = EPC will ignore ALL IO accesses on the PCI bus

a. Clearing this bit effectively prohibits any local bus reads/writes to PCI space. If PCI bus mastering is disabled, all local bus writes to PCI space, and all DMA transfers destined for PCI space, will be queued in the Local-to-PCI FIFO until either this bit is set, or the FIFO is full.

PCI_STAT: PCI STATUS REGISTER

Mnemonic: PCI_STAT
Offset: 06H
Size: 16 bits

PCI_STAT				
Bits	Mnemonic	Type	Reset Value	Description
15	PAR_ERR	FRW	0H	Parity Error: set (1) in response to a parity error being detected on the PCI bus. Cleared by writing '1' to this bit.
14	SYS_ERR	FRW	0H	System Error: set (1) in response to a system error being detected by this device and reported on the SERR pin on the PCI bus. Cleared by writing '1' to this bit.
13	M_ABORT	FRW	0H	Master Abort: set (1) in response to a master abort being detected during transaction in which the EPC was acting as a bus master. Cleared by writing a '1' to this bit.
12	T_ABORT	FRW	0H	Target Abort: set (1) in response to a target abort being detected during transaction in which the EPC was acting as a bus master. Cleared by writing '1' to this bit.
11	-	R	0H	reserved
10-9	DEVSEL	FR	0H	Device Select Timing: Programmable during initialization for the benefit of other PCI bus masters. Doesn't affect the operation of the EPC.
8	PAR_REP	FRW	0H	Data Parity Error Report: set (1) whenever the EPC acts as a bus master and observes the $\overline{\text{PERR}}$ signal being driven. The PAR_EN bit in PCI_CMD must also be enabled for this bit to be set. Cleared by writing '1' to the bit.
7	FAST_BACK	FR	0H	Fast Back-to-Back Target Enable: Used to indicate to other bus masters the ability of this device to respond to fast back-to-back transfers. Note: The state of this bit will not effect the internal operation of the EPC and it will always respond properly to fast back-to-back transfers.
6-0	-	R	0H	reserved

PCI_CC_REV: PCI CLASS AND REVISION REGISTER (PCI REQUIRED)

Mnemonic: PCI_CC_REV
 Offset: 08h
 Size: 32 bits

PCI_CC_REV				
Bits	Mnemonic	Type	Reset Value	Description
31-24	BASE_CLASS	FR	0H ^a	PCI Base Class Code (see PCI specification)
23-16	SUB_CLASS	FR	0H	PCI Sub Class Code (see PCI specification)
15-8	PROG_IF	FR	0H	PCI Programming Interface Code (see PCI specification)
7-4	UREV	FR	0H	User Revision ID. These bits are programmable to indicate the revision level of the system built with the EPC.
3-0	VREV	R	Stepping ID	V3 Revision. These bits are hardwired to reflect the revision number of the component. Rev A0=4h

a. 6H is the base class for a host to PCI bridge. This can be programmed to suit the application.

PCI_HDR_CFG: PCI HEADER/CONFIG. REGISTER (PCI REQUIRED)

Mnemonic: PCI_HDR_CFG
 Offset: 0CH
 Size: 32 bits

PCI_HDR_CFG				
Bits	Mnemonic	Type	Reset Value	Description
31-24	BIST	R	0H	Built in Self Test: Unimplemented; reads back as all "0"
23-16	HDR_TYPE	R	0H	Header Type: Unimplemented; reads back as all "0"
15-11	LT	FRW	0H	Latency Timer: Latency value in multiples of 8 clocks.
10-8	LTL	R	0H	Latency Timer: Unimplemented lower bits of the PCI latency timer register
7-0	LINE_SIZE	FRW	0H	Cache Line Size: Has no effect on the internal operation of the EPC

Register Descriptions

Register Map

PCI_I2O_BASE : PCI I₂O BASE ADDRESS REGISTER¹

Mnemonic: PCI_I2O_BASE
 Offset: 10H
 Size: 32 bits

PCI_I2O_BASE				
Bits	Mnemonic	Type	Reset Value	Description
31-20	ADR_BASE	FRW	0H	Base Address: If the value of ADR_BASE matches that of AD[31:20] during the address phase of a PCI access then a match is detected. Since bits 31-20 are significant to the decoder, the size of the aperture is 1MB. This size can be increased using the corresponding ADR_SIZE register bits so that lower bits of the decode are masked off.
19-4	-	R	0H	reserved
3	PREFETCH	FR	0H	Prefetchable: enables prefetching of data for read cycles by anticipating sequential reads.
2-1	TYPE = "00"	R	0H	Address Range Type: These read only bits are hardwired to "00" to indicate that the device can be mapped anywhere in the 32 bit address space.
0	IO	FR	0H	'1' = access through I/O space. '0' = access from memory space.

PCI_IO_BASE: PCI ACCESS TO INTERNAL EPC REGISTERS

Mnemonic: PCI_IO_BASE
 Offset: 10H, 14H²
 Size: 32 bits

PCI_IO_BASE				
Bits	Mnemonic	Type	Reset Value	Description
31-8	ADR_BASE	FRW	0H	Base Address: If the value of ADR_BASE matches that of AD[31:8] during the address phase of a PCI access then a match is detected. Since bits 31-8 are significant to the decoder, the size of the PCI-to-Internal register aperture is 256 bytes.
7-4	-	R	0H	reserved (Zero)
3	PREFETCH	R	0H	Prefetchable. This bit is for configuration information only and has no affect on the operation of the EPC. Accesses to internal registers are never prefetched.
2-1	TYPE	R	0H	Address Range Type: These read only bits are hardwired to "00" to indicate that the device can be mapped anywhere in the 32 bit address space.
0	IO	FR	0H	1 = The PCI-to-Internal register aperture is in PCI I/O space 0 = The PCI-to-Internal register aperture is in PCI memory space

1. Only available when I2O mode is enabled.

2. Located at 10H when I2O mode is disabled. When I2O is enabled, it is located at 14H.

PCI_BASE0: PCI TO LOCAL BUS APERTURE 0 BASE ADDRESS¹

Mnemonic: PCI_BASE0
 Offset: 14H
 Size: 32 bits

PCI_BASE0				
Bits	Mnemonic	Type	Reset Value	Description
31-20	ADR_BASE	FRW	0H	Base Address: If the value of ADR_BASE matches that of AD[31:20] during the address phase of a PCI access then a match is detected. A larger address space can be decoded by changing the ADR_SIZE field in the PCI_MAP0 register. This will mask off some of the lower bits of this field to allow automatic configuration software to determine the size of the aperture.
19-8	ADR_BASEL	FRW	0H	Low order base address bits used for fine grain I/O decode only. These bits are only used when IO=1 and ADR_SIZE is set to 0100-0111 in the PCI_MAP0 register.
7-4	-	R	0H	reserved
3	PREFETCH	FR	0H	Prefetchable: 1 = Enable read prefetching for this aperture 0 = Disable read prefetching for this aperture When LOCK is disabled and both the IO and PREFETCH bits are written to '1', the PREFETCH bit will read '0' even though it is internally set to '1' and the aperture will exhibit prefetch behavior.
2-1	TYPE	R	0H	Address Range Type: These read only bits are hardwired to "00" to indicate that the device can be mapped anywhere in the 32 bit address space.
0	IO	FR	0H	1 = The PCI-to-Local aperture 0 will respond to PCI IO space access (CBE=2h, 3h) 0 = The PCI-to-Local aperture 0 will respond to PCI memory space access (CBE=6h, 7h, Ch, Eh, Fh)

1. Only available when I2O mode is disabled.

Register Descriptions

Register Map

PCI_BASE1: PCI TO LOCAL BUS APERTURE 1 BASE ADDRESS¹

Mnemonic: PCI_BASE1
 Offset: 18H
 Size: 32 bits

PCI_BASE1				
Bits	Mnemonic	Type	Reset Value	Description
31-20	ADR_BASE	FRW	0H	Base Address: If the value of ADR_BASE matches that of AD[31:20] during the address phase of a PCI access then a match is detected. In legacy DOS mode the address comparison has increased granularity (see DOS Compatibility chapter).
19-14	ADR_BASEL	FRW R	0H	Base address bits used only for DOS compatibility mode. See PC Compatibility chapter. These bits read back as '0' unless DOS mode is selected in the PCI_MAP1 register.
13-11	-	R	0H	reserved
10-8	DOS_MEM	FRW R	0H	DOS Mode Memory Size: When IO=0 and DOS mode is selected, these bits set the size of the real mode DOS memory hole: 100 = 16K bytes (A[31:14]) 101 = 32K bytes (A[31:15]) 110 = 64K bytes (A[31:16]) 111 = 128K bytes (A[31:17]) others = disabled These bits read back as '0' unless DOS mode is selected in the PCI_MAP1 register.
7-4	-	R	0H	reserved
3	PREFETCH	FR	0H	Prefetchable: 1 = Enable read prefetching for this aperture 0 = Disable read prefetching for this aperture When LOCK is disabled and both the IO and PREFETCH bits are written to '1', the PREFETCH bit will read '0' even though it is internally set to '1' and the aperture will exhibit prefetch behavior.
2-1	TYPE	R	0H	Address Range Type: These read only bits are hardwired to "00" to indicate that the device can be mapped anywhere in the 32 bit address space.
0	IO	FR	0H	1 = The PCI-to-Local aperture 0 is in PCI IO space 0 = The PCI-to-Local aperture 0 is in PCI memory space

PCI_SUB_VENDOR: PCI SUBSYSTEM VENDOR

Mnemonic: PCI_SUB_VENDOR
 Offset: 2CH
 Size: 16 bits

PCI_SUB_VENDOR				
Bits	Mnemonic	Type	Reset Value	Description
15-0	VENDOR	FRW	0H	Subsystem Vendor ID: firmware programmable (PCI 2.1 and Windows95® required)

1. Only available when I2O mode is disabled.

PCI_SUB_VENDOR: PCI SUBSYSTEM ID

Mnemonic: PCI_SUB_ID
 Offset: 2EH
 Size: 16 bits

PCI_SUB_ID				
Bits	Mnemonic	Type	Reset Value	Description
15-0	ID	FRW	0H	Subsystem ID: firmware programmable (PCI 2.1 and Windows95® required)

PCI_ROM: EXPANSION ROM BASE

Mnemonic: PCI_ROM
 Offset: 30H
 Size: 32 bits

PCI_ROM				
Bits	Mnemonic	Type	Reset Value	Description
31-12	ROM_BASE	FRW	0H	Expansion ROM Base Address. The size of the ROM aperture is controlled in PCI_MAP0 and the decoders are shared between these apertures.
11-1	-	R	0H	reserved
0	ENABLE	FRW	0H	Expansion ROM Enable. 1 = enabled, 0 = disabled

PCI_BPARAM: PCI BUS PARAMETER REGISTER (PCI REQUIRED)

Mnemonic: PCI_BPARAM
 Offset: 3CH
 Size: 32 bits

PCI_BPARAM				
Bits	Mnemonic	Type	Reset Value	Description
31-24	MAX_LAT	FR	0H	Maximum Latency: For PCI autoconfiguration reporting only. Has no effect on the internal operation of the EPC
23-16	MIN_GNT	FR	0H	Minimum Grant: For PCI autoconfiguration reporting only. Has no effect on the internal operation of the EPC
15-11	-	R	0H	reserved
10-8	INT_PIN	FR*	0H	Interrupt Pin: Selects which interrupt pin will be driven to the PCI bus (PCI Spec). Chooses which INTx pin will receive internal EPC interrupts (see "Interrupt Control" chapter). 000 = Disabled 001 = Use <u>INTA</u> 010 = Use <u>INTB</u> 011 = Use <u>INTC</u> 100 = Use <u>INTD</u>
7-0	INT_LINE	FR*	0H	Interrupt Line: For PCI autoconfiguration reporting only. Has no effect on the internal operation of the EPC

Register Descriptions

Register Map

PCI_MAP0: PCI BUS TO LOCAL BUS ADDRESS MAP 0

Mnemonic: PCI_MAP0
 Offset: 40H
 Size: 32 bits

PCI_MAP0																																		
Bits	Mnemonic	Type	Reset Value	Description																														
31-20	MAP_ADR	FRW	0H	Map Address: These bits correspond to bits LAD[31:20] in local address space when a PCI to Local access is made. The lower bits of MAP_ADR are masked off according to the ADR_SIZE bits in the PCI_MAP registers.																														
19-16	-	R	0H	reserved																														
15	RD_POST_INH	FRW	0H	Read Posting Inhibit: When set '1' the very first read of a burst read from the corresponding aperture will not generate a <u>STOP</u> regardless of the latency of the access.																														
14-12	-	R	0H	reserved																														
11-10	ROM_SIZE	FRW	0H	ROM Size: Determines the size of the expansion ROM address decoder: 00 = expansion ROM base register disabled 01 = 4K byte expansion ROM (A[31:12] significant) 10 = 16K byte expansion ROM (A[31:14] significant) 11 = 64K byte expansion ROM (A[31:16] significant)																														
9-8	SWAP	FRW	0H	Byte Swap Control: Selects byte lane swapping for read and write cycles according to the following table: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th></th> <th>SWAP</th> <th>D[31:24]</th> <th>D[23:16]</th> <th>D[15:8]</th> <th>D[7:0]</th> </tr> </thead> <tbody> <tr> <td>no swap, 32 bit</td> <td>00</td> <td>Q[31:24]</td> <td>Q[23:16]</td> <td>Q[15:8]</td> <td>Q[7:0]</td> </tr> <tr> <td>16 bit</td> <td>01</td> <td>Q[15:8]</td> <td>Q[7:0]</td> <td>Q[31:24]</td> <td>Q[23:16]</td> </tr> <tr> <td>8 bit</td> <td>10</td> <td>Q[7:0]</td> <td>Q[15:8]</td> <td>Q[23:16]</td> <td>Q[31:24]</td> </tr> <tr> <td></td> <td>11</td> <td colspan="4" style="text-align: center;">Auto Swap</td> </tr> </tbody> </table> <p>Auto Swap: When local bus $\overline{BE}[3:0] = "1100"$ or $"0011"$ then a 16 bit swap is done. When local bus $\overline{BE}[3:0] = "1110"$, $"1101"$, $"1011"$ or $"0111"$ then an 8 bit swap is done. Any other combination results in non-swapped data.</p>		SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]	no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]	16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]	8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]		11	Auto Swap			
	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]																													
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]																													
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]																													
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]																													
	11	Auto Swap																																

PCI_MAP0 (cont'd)																																														
Bits	Mnemonic	Type	Reset Value	Description																																										
7-4	ADR_SIZE	FRW	0H	<p>Aperture Size: Legacy DOS mode uses a different decoding scheme described in the "PC Compatibility" chapter of the manual.</p> <table border="1"> <thead> <tr> <th>ADDR_SIZE</th> <th>Size</th> <th>Valid ADR BASE Bits</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>1MB</td> <td>31:20</td> </tr> <tr> <td>0001</td> <td>2MB</td> <td>31:21</td> </tr> <tr> <td>0010</td> <td>4MB</td> <td>31:22</td> </tr> <tr> <td>0011</td> <td>8MB</td> <td>31:23</td> </tr> <tr> <td>0100</td> <td>16MB memory 256 byte I/O</td> <td>31:24 (mem) 31:8 (IO)</td> </tr> <tr> <td>0101</td> <td>32MB memory 512 byte I/O</td> <td>31:25 (mem) 31:9 (IO)</td> </tr> <tr> <td>0110</td> <td>64MB memory 1024 byte I/O</td> <td>31:26 (mem) 31:10 (IO)</td> </tr> <tr> <td>0111</td> <td>128MB memory 2048 byte I/O</td> <td>31:27 (mem) 31:11 (IO)</td> </tr> <tr> <td>1000</td> <td>256MB</td> <td>31:28</td> </tr> <tr> <td>1001</td> <td>512MB</td> <td>31:29</td> </tr> <tr> <td>1010</td> <td>1GB</td> <td>31:30</td> </tr> <tr> <td>11xx</td> <td>1MB DOS Mode (PCI_MAP1 only)</td> <td>31:20</td> </tr> <tr> <td>others</td> <td>-</td> <td>reserved</td> </tr> </tbody> </table>	ADDR_SIZE	Size	Valid ADR BASE Bits	0000	1MB	31:20	0001	2MB	31:21	0010	4MB	31:22	0011	8MB	31:23	0100	16MB memory 256 byte I/O	31:24 (mem) 31:8 (IO)	0101	32MB memory 512 byte I/O	31:25 (mem) 31:9 (IO)	0110	64MB memory 1024 byte I/O	31:26 (mem) 31:10 (IO)	0111	128MB memory 2048 byte I/O	31:27 (mem) 31:11 (IO)	1000	256MB	31:28	1001	512MB	31:29	1010	1GB	31:30	11xx	1MB DOS Mode (PCI_MAP1 only)	31:20	others	-	reserved
ADDR_SIZE	Size	Valid ADR BASE Bits																																												
0000	1MB	31:20																																												
0001	2MB	31:21																																												
0010	4MB	31:22																																												
0011	8MB	31:23																																												
0100	16MB memory 256 byte I/O	31:24 (mem) 31:8 (IO)																																												
0101	32MB memory 512 byte I/O	31:25 (mem) 31:9 (IO)																																												
0110	64MB memory 1024 byte I/O	31:26 (mem) 31:10 (IO)																																												
0111	128MB memory 2048 byte I/O	31:27 (mem) 31:11 (IO)																																												
1000	256MB	31:28																																												
1001	512MB	31:29																																												
1010	1GB	31:30																																												
11xx	1MB DOS Mode (PCI_MAP1 only)	31:20																																												
others	-	reserved																																												
3-2	-	R	0H	reserved																																										
1	REG_EN ^a	FRW	0H	PCI_BASE0 register enable. 1 = PCI_BASE0 enabled, 0=PCI_BASE0 disabled (reads back as 0H).																																										
0	ENABLE ^a	FRW	0H	PCI_BASE0 Aperture Enable. 1 = PCI-to-Local aperture 0 is enabled, 0 = PCI-to-Local aperture 0 is disabled.																																										

a. Must be written '0' when I2O mode is enabled

Register Descriptions

Register Map

PCI_MAP1: PCI BUS TO LOCAL BUS ADDRESS MAP 1¹

Mnemonic: PCI_MAP1
 Offset: 44H
 Size: 32 bits

PCI_MAP1																																		
Bits	Mnemonic	Type	Reset Value	Description																														
31-20	MAP_ADR	FRW	0H	Map Address: These bits correspond to bits LAD(31:20) in local address space when a PCI to Local access is made. The lower bits of MAP_ADR are masked off according to the ADR_SIZE bits in the PCI_MAP registers.																														
19-16	-	R	0H	reserved																														
15	RD_POST_INH	FRW	0H	Read Posting Inhibit: When set '1' the very first read of a burst read from the corresponding aperture will not generate a <u>STOP</u> regardless of the latency of the access.																														
14-10	-	R	0H	reserved																														
9-8	SWAP	FRW	0H	<p>Byte Swap Control: Selects byte lane swapping for read and write cycles according to the following table:</p> <table border="1"> <thead> <tr> <th></th> <th>SWAP</th> <th>D[31:24]</th> <th>D[23:16]</th> <th>D[15:8]</th> <th>D[7:0]</th> </tr> </thead> <tbody> <tr> <td>no swap, 32 bit</td> <td>00</td> <td>Q[31:24]</td> <td>Q[23:16]</td> <td>Q[15:8]</td> <td>Q[7:0]</td> </tr> <tr> <td>16 bit</td> <td>01</td> <td>Q[15:8]</td> <td>Q[7:0]</td> <td>Q[31:24]</td> <td>Q[23:16]</td> </tr> <tr> <td>8 bit</td> <td>10</td> <td>Q[7:0]</td> <td>Q[15:8]</td> <td>Q[23:16]</td> <td>Q[31:24]</td> </tr> <tr> <td></td> <td>11</td> <td colspan="4">Auto Swap</td> </tr> </tbody> </table> <p>Auto Swap: When local bus $\overline{BE}[3:0] = "1100"$ or $"0011"$ then a 16 bit swap is done. When local bus $\overline{BE}[3:0] = "1110"$, $"1101"$, $"1011"$ or $"0111"$ then an 8 bit swap is done. Any other combination results in non-swapped data.</p>		SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]	no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]	16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]	8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]		11	Auto Swap			
	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]																													
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]																													
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]																													
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]																													
	11	Auto Swap																																

1. Only available when I2O mode is disabled.

PCI_MAP1 (Cont'd)																																														
Bits	Mnemonic	Type	Reset Value	Description																																										
7-4	ADR_SIZE	FRW	0H	<p>Aperture Size: Legacy DOS mode uses a different decoding scheme described in the "PC Compatibility" chapter of the manual.</p> <table border="1"> <thead> <tr> <th>ADDR_SIZE</th> <th>Size</th> <th>Valid ADR BASE Bits</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>1MB</td> <td>31:20</td> </tr> <tr> <td>0001</td> <td>2MB</td> <td>31:21</td> </tr> <tr> <td>0010</td> <td>4MB</td> <td>31:22</td> </tr> <tr> <td>0011</td> <td>8MB</td> <td>31:23</td> </tr> <tr> <td>0100</td> <td>16MB memory 256 byte I/O</td> <td>31:24 (mem) 31:8 (IO)</td> </tr> <tr> <td>0101</td> <td>32MB memory 512 byte I/O</td> <td>31:25 (mem) 31:9 (IO)</td> </tr> <tr> <td>0110</td> <td>64MB memory 1024 byte I/O</td> <td>31:26 (mem) 31:10 (IO)</td> </tr> <tr> <td>0111</td> <td>128MB memory 2048 byte I/O</td> <td>31:27 (mem) 31:11 (IO)</td> </tr> <tr> <td>1000</td> <td>256MB</td> <td>31:28</td> </tr> <tr> <td>1001</td> <td>512MB</td> <td>31:29</td> </tr> <tr> <td>1010</td> <td>1GB</td> <td>31:30</td> </tr> <tr> <td>11xx</td> <td>1MB DOS Mode (PCI_MAP1 only)</td> <td>31:20</td> </tr> <tr> <td>others</td> <td>-</td> <td>reserved</td> </tr> </tbody> </table>	ADDR_SIZE	Size	Valid ADR BASE Bits	0000	1MB	31:20	0001	2MB	31:21	0010	4MB	31:22	0011	8MB	31:23	0100	16MB memory 256 byte I/O	31:24 (mem) 31:8 (IO)	0101	32MB memory 512 byte I/O	31:25 (mem) 31:9 (IO)	0110	64MB memory 1024 byte I/O	31:26 (mem) 31:10 (IO)	0111	128MB memory 2048 byte I/O	31:27 (mem) 31:11 (IO)	1000	256MB	31:28	1001	512MB	31:29	1010	1GB	31:30	11xx	1MB DOS Mode (PCI_MAP1 only)	31:20	others	-	reserved
ADDR_SIZE	Size	Valid ADR BASE Bits																																												
0000	1MB	31:20																																												
0001	2MB	31:21																																												
0010	4MB	31:22																																												
0011	8MB	31:23																																												
0100	16MB memory 256 byte I/O	31:24 (mem) 31:8 (IO)																																												
0101	32MB memory 512 byte I/O	31:25 (mem) 31:9 (IO)																																												
0110	64MB memory 1024 byte I/O	31:26 (mem) 31:10 (IO)																																												
0111	128MB memory 2048 byte I/O	31:27 (mem) 31:11 (IO)																																												
1000	256MB	31:28																																												
1001	512MB	31:29																																												
1010	1GB	31:30																																												
11xx	1MB DOS Mode (PCI_MAP1 only)	31:20																																												
others	-	reserved																																												
3-2	-	R	0H	reserved																																										
1	REG_EN	FRW	0H	PCI_BASE1 register enable. 1 = PCI_BASE1 enabled, 0=PCI_BASE1 disabled (reads back as 0H from PCI and Local).																																										
0	ENABLE	FRW	0H	PCI_BASE1 Aperture Enable. 1 = PCI-to-Local aperture 1 is enabled, 0 = PCI-to-Local aperture 1 is disabled.																																										

Register Descriptions

Register Map

PCI_I2O_MAP: PCI BUS I₂O ATU LOCAL BUS ADDRESS MAP¹

Mnemonic: PCI_I2O_MAP
 Offset: 44H
 Size: 32 bits

PCI_I2O_MAP																																														
Bits	Mnemonic	Type	Reset Value	Description																																										
31-20	MAP_ADR	FRW	0H	Map Address: These bits correspond to bits LAD(31:20) in local address space when a PCI to Local access is made. Address bits LAD(19:2) are derived from the PCI bus itself (for a 1MB aperture size). If the size of the aperture is increased, then the lower bits of MAP_ADR become masked off according to the ADR_SIZE bits in the PCI_MAP registers.																																										
19-16	-	R	0H	reserved																																										
15	RD_POST_DIS	FR	0H	Read Post Disable: When set '1' the very first read of a burst read from the corresponding aperture will not generate a STOP#.																																										
14-10	-	R	0H	reserved																																										
9-8	SWAP	FRW	0H	<p>Byte Swap Control: Selects byte lane swapping for read and write cycles according to the following table:</p> <table border="1"> <thead> <tr> <th></th> <th>SWAP</th> <th>D[31:24]</th> <th>D[23:16]</th> <th>D[15:8]</th> <th>D[7:0]</th> </tr> </thead> <tbody> <tr> <td>no swap, 32 bit</td> <td>00</td> <td>Q[31:24]</td> <td>Q[23:16]</td> <td>Q[15:8]</td> <td>Q[7:0]</td> </tr> <tr> <td>16 bit</td> <td>01</td> <td>Q[15:8]</td> <td>Q[7:0]</td> <td>Q[31:24]</td> <td>Q[23:16]</td> </tr> <tr> <td>8 bit</td> <td>10</td> <td>Q[7:0]</td> <td>Q[15:8]</td> <td>Q[23:16]</td> <td>Q[31:24]</td> </tr> <tr> <td></td> <td>11</td> <td colspan="4">Auto Swap</td> </tr> </tbody> </table> <p>Auto Swap: When local bus $\overline{BE}[3:0] = "1100"$ or $"0011"$ then a 16 bit swap is done. When local bus $\overline{BE}[3:0] = "1110"$, $"1101"$, $"1011"$ or $"0111"$ then an 8 bit swap is done. Any other combination results in non-swapped data.</p>		SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]	no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]	16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]	8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]		11	Auto Swap															
	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]																																									
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]																																									
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]																																									
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]																																									
	11	Auto Swap																																												
7-4	ADR_SIZE	FRW	0H	<p>Aperture Size: Legacy DOS mode uses a different decoding scheme described in the "PC Compatibility" chapter of the manual.</p> <table border="1"> <thead> <tr> <th>ADDR_SIZE</th> <th>Size</th> <th>Valid ADR BASE Bits</th> </tr> </thead> <tbody> <tr> <td>0000</td> <td>1MB</td> <td>31:20</td> </tr> <tr> <td>0001</td> <td>2MB</td> <td>31:21</td> </tr> <tr> <td>0010</td> <td>4MB</td> <td>31:22</td> </tr> <tr> <td>0011</td> <td>8MB</td> <td>31:23</td> </tr> <tr> <td>0100</td> <td>16MB memory 256 byte I/O</td> <td>31:24 (mem) 31:8 (IO)</td> </tr> <tr> <td>0101</td> <td>32MB memory 512 byte I/O</td> <td>31:25 (mem) 31:9 (IO)</td> </tr> <tr> <td>0110</td> <td>64MB memory 1024 byte I/O</td> <td>31:26 (mem) 31:10 (IO)</td> </tr> <tr> <td>0111</td> <td>128MB memory 2048 byte I/O</td> <td>31:27 (mem) 31:11 (IO)</td> </tr> <tr> <td>1000</td> <td>256MB</td> <td>31:28</td> </tr> <tr> <td>1001</td> <td>512MB</td> <td>31:29</td> </tr> <tr> <td>1010</td> <td>1GB</td> <td>31:30</td> </tr> <tr> <td>11xx</td> <td>1MB DOS Mode (PCI_MAP1 only)</td> <td>31:20</td> </tr> <tr> <td>others</td> <td>-</td> <td>reserved</td> </tr> </tbody> </table>	ADDR_SIZE	Size	Valid ADR BASE Bits	0000	1MB	31:20	0001	2MB	31:21	0010	4MB	31:22	0011	8MB	31:23	0100	16MB memory 256 byte I/O	31:24 (mem) 31:8 (IO)	0101	32MB memory 512 byte I/O	31:25 (mem) 31:9 (IO)	0110	64MB memory 1024 byte I/O	31:26 (mem) 31:10 (IO)	0111	128MB memory 2048 byte I/O	31:27 (mem) 31:11 (IO)	1000	256MB	31:28	1001	512MB	31:29	1010	1GB	31:30	11xx	1MB DOS Mode (PCI_MAP1 only)	31:20	others	-	reserved
ADDR_SIZE	Size	Valid ADR BASE Bits																																												
0000	1MB	31:20																																												
0001	2MB	31:21																																												
0010	4MB	31:22																																												
0011	8MB	31:23																																												
0100	16MB memory 256 byte I/O	31:24 (mem) 31:8 (IO)																																												
0101	32MB memory 512 byte I/O	31:25 (mem) 31:9 (IO)																																												
0110	64MB memory 1024 byte I/O	31:26 (mem) 31:10 (IO)																																												
0111	128MB memory 2048 byte I/O	31:27 (mem) 31:11 (IO)																																												
1000	256MB	31:28																																												
1001	512MB	31:29																																												
1010	1GB	31:30																																												
11xx	1MB DOS Mode (PCI_MAP1 only)	31:20																																												
others	-	reserved																																												
3-2	-	R	0H	reserved																																										

1. Only available when I2O mode is enabled

PCI_I2O_MAP (cont'd)				
Bits	Mnemonic	Type	Reset Value	Description
1	REG_EN	FRW	0H	PCI_BASE1 register enable. 1 = PCI_BASE1 enabled, 0=PCI_BASE1 disabled (reads back as 0H from PCI and Local).
0	ENABLE	FRW	0H	PCI_BASE1 Aperture Enable. 1 = PCI-to-Local aperture 1 is enabled, 0 = PCI-to-Local aperture 1 is disabled.

PCI_INT_STAT: PCI INTERRUPT STATUS REGISTER

Mnemonic: PCI_INT_STAT
 Offset: 48H
 Size: 32 bits

PCI_INT_STAT				
Bits	Mnemonic	Type	Reset Value	Description
31	MAILBOX	R	0H	Mailbox Interrupt: 1 = Mailbox (Doorbell) Interrupt request active 0 = No mailbox interrupts pending Cleared by clearing MAIL_RD_STAT and MAIL_WR_STAT
30	LOCAL	FRW	0H	Local Bus Direct Interrupt: 1 = Local bus master requests a PCI interrupt 0 = No operation This bit is set by writing '1' and cleared by writing '0'
29-28	-	R	0H	reserved
27	OUT_POST	FRW	0H	I2O Outbound Post List Not Empty: Indicates that the outbound post list head pointer not equals the tail pointer (OPL_HEAD'OPL_TAIL). This bit is equivalent to the PCI_I2O_I2STAT register bit 3 and can be read there also. It is masked off only when the I2O_EN bit in the PCI_CFG register is clear otherwise the not empty status will be readable here regardless of the mask bit in PCI_INT_CFG. This bit is also mapped into the PCI_I2O_I2STAT register bit 3 and can be read there also.
26	-	R	0H	reserved
25	DMA1	FRW	0H	DMA channel 1 Interrupt: 1 = DMA channel 1 has requested an interrupt 0 = DMA channel 1 has not requested an interrupt
24	DMA0	FRW	0H	DMA channel 0 Interrupt: 1 = DMA channel 0 has requested an interrupt 0 = DMA channel 0 has not requested an interrupt
23-15		R	0H	reserved
14	INTC_TO_D	FRW	0H	INTD Output from INTC Input: Set ('1') when enabled (INTC_INTD bit in the corresponding PCI_INT_CFG register field) and INTC is used as an input and an interrupt event has occurred on INTC
13	INTB_TO_D	FRW	0H	INTD Output from INTB Input: Set ('1') when enabled (INTB_INTD bit in the corresponding PCI_INT_CFG register field) and INTB is used as an input and an interrupt event has occurred on INTB

Register Descriptions

Register Map

PCI_INT_STAT (cont'd)				
Bits	Mnemonic	Type	Reset Value	Description
12	INTA_TO_D	FRW	0H	INTD Output from INTA Input: Set ('1') when enabled (INTA_INTD bit in the corresponding PCI_INT_CFG register field) and INTA is used as an input and an interrupt event has occurred on INTA
11	INTD_TO_C	FRW	0H	See description above for INTx_TO_y ^a
10	-	R	0H	reserved
9	INTB_TO_C	FRW	0H	See description above for INTx_TO_y
8	INTA_TO_C	FRW	0H	See description above for INTx_TO_y
7	INTD_TO_B	FRW	0H	See description above for INTx_TO_y
6	INTC_TO_B	FRW	0H	See description above for INTx_TO_y
5	-	R	0H	reserved
4	INTA_TO_B	FRW	0H	See description above for INTx_TO_y
3	INTD_TO_A	FRW	0H	See description above for INTx_TO_y
2	INTC_TO_A	FRW	0H	See description above for INTx_TO_y
1	INTB_TO_A	FRW	0H	See description above for INTx_TO_y
0	-	R	0H	reserved

a. All of the INTx_TO_y bits function identically with "x" being the source of the interrupt (PCI INTx) and "y" being the destination for the request (PCI INTy).

Note: LOCAL interrupt request is cleared by writing '0'; writing '1' has no effect.

All other writable status bits are cleared by writing '1'; writing '0' has no effect.

PCI_INT_CFG: PCI INTERRUPT CONFIGURATION REGISTER

Mnemonic: PCI_INT_CFG
 Offset: 4CH
 Size: 32 bits

PCI_INT_CFG				
Bits	Mnemonic	Type	Reset Value	Description
31	MAILBOX	FRW	0H	Mailbox Interrupt Enable: Enables a PCI interrupt from the mailbox unit. (see the Mailbox Registers chapter).
30	LOCAL	FRW	0H	Local Bus Direct Interrupt Enable: Enables direct local bus to PCI interrupts
29	MASTER_PI	FRW	0H	PCI Master Local Interrupt Enable: When enabled (1) together with the PCI_PERR bit in LB_IMASK (bit 3), a local bus interrupt will be generated whenever the VxxxEPC acts as a bus master and a parity error occurs.
28	SLAVE_PI	FRW	0H	PCI Slave Local Interrupt Enable: When enabled (1) together with the PCI_PERR bit in LB_IMASK (bit 3), a local bus interrupt will be generated whenever the VxxxEPC acts as a bus slave and a parity error occurs.
27	OUT_POST	R	0H	I2O Outbound Post List Not Empty: When Enabled ('1') the PCI interrupt pin (selected by the INT_PIN field of the PCI_BPARAM register) is asserted whenever the outbound post list head pointer not equals the tail pointer (OPL_HEAD ¹ OPL_TAIL). This bit is equivalent to the PCI_I2O_MASK register bit 3 and can be read/written there also.
26	-	R	0H	reserved

PCI_INT_CFG (cont'd)														
Bits	Mnemonic	Type	Reset Value	Description										
25	DMA1	FRW	0H	DMA Channel 1 interrupt enable										
24	DMA0	FRW	0H	DMA Channel 0 interrupt enable										
23-22	MODE_D	FRW	0H	INTD Interrupt Mode: Determines use of the corresponding interrupt pin. <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>MODE_D</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>Active low level triggered input</td> </tr> <tr> <td>01</td> <td>High-to-low edge triggered input</td> </tr> <tr> <td>10</td> <td>Software cleared output. INTD pin is asserted via an interrupt event and cleared through the PCI_INT_STAT register.</td> </tr> <tr> <td>11</td> <td>reserved</td> </tr> </tbody> </table>	MODE_D	Description	00	Active low level triggered input	01	High-to-low edge triggered input	10	Software cleared output. INTD pin is asserted via an interrupt event and cleared through the PCI_INT_STAT register.	11	reserved
MODE_D	Description													
00	Active low level triggered input													
01	High-to-low edge triggered input													
10	Software cleared output. INTD pin is asserted via an interrupt event and cleared through the PCI_INT_STAT register.													
11	reserved													
21-20	MODE_C	FRW	0H	INTC Interrupt Mode Note: See MODE_D description for bit settings.										
19-18	MODE_B	FRW	0H	INTB Interrupt Mode Note: See MODE_D description for bit settings.										
17-16	MODE_A	FRW	0H	INTA Interrupt Mode Note: See MODE_D description for bit settings.										
15	INTD_TO_LB	FRW	0H	1 = INTD will request local ICU interrupts when the input is active 0 = INTD will never request LICU interrupts										
14	INTC_TO_D	FRW	0H	1 = INTC will act as interrupt request for INTD output 0 = INTC will not act as interrupt request for INTD output										
13	INTB_TO_D	FRW	0H	1 = INTB will act as interrupt request for INTD output 0 = INTB will not act as interrupt request for INTD output										
12	INTA_TO_D	FRW	0H	1 = INTA will act as interrupt request for INTD output 0 = INTA will not act as interrupt request for INTD output										
11	INTD_TO_C	FRW	0H	1 = INTD will act as interrupt request for INTC output 0 = INTD will not act as interrupt request for INTC output										
10	INTC_TO_LB	FRW	0H	1 = INTC will request local ICU interrupts when the input is active 0 = INTC will never request LICU interrupts										
9	INTB_TO_C	FRW	0H	1 = INTB will act as interrupt request for INTC output 0 = INTB will not act as interrupt request for INTC output										
8	INTA_TO_C	FRW	0H	1 = INTA will act as interrupt request for INTC output 0 = INTA will not act as interrupt request for INTC output										
7	INTD_TO_B	FRW	0H	1 = INTD will act as interrupt request for INTB output 0 = INTD will not act as interrupt request for INTB output										
6	INTC_TO_B	FRW	0H	1 = INTC will act as interrupt request for INTB output 0 = INTC will not act as interrupt request for INTB output										
5	INTB_TO_LB	FRW	0H	1 = INTB will request local ICU interrupts when the input is active 0 = INTB will never request LICU interrupts										
4	INTA_TO_B	FRW	0H	1 = INTA will act as interrupt request for INTB output 0 = INTA will not act as interrupt request for INTB output										
3	INTD_TO_A	FRW	0H	1 = INTD will act as interrupt request for INTA output 0 = INTD will not act as interrupt request for INTA output										
2	INTC_TO_A	FRW	0H	1 = INTC will act as interrupt request for INTA output 0 = INTC will not act as interrupt request for INTA output										
1	INTB_TO_A	FRW	0H	1 = INTB will act as interrupt request for INTA output 0 = INTB will not act as interrupt request for INTA output										
0	INTA_TO_LB	FRW	0H	1 = INTA will request local ICU interrupts when the input is active 0 = INTA will never request LICU interrupts										

Register Descriptions

Register Map

LB_BASE 0,1 : LOCAL BUS TO PCI BUS APERTURE 0,1 ADDRESS

Mnemonic: LB_BASE0, 1
 Offset: 54H, 58H
 Size: 32 bits

LB_BASE0, 1																																														
Bits	Mnemonic	Type	Reset Value	Description																																										
31-20	ADR_BASE	FRW	0H	Base Address: If the value of ADR_BASE matches that of local address bits 31:20 during the address phase of a local access then a match is detected. Since bits 31-20 are significant to the decoder, the size of the aperture is 1MB. The aperture size can be increased using the corresponding ADR_SIZE register bits so that lower bits of the decode are masked off.																																										
19-10	-	R	0H	reserved																																										
9-8	SWAP	FRW	0H	<p>Byte Swap Control: Selects byte lane swapping for read and write cycles according to the following table:</p> <table border="1"> <thead> <tr> <th></th> <th>SWAP</th> <th>D[31:24]</th> <th>D[23:16]</th> <th>D[15:8]</th> <th>D[7:0]</th> </tr> </thead> <tbody> <tr> <td>no swap, 32 bit</td> <td>00</td> <td>Q[31:24]</td> <td>Q[23:16]</td> <td>Q[15:8]</td> <td>Q[7:0]</td> </tr> <tr> <td>16 bit</td> <td>01</td> <td>Q[15:8]</td> <td>Q[7:0]</td> <td>Q[31:24]</td> <td>Q[23:16]</td> </tr> <tr> <td>8 bit</td> <td>10</td> <td>Q[7:0]</td> <td>Q[15:8]</td> <td>Q[23:16]</td> <td>Q[31:24]</td> </tr> <tr> <td></td> <td>11</td> <td colspan="4">Auto Swap</td> </tr> </tbody> </table> <p>Auto Swap: When local bus $\overline{BE}[3:0] = "1100"$ or $"0011"$ then a 16 bit swap is done. When local bus $\overline{BE}[3:0] = "1110"$, $"1101"$, $"1011"$ or $"0111"$ then an 8 bit swap is done. Any other combination results in non-swapped data.</p>		SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]	no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]	16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]	8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]		11	Auto Swap															
	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]																																									
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]																																									
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]																																									
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]																																									
	11	Auto Swap																																												
7-4	ADR_SIZE	FRW	0H	<p>Aperture Size: The size of the aperture is determined as follows:</p> <table border="1"> <thead> <tr> <th>ADDR_SIZE</th> <th>Size</th> <th>Valid ADR BASE Bits</th> </tr> </thead> <tbody> <tr><td>0000</td><td>1MB</td><td>31:20</td></tr> <tr><td>0001</td><td>2MB</td><td>31:21</td></tr> <tr><td>0010</td><td>4MB</td><td>31:22</td></tr> <tr><td>0011</td><td>8MB</td><td>31:23</td></tr> <tr><td>0100</td><td>16MB</td><td>31:24</td></tr> <tr><td>0101</td><td>32MB</td><td>31:25</td></tr> <tr><td>0110</td><td>64MB</td><td>31:26</td></tr> <tr><td>0111</td><td>128MB</td><td>31:27</td></tr> <tr><td>1000</td><td>256MB</td><td>31:28</td></tr> <tr><td>1001</td><td>512MB</td><td>31:29</td></tr> <tr><td>1010^a</td><td>1GB</td><td>31:29</td></tr> <tr><td>1011^a</td><td>2GB</td><td>31:29</td></tr> <tr><td>others</td><td colspan="2">reserved</td></tr> </tbody> </table> <p>a.(The 1GB and 2GB aperture can be placed on a 512MB boundary</p>	ADDR_SIZE	Size	Valid ADR BASE Bits	0000	1MB	31:20	0001	2MB	31:21	0010	4MB	31:22	0011	8MB	31:23	0100	16MB	31:24	0101	32MB	31:25	0110	64MB	31:26	0111	128MB	31:27	1000	256MB	31:28	1001	512MB	31:29	1010 ^a	1GB	31:29	1011 ^a	2GB	31:29	others	reserved	
ADDR_SIZE	Size	Valid ADR BASE Bits																																												
0000	1MB	31:20																																												
0001	2MB	31:21																																												
0010	4MB	31:22																																												
0011	8MB	31:23																																												
0100	16MB	31:24																																												
0101	32MB	31:25																																												
0110	64MB	31:26																																												
0111	128MB	31:27																																												
1000	256MB	31:28																																												
1001	512MB	31:29																																												
1010 ^a	1GB	31:29																																												
1011 ^a	2GB	31:29																																												
others	reserved																																													
3	PREFETCH	FRW	0H	<p>Prefetch Enable: 1 = enable the aperture for read prefetching 0 = disable read prefetching</p>																																										
2-1	-	R	0H	reserved																																										
0	ENABLE	FRW	0H	<p>1 = enable Local-to-PCI aperture 0. 0 = disable Local-to-PCI aperture 0.</p>																																										

LB_MAP0, 1: LOCAL BUS TO PCI BUS ADDRESS MAP 0, 1

Mnemonic: LB_MAP0,1
 Offset: 5EH, 62H
 Size: 16 bits

LB_MAP0, 1				
Bits	Mnemonic	Type	Reset Value	Description
15-4	MAP_ADR	FRW	0H	Map Address: These bits correspond to bits AD[31:20] in PCI address space when a Local to PCI access is made. Address bits AD[19:2] are derived from the local bus itself. If the size of the aperture is increased, then the lower bits of MAP_ADR become masked off according to the ADR_SIZE bits in the LB_BASE registers.
3-1	TYPE	FRW	0H	Access Type: Determines which PCI bus command will be driven for local bus to PCI bus access: ^a 000 = Interrupt Acknowledge (Read) 001 = I/O Read/Write 011 = Memory Read/Write 101 = Configuration Read/Write 110 = Memory Read Multiple/Memory Write others = reserved
0	AD_LOW_EN	FRW	0H	Low Address Override Enable: When Set (1) the AD[1:0] value transmitted during the address phase will be generated from the AD_LOW value in the PCI_CFG register. When cleared (0) the value of AD[1:0] will be "00" except for I/O cycles where AD[1:0] correspond to the byte enables.

a. The value in this bit field is driven on the C/BE[3:1] PCI bus pins directly (except for TYPE="110": see note). C/BE0 is set based on whether the cycle is a read (0) or a write (1). No checking is done by the EPC device to see if the command type is supported (see the "Aperture Operation" section for more details). Reserved combinations can be used to generate other PCI commands directly.

Register Descriptions

Register Map

LB_BASE2: LOCAL BUS TO PCI BUS I/O APERTURE ADDRESS^a

Mnemonic: LB_BASE2
 Offset: 64H
 Size: 16 bits

LB_BASE 2																																		
Bits	Mnemonic	Type	Reset Value	Description																														
15-8	ADR_BASE	FRW	0H	Base Address: If the value of ADR_BASE matches that of local address bits 31:24 during the address phase of a local access then a match is detected. Since bits 31-24 are significant to the decoder, the size of the aperture is 16MB and is fixed at that size.																														
7-6	SWAP	FRW	0H	Byte Swap Control: Selects byte lane swapping for read and write cycles according to the following table: <table border="1" style="margin: 10px auto;"> <thead> <tr> <th></th> <th>SWAP</th> <th>D[31:24]</th> <th>D[23:16]</th> <th>D[15:8]</th> <th>D[7:0]</th> </tr> </thead> <tbody> <tr> <td>no swap, 32 bit</td> <td>00</td> <td>Q[31:24]</td> <td>Q[23:16]</td> <td>Q[15:8]</td> <td>Q[7:0]</td> </tr> <tr> <td>16 bit</td> <td>01</td> <td>Q[15:8]</td> <td>Q[7:0]</td> <td>Q[31:24]</td> <td>Q[23:16]</td> </tr> <tr> <td>8 bit</td> <td>10</td> <td>Q[7:0]</td> <td>Q[15:8]</td> <td>Q[23:16]</td> <td>Q[31:24]</td> </tr> <tr> <td></td> <td>11</td> <td colspan="4" style="text-align: center;">Auto Swap</td> </tr> </tbody> </table> <p>Auto Swap: When local bus $\overline{BE}[3:0] = "1100"$ or $"0011"$ then a 16 bit swap is done. When local bus $\overline{BE}[3:0] = "1110"$, $"1101"$, $"1011"$ or $"0111"$ then an 8-bit swap is done. Any other combination results in non-swapped data.</p>		SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]	no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]	16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]	8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]		11	Auto Swap			
	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]																													
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]																													
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]																													
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]																													
	11	Auto Swap																																
5-1	-	R	0H	reserved																														
0	ENABLE	FRW	0H	LB_BASE Enable: "1" to enable the aperture.																														

LB_MAP2: LOCAL BUS TO PCI BUS I/O ADDRESS MAP

Mnemonic: LB_MAP2
 Offset: 66H
 Size: 16 bits

LB_MAP2				
Bits	Mnemonic	Type	Reset Value	Description
15-8	MAP_ADR	FRW	0H	Map Address: These bits correspond to bits AD[31:24] in PCI address space when a Local to PCI access is made. Address bits AD[23:2] are derived from the local bus itself. Access through this aperture results in an I/O cycle on the PCI bus.
7-0	-	R	0H	reserved

LB_SIZE: LOCAL BUS SIZE RREGISTER

Mnemonic: LB_SIZE
 Offset: 68H
 Size: 32 bits

LB_SIZE																			
Bits	Mnemonic	Type	Reset Value	Description															
31-30	REGION_F	FRW	0H	Local Bus Width for address region 0xF0000000 to 0xFFFFFFFF: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>REGION_EF</th> <th>V350EPC (32 Bit mode) V360EPC</th> <th>V350EPC in 16 Bit Slave Mode Only</th> </tr> </thead> <tbody> <tr> <td>00</td> <td>32-bit</td> <td>16-bit Unpacked</td> </tr> <tr> <td>01</td> <td>16-bit Packed</td> <td>8-bit</td> </tr> <tr> <td>10</td> <td>8-bit</td> <td>16-bit Packed</td> </tr> <tr> <td>11</td> <td>16-bit Unpacked</td> <td>32-bit</td> </tr> </tbody> </table>	REGION_EF	V350EPC (32 Bit mode) V360EPC	V350EPC in 16 Bit Slave Mode Only	00	32-bit	16-bit Unpacked	01	16-bit Packed	8-bit	10	8-bit	16-bit Packed	11	16-bit Unpacked	32-bit
REGION_EF	V350EPC (32 Bit mode) V360EPC	V350EPC in 16 Bit Slave Mode Only																	
00	32-bit	16-bit Unpacked																	
01	16-bit Packed	8-bit																	
10	8-bit	16-bit Packed																	
11	16-bit Unpacked	32-bit																	
29-28	REGION_E	FRW	0H	Local Bus Width for address region 0xE0000000 to 0xEFFFFFFF															
27-26	REGION_D	FRW	0H	Local Bus Width for address region 0xD0000000 to 0xDFFFFFFF															
25-24	REGION_C	FRW	0H	Local Bus Width for address region 0xC0000000 to 0xCFFFFFFF															
23-22	REGION_B	FRW	0H	Local Bus Width for address region 0xB0000000 to 0xBFFFFFFF															
21-20	REGION_A	FRW	0H	Local Bus Width for address region 0xA0000000 to 0xAFFFFFFF															
19-18	REGION_9	FRW	0H	Local Bus Width for address region 0x90000000 to 0x9FFFFFFF															
17-16	REGION_8	FRW	0H	Local Bus Width for address region 0x80000000 to 0x8FFFFFFF															
15-14	REGION_7	FRW	0H	Local Bus Width for address region 0x70000000 to 0x7FFFFFFF															
13-12	REGION_6	FRW	0H	Local Bus Width for address region 0x60000000 to 0x6FFFFFFF															
11-10	REGION_5	FRW	0H	Local Bus Width for address region 0x50000000 to 0x5FFFFFFF															
9-8	REGION_4	FRW	0H	Local Bus Width for address region 0x40000000 to 0x4FFFFFFF															
7-6	REGION_3	FRW	0H	Local Bus Width for address region 0x30000000 to 0x3FFFFFFF															
5-4	REGION_2	FRW	0H	Local Bus Width for address region 0x20000000 to 0x2FFFFFFF															
3-2	REGION_1	FRW	0H	Local Bus Width for address region 0x10000000 to 0x1FFFFFFF															
1-0	REGION_0	FRW	0H	Local Bus Width for address region 0x00000000 to 0x0FFFFFFF															

LB_IO_BASE: LOCAL BUS I/O BASE

Mnemonic: LB_IO_BASE
 Offset: 6EH
 Size: 16 bits

LB_IO_BASE				
Bits	Mnemonic	Type	Reset Value	Description
15-0	ADR_BASE	FRW	0	Local to Internal Register Map Base Address: sets the base address for local bus side accesses to the EPC internal registers. If the value of ADR_BASE matches that of local address bits 31:16 during the address phase of a local access then a match is detected. Since bits 31-16 are significant to the decoder, the size of the aperture is 64KB (only a small fraction of this is used).

Register Descriptions

Register Map

FIFO_CFG: FIFO CONFIGURATION REGISTER

Mnemonic: FIFO_CFG
 Offset: 70H
 Size: 16 bits

FIFO_CFG				
Bits	Mnemonic	Type	Reset Value	Description
15-14	PBRST_MAX	FRW	0H	PCI Bus Maximum Burst Size : 00 = 4 Words 01 = 8 Words 10 = 16 Words 11 = 256 Words
13-12	PCI_WR_LB	FRW	0H	Write FIFO drain strategy for PCI Bus Write to Local Bus Aperture 0 and 1: 00 = Assert Local bus request immediately whenever the corresponding FIFO is not empty 01 = FIFO not empty and the PCI cycle filling it is finished ^a 10 = Assert Local bus request whenever the PCI bus to Local corresponding FIFO has 3 or more words of data pending 11 = Assert Local bus request whenever the PCI bus to Local FIFO has 3 or more words of data pending <i>or</i> the FIFO is not empty and the PCI cycle filling it is finished
11-10	PCI_RD_LB1 ^b	FRW	0H	Read FIFO fill strategy for PCI Bus Read from Local Bus Aperture 1: 00 = Assert Local bus request whenever the corresponding read FIFO is not full (room for 1 or more words available). 01 = Assert Local bus request whenever the corresponding read FIFO is at most half full (room for 2 or more words available). 10 = Assert Local bus request whenever the corresponding FIFO is empty 11 = reserved
9-8	PCI_RD_LB0 ^b	FRW	0H	FIFO control for PCI Bus Read from Local Bus Aperture 0: see description under PCI_RD_LB1, above.
7-6	LBRST_MAX	FRW	0H	Local Bus Maximum Burst Size: 00 =4 Words 10 =16 Words 01 =8 Words 11 =256 Words
5-4	LB_WR_PCI	FRW	0H	FIFO control for Local Bus Write to PCI Bus Aperture 0 and 1: 00 = Assert PCI bus request immediately whenever the corresponding FIFO is not empty 01 = FIFO not empty and the LB cycle filling it is finished 10 = Assert PCI bus request whenever the Local bus to PCI corresponding FIFO has 3 or more words of data pending 11 = Assert PCI bus request whenever the Local bus to PCI corresponding FIFO has 3 or more words of data pending <i>or</i> the FIFO is not empty and LB cycle filling it is finished
3-2	LB_RD_PCI1 ^b	FRW	0H	FIFO control for Local Bus Read from PCI Bus Aperture 1: 00 = Assert PCI bus request whenever the corresponding read FIFO is not full (room for 1 or more words available) 01 = Assert PCI bus request whenever the corresponding read FIFO is at most half full (room for 2 or more words available). 10 = Assert PCI bus request whenever the corresponding FIFO is empty 11 = reserved
1-0	LB_RD_PCI0 ^b	FRW	0H	FIFO control for Local Bus Read from PCI Bus Aperture 0: see description under LB_RD_PCI1, above.

a. The cycle filling the FIFO could be a DMA read or slave write

b. Has no effect on DMA transfers.

FIFO_PRIORITY: FIFO PRIORITY CONTROL REGISTER

Mnemonic: FIFO_PRIORITY
 Offset: 72H
 Size: 16 bit

FIFO_PRIORITY				
Bits	Mnemonic	Type	Reset Value	Description
15-13	-	R	0H	reserved
12	LOCAL	FRW	0H	Local Bus Request Priority: this controls the relative priority of pending transfers involving PCI to local bus. 0 = PCI write to local bus has priority over PCI read from local bus 1 = PCI read from local bus has priority over PCI write to local bus
11-10	LB_RD1	FRW	0H	Local Bus Read Flush Strategy for Aperture 1: this controls the condition that will cause Aperture 1 Local bus read prefetch FIFO to be flushed for the purpose of maintaining data coherency: 00 = Local bus to PCI writes never cause a flush ^a 01 = Flush at the end of a burst (don't keep extra data) 10 = Local bus to PCI write via aperture 1 will cause a flush (but not aperture 0) 11 = Any local bus to PCI write will cause a flush
9-8	LB_RD0	FRW	0H	Local Bus Read Flush Strategy for Aperture 0: this controls the condition that will cause aperture 0 Local bus read prefetch FIFO to be flushed for the purpose of maintaining data coherency: 00 = Local bus to PCI writes never cause a flush ^a 01 = Flush at the end of a burst (don't keep extra data) 10 = Local bus to PCI write via aperture 0 will cause a flush (but not aperture 1) 11 = Any local bus to PCI write will cause a flush
7-5	-	R	0H	reserved
4	PCI	FRW	0H	PCI Bus Request Priority: this controls the relative priority of pending transfers involving local bus to PCI bus. 0 = Local bus write to PCI has priority over Local bus read from PCI 1 = Local bus read from PCI has priority over Local bus write to PCI
3-2	PCI_RD1	FRW	0H	PCI Read Flush Strategy for Aperture 1: this controls the condition that will cause Aperture 1 PCI read prefetch FIFO to be flushed for the purpose of maintaining data coherency: 00 = PCI to local bus writes never cause a flush ^a 01 = Flush at the end of a burst (don't keep extra data) 10 = PCI to local bus write via aperture 1 will cause a flush (but not writes to aperture 0) 11 = Any PCI to local bus write will cause a flush
1-0	PCI_RD0	FRW	0H	PCI Read Flush Strategy for Aperture 0: this controls the condition that will cause aperture 0 PCI read prefetch FIFO to be flushed for the purpose of maintaining data coherency: 00 = PCI to local bus writes never cause a flush ^a 01 = Flush at the end of a burst (don't keep extra data) 10 = PCI to local bus write via aperture 0 will cause a flush (but not writes to aperture 1) 11 = Any PCI to local bus write will cause a flush

a. Only this option should be chosen when prefetching is disabled for the aperture. When prefetching is disabled, there is never any prefetch data to flush anyway.

Register Descriptions

Register Map

FIFO_STAT: FIFO STATUS REGISTER

Mnemonic: FIFO_STAT
 Offset: 74H
 Size: 16 bits

FIFO_STAT				
Bits	Mnemonic	Type	Reset Value	Description
15-14	-	R	0H	reserved
13-12	L2P_WR	R	0H	Status of the Local-to-PCI write FIFO: ^a 00 = Empty 01 = (While Filling) One or more words has been placed in the FIFO (While Draining) One data word remaining in the FIFO to be written to the PCI bus 10 = FIFO is completely full 11 = FIFO has room for only one more word
11-10	L2P_RD1	R	0H	PCI Bus Read from Local Bus Aperture 1 FIFO Status: 00 = Between empty and full 01 = Empty 10 = Full 11 = Between empty and full
9-8	L2P_RD0	R	0H	PCI Bus Read from Local Bus Aperture 0 FIFO Status: 00 = Between empty and full 01 = Empty 10 = Full 11 = Between empty and full
7-6	-	R	0H	reserved
5-4	P2L_WR	R	0H	Status of the PCI-to-Local write FIFO: ^b 00 = Empty 01 = (While Filling) One or more words has been placed in the FIFO (While Draining) One data word remaining in the FIFO to be written to the PCI bus 10 = FIFO is completely full 11 = FIFO has room for only one more word
3-2	P2L_RD1	R	0H	Local Bus Read from PCI Bus Aperture 1 FIFO Status: 00 = Between empty and full 01 = Empty 10 = Full 11 = Between empty and full
1-0	P2L_RD0	R	0H	Local Bus Read from PCI Bus Aperture 0 FIFO Status: 00 = Between empty and full 01 = Empty 10 = Full 11 = Between empty and full

a. The transitions for the bits in this field follows a hysteresis curve depending on whether the FIFO has reached certain levels of "fullness". Please see the "Bridge Operation" section for more detail.

b. The transitions for the bits in this field follows a hysteresis curve depending on whether the FIFO has reached certain levels of "fullness". Please see the "Bridge Operation" section for more detail.

LB_ISTAT: LOCAL BUS INTERRUPT CONTROL AND STATUS REGISTER

Mnemonic: LB_ISTAT
 Offset: 76H
 Size: 8 bits

LB_ISTAT				
Bits	Mnemonic	Type	Reset Value	Description
7	MAILBOX ^a	R	0H	1 = an interrupt has been requested by one or more of the mailbox registers 0 = no mailbox interrupts pending
6	PCI_RD	FRW	0H	1 = Target Abort or lack of Device Select has been seen during a local bus to PCI bus read access and an interrupt request for such events has been enabled; clear by writing "0" 0 = no local bus read of PCI space error interrupt request is pending.
5	PCI_WR	FRW	0H	1 = Target Abort or lack of Device Select is seen during a local bus to PCI bus write access and an interrupt request for such events has been enabled; clear by writing "0" 0 = no local bus write to PCI space error interrupt request is pending.
4	PCI_INT ^b	FRW	0H	1 = a PCI interrupt pin has requested an interrupt 0 = no pending PCI interrupt events
3	PCI_PERR	RW	0H	PCI Parity Error Interrupt: This bit is set in response to parity error seen on the PCI bus as a result of the VxxxEPC acting as either a master or a slave for the cycle.
2	I2O_QWR ^a	RW	0H	I2O Inbound Post Queue Write Interrupt: This bit is set when 3 conditions are met: I2O is enabled, the corresponding bit in LB_IMASK is enabled and the inbound post list is written. Cleared by writing '0'.
1	DMA1	FRW	0H	1 = DMA channel 1 has requested an interrupt;clear by writing "0" 0 = DMA channel 1 has not requested an interrupt.
0	DMA0	FRW	0H	1 = DMA channel 0 has requested an interrupt;clear by writing "0" 0 = DMA channel 0 has not requested an interrupt.

a. This bit is a logical OR of all of the mailbox interrupt requests. It is only cleared when all of the individual mailbox interrupt requests have been cleared via reading or writing the applicable mailbox register. See the "Mailbox Register" section of the "Bridge Operation" chapter for more information.

b. Note: All writable status bits are cleared by writing '0'. Writing '1' has no effect.

Register Descriptions

Register Map

LB_IMASK: LOCAL BUS INTERRUPT MASK REGISTER

Mnemonic: LB_IMASK
 Offset: 77H
 Size: 8 bits

LB_IMASK				
Bits	Mnemonic	Type	Reset Value	Description
7	MAILBOX	FRW	0H	Global Mailbox Interrupt Enable: 1 = mailbox interrupts are enabled 0 = all mailbox interrupts are masked
6	PCI_RD	FRW	0H	PCI Read Error Interrupt Enable: 1 = enable local interrupt requests for PCI read errors 0 = mask local interrupt requests for PCI read errors
5	PCI_WR	FRW	0H	PCI Write Error Interrupt Enable: 1 = enable local interrupt requests for PCI write errors 0 = mask local interrupt requests for PCI write errors
4	PCI_INT	FRW	0H	Global PCI Interrupt to Local Interrupt Enable: 1 = enable PCI interrupt requests to request a local interrupt 0 = mask PCI interrupt requests to request a local interrupt
3	PCI_PERR	RW	0H	PCI Parity Error Interrupt Enable: When enabled (1) the corresponding bit in LB_ISTAT is set in response to a parity error event seen on the PCI bus. In order for a PCI parity event to be detected one or more of the MASTER_PI and/or the SLAVE_PI bits in PCI_INT_CFG must be enabled in addition to this bit.
2	I2O_QWR	RW	0H	I2O Inbound Post Queue Write Interrupt Enable: Set (1) to enable inbound post queue write cycles to generate interrupts on the Local Bus.
1	DMA1	FRW	0H	DMA Channel 1 Interrupt Enable: 1 = enable DMA Channel 1 interrupt requests 0 = mask DMA Channel 1 interrupt requests
0	DMA0	FRW	0H	DMA Channel 0 Interrupt Enable: 1 = enable DMA Channel 0 interrupt requests 0 = mask DMA Channel 0 interrupt requests

SYSTEM REGISTER

Mnemonic: SYSTEM
 Offset: 78H
 Size: 16 bits

SYSTEM				
Bits	Mnemonic	Type	Reset Value	Description
15	RST_OUT	FR	0H	Reset Output Control: When set to '1' the reset output (see RDIR pin description) is de-asserted.
14	LOCK	FR	0H	Lock Register Contents: When set to '1' the SYSTEM register becomes unwritable. LOCK can only be cleared by writing the SYSTEM register with 0xA05F ^a .
13	SPROM_EN	FR	0H	Serial PROM Software Access Enable. 1 = SCL/Local parity error pin functions as SCL 0 = SCL/Local parity error pin functions as local parity error
12	SCL	FR	0H	Serial PROM Clock Output. When SPROM_EN is enabled ('1') then this bit controls the state of the SCL pin.
11	SDA_OUT	FR	0H	Serial PROM Data Output. When SPROM_EN is enabled ('1') then this bit controls the state of the SDA pin. When SDA_OUT is '0' then the SDA pin will be driven low. For SDA_OUT = '1' the SDA pin floats to high impedance.
10	SDA_IN	R	X	Serial PROM Data In. Reads back the SDA input directly from the pin.
9	POE	FR	0H	Local Bus Parity: 1=odd parity 0=even parity
8	FAST_REQ	FR	0H	0 = bus follows strict Am29030 protocol 1 = bus follows high-performance Am29K protocol This bit has no affect on devices other than the V292EPC.
7	-	R	0H	reserved
6	LB_RD_PCI1	W	0H	1 = Local Read from PCI Bus (Aperture 1) FIFO Flush ^b 0 = no operation
5	LB_RD_PCI0	W	0H	1 = Local Read from PCI Bus (Aperture 0) FIFO Flush ^b 0 = no operation
4	LB_WR_PCI	W	0H	1 = Local to PCI Write FIFO Flush ^b 0 = no operation
3	-	R	0H	reserved
2	PCI_RD_LB1	W	0H	1 = PCI Read from Local Bus (Aperture 1) FIFO Flush ^b 0 = no operation
1	PCI_RD_LB0	W	0H	1 = PCI Read from Local Bus (Aperture 0) FIFO Flush ^b 0 = no operation
0	PCI_WR_LB	W	0H	1 = PCI to Local Write FIFO Flush ^b 0 = no operation

a. Writing 0xA05F to un-lock the system register will not overwrite the current System register values.

b.LOCK bit in SYSTEM register must be set to "0" to write this bit.

Register Descriptions

Register Map

LB_CFG REGISTER: LOCAL BUS CONFIGURATION REGISTER^a

Mnemonic: LB_CFG
 Offset: 7AH
 Size: 16 bits

LB_CFG				
Bits	Mnemonic	Type	Reset Value	Description
15	-	R	0H	reserved
14-13	TO_LENGTH	FRW		Local Bus Time-out Time Constant: "11" = 1024 cycles, "10" = 512 cycles, "01" = 256 cycles, "00" = 64 cycles
12	LB_RST	FRW	0H	Local Bus Reset control: when set to '1', the RST_OUT bit in SYSTEM will also control the operation of the local bus master/slave state machines so that they will return to their idle states when RST_OUT is '0'. when LB_RST is cleared to '0', then <u>RST_OUT</u> only controls the state of the reset output (either LRST or PRST) and the state of the local master/slave are not affected.
11	PPC_RDY	FRW	0H	Power PC Ready: this bit is defined only for the V292EPC and determines how the <u>RDY</u> signal operates when a local bus master (such as a PPC403Gx) accesses the V292EPC for read cycles. When disabled (0) the <u>RDY</u> signal operates normally. However, when enabled (1) the relationship of read data to the <u>RDY</u> signal is modified so that data will exist for one cycle after <u>RDY</u> (normally valid data would be seen at the same time as <u>RDY</u> is seen). This allows the V292EPC to be used with the PPC403Gx with only a small single programmable logic device. No address/data path registers are required.
10	LB_INT	FRW	0H	Local Bus Interrupt Enable: When this bit is enabled and the PCI_RD and/or PCI_WR bits are enabled in LB_IMASK, a time-out event will cause the local bus interrupt to be asserted via the LB_ISTAT register bits PCI_RD (for time-out on a read) or PCI_WR (for time-out on a write)
9	ERR_EN ^a	FRW	0H	BTERM/ERR Enable: When enabled ('1') the V962EPC will assert the local bus BTERM(V961EPC, V962EPC) or ERR(V292EPC) signal for one clock whenever a time-out occurs. A time-out event occurs when a request to the EPC is outstanding for longer than the value determined by the TO_LENGTH register bit.
8	RDY_EN	FRW	0H	Ready Enable: When enabled ('1') the EPC will assert the local bus <u>READY</u> or <u>RDY</u> signal for one clock whenever a time-out occurs.
7 ^b	BE_IMODE	FRW	0H	Byte Enable Input Mode: this bit is defined only for the EPC when in 29K bus mode and determines how the <u>BWE[3:0]</u> signals operate as inputs. When this bit is clear (0) the local bus slave controller on the EPC will assume that any read access to the EPC by an external master device is 32 bits (all bytes are enabled). When set (1) the EPC slave controller will treat the <u>BWE[3:0]</u> as byte enables for both read and write.
6 ^b	BE_OMODE	FRW	0H	Byte Enable Output Mode: this bit is defined only for the EPC when in 29K bus mode to determine how the <u>BWE[3:0]</u> signals operate as outputs. When this bit is clear (0) the local bus master controller on the EPC will drive the <u>BWE[3:0]</u> active only during write cycles and remain de-asserted for read cycles. When set (1) the V292EPC will assert the appropriate byte enable information onto the <u>BWE[3:0]</u> during both read and write cycles.
5	ENDIAN	FRW	0H	Endian Mode: This determines where 8 and 16-bit data is connected to the 32-bit local data bus for 8 and 16 bit local master access. It also determines the byte order for 8/16 bit operations. 0=little endian, 1=big endian.

LB_CFG (cont'd)				
Bits	Mnemonic	Type	Reset Value	Description
4	PARK_EN	FRW	0H	Local Bus Parking Enable: When set (1), the local bus address and data lines will be driven when the grant is asserted to the EPC and the bus state is idle.
3	-	R	0H	reserved
2	FBB_DIS ^b	FRW	0H	Fast Back-to-Back Disabled: When set (1), the local bus master will not perform fast back-to-back cycles.
1-0	-	R	0H	reserved

a. The function of ERR_EN='1' and RDY_EN='1' used together is undefined.

b. This is a reserved bit in EPC Revision A0

Register Descriptions

Register Map

PCI_CFG: PCI BUS CONFIGURATION REGISTER^a

Mnemonic: PCI_CFG
 Offset: 7CH
 Size: 16 bits

PCI_CFG				
Bits	Mnemonic	Type	Reset Value	Description
15	I2O_EN	FRW	0H	I2O Enable
14	IO_REG_DIS	FRW	0H	Disable PCI_IO_BASE register: when set (1) the contents of PCI_IO_BASE register will read back '0' although the register may actually contain non-zero data. The register remains writeable and the decode function of the chip will remain intact.
13	IO_DIS	FRW	0H	Disable PCI_IO_BASE Decoder: when set (1) PCI_IO_BASE will not respond to PCI cycles.
12	EN3V	R	0H	Enable I/O buffers for 3.3V operation.
11	-	R	0H	reserved
10	RETRY_EN	FR	0/1H ^a	PCI Configuration Retry: When set ('1'), the EPC will retry all PCI configuration cycles that are targeted at the internal registers.
9-8	AD_LOW	FRW	0H	PCI AD[1:0]. Override Value: When one of the LB_MAP registers is programmed with AD_LOW_EN set (1) then the value in this register is used to generate AD[1:0] instead of the normal value ("00" for all except I/O cycles where the byte enables determine the value).
7-5	DMA_RTYPE	FRW	0H	DMA Read from PCI Bus Command Type: determines the PCI command type applied to the C/BE#(3:1) outputs for a PCI read cycle from the DMA controller. C/BE# bit 0 will be driven '0' for all write cycles and thus there is no corresponding register bit. Writing DMA_RTYPE "000" will cause the value 011 to be written instead. This results in a "memory read" command type.
4	-	R	0H	reserved
3-1	DMA_WTYPE	FRW	0H	DMA Write to PCI Bus Command Type: determines the PCI command type applied to the C/BE[3:1] outputs for a PCI write cycle from the DMA controller. C/BE bit 0 will be driven '1' for all write cycles and thus there is no corresponding register bit. Writing DMA_WTYPE "000" will cause the value 011 to be written instead. This results in a "memory write" command type.
0	-	R	0H	reserved

a. This bit is initialized to '1' when SDA is pulled high and no EEPROM device attached.

DMA_PCI_ADR: PCI DMA ADDRESS REGISTERS

Mnemonic: DMA_PCI_ADDR0, 1
 Offset: 80H, 90H
 Size: 32 bits

DMA_PCI_ADDR0, 1				
Bits	Mnemonic	Type	Reset Value	Description
31-2	ADR	RW	0H	PCI Byte Address
1-0		R	0H	These low address bits read back zero since all DMA transfers are 32 bit aligned



Register Descriptions

Register Map

DMA_LOCAL_ADR: LOCAL DMA ADDRESS REGISTERS

Mnemonic: DMA_LOCAL_ADDR0, 1
 Offset: 84H, 94H
 Size: 32 bits

DMA_LOCAL_ADDR0, 1				
Bits	Mnemonic	Type	Reset Value	Description
31-2	ADR	RW	0H	Local Byte Address
1-0		R	0H	These low address bits read back zero since all DMA transfers are 32 bit aligned

DMA_LENGTH0, 1: DMA TRANSFER LENGTH REGISTER 0, 1

Mnemonic: DMA_LENGTH0, 1
 Offset: 88H, 98H
 Size: 24 bits

DMA_LENGTH0, 1				
Bits	Mnemonic	Type	Reset Value	Description
23	DREQ_EN	RW	0H	External DMA Request Enable: When set (1) the DMA will be throttled by to the state of the INTC# input pin (DMA Channel 0) or INTD# input pin (DMA Channel 1). The corresponding pin must be low (0) to allow the DMA to fetch the data source.
22	INTR_EN	RW	0H	Interrupt on Link Complete: When set (1) an internal interrupt from the DMA controller will generated whenever the data transfer portion of a link is complete. The internal DMA interrupt can be routed to PCI or local interrupt outputs by enabling them in the PCI_INT_CFG and/or LB_IMASK registers. An internal interrupt is always generated upon chain completion when the DMA_IPR bit is cleared by the hardware.
21-20	-	R	0H	reserved
19-0	COUNT	RW	0H	Transfer Count Remaining. This register holds the initial transfer count (in 32-bit words) and is updated with the remaining count after each DMA transfer.

DMA_CSR: DMA CONTROL AND STATUS REGISTERS

Mnemonic: DMA_CSR0, 1
 Offset: 8BH, 9BH
 Size: 8 bits

DMA_CSR0, 1																																		
Bits	Mnemonic	Type	Reset Value	Description																														
7	CHAIN	RW	0H	1 = enable DMA chaining for this transfer 0 = disable DMA chaining for this transfer																														
6	CLR_LEN	RW	0H	Clear Length: when set (1), the DMA_LENGTH value in the memory based descriptor will be cleared after the transfer is complete.																														
5	PRIORITY	RW	0H	Controls the relative priority of DMA channels. See "DMA Controller" chapter.																														
4	DIRECTION	RW	0H	DMA Direction: 0 = Local to PCI 1 = PCI to Local																														
3-2	SWAP	RW	0H	Byte Swap Control: Selects byte order conversion options: ^a <table border="1" data-bbox="624 872 1241 1025"> <thead> <tr> <th></th> <th>SWAP</th> <th>D[31:24]</th> <th>D[23:16]</th> <th>D[15:8]</th> <th>D[7:0]</th> </tr> </thead> <tbody> <tr> <td>no swap, 32 bit</td> <td>00</td> <td>Q[31:24]</td> <td>Q[23:16]</td> <td>Q[15:8]</td> <td>Q[7:0]</td> </tr> <tr> <td>16 bit</td> <td>01</td> <td>Q[15:8]</td> <td>Q[7:0]</td> <td>Q[31:24]</td> <td>Q[23:16]</td> </tr> <tr> <td>8 bit</td> <td>10</td> <td>Q[7:0]</td> <td>Q[15:8]</td> <td>Q[23:16]</td> <td>Q[31:24]</td> </tr> <tr> <td>reserved</td> <td>11</td> <td></td> <td></td> <td></td> <td></td> </tr> </tbody> </table>		SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]	no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]	16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]	8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]	reserved	11				
	SWAP	D[31:24]	D[23:16]	D[15:8]	D[7:0]																													
no swap, 32 bit	00	Q[31:24]	Q[23:16]	Q[15:8]	Q[7:0]																													
16 bit	01	Q[15:8]	Q[7:0]	Q[31:24]	Q[23:16]																													
8 bit	10	Q[7:0]	Q[15:8]	Q[23:16]	Q[31:24]																													
reserved	11																																	
1	ABORT	W	0H	1 = immediately abort current DMA transfer in process ^a 0 = no operation																														
0	DMA_IPR	RW	0H	DMA Initiate Process: Write 1 = begin DMA operation Write 0 = no operation Read 1 = DMA in Progress Read 0 = DMA Idle Automatically cleared when the transfer count expires and there are no further chains to process.																														

- a. Writing to DMA_CSRx with the ABORT bit set will cause all other bits in the register to maintain their previous value (they are not written). The transfer can be restarted by setting DMA_IPR again using a read-modify-write to maintain the contents of the other register bits.

Register Descriptions

Register Map

DMA_CTLB_ADR: DMA CONTROL BLOCK ADDRESS REGISTER 0,1

Mnemonic: DMA_CTLB_ADDR0, 1
Offset: 8CH, 9CH
Size: 32 bits

DMA_CTLB_ADDR0, 1				
Bits	Mnemonic	Type	Reset Value	Description
31-4	CTLB_ADR	RW	0H	DMA control block address. Address of the first control block in a DMA chain. Must be aligned to a 16 byte boundary and reside in Local memory.
3-0	-	R	0H	reserved

DMA_DELAY: DMA DESCRIPTOR DELAY

Mnemonic: DMA_DELAY
Offset: E0H
Size: 8 bits

DMA_DELAY				
Bits	Mnemonic	Type	Reset Value	Description
7-0	DELAY	RW	0H	Determines the delay between the completion of processing a DMA descriptor and the loading of the next DMA descriptor.

MAIL_DATA0-15: MAILBOX DATA REGISTER 0-15

Mnemonic: MAIL_DATA
Offset: C0H - CFH (see register map)
Size: 8 bits

MAIL_DATA0-15				
Bits	Mnemonic	Type	Reset Value	Description
7-0	DATA	RW	0H	Mailbox Data (mapped into local bus address space): Application specific, software defined data register.

PCI BUS MAILBOX WRITE INTERRUPT CONTROL REGISTER

Mnemonic: PCI_MAIL_IEMR
 Offset: D0H
 Size: 16 bits

PCI_MAIL_IEMR				
Bits	Mnemonic	Type	Reset Value	Description
15	EN15	RW	0H	1 = enable local interrupts on PCI bus writes to mailbox 15 0 = disable local interrupts on PCI bus writes to mailbox 15
14	EN14	RW	0H	Same as above for mailbox 14.
13	EN13	RW	0H	Same as above for mailbox 13.
12	EN12	RW	0H	Same as above for mailbox 12.
11	EN11	RW	0H	Same as above for mailbox 11.
10	EN10	RW	0H	Same as above for mailbox 10.
9	EN9	RW	0H	Same as above for mailbox 9.
8	EN8	RW	0H	Same as above for mailbox 8.
7	EN7	RW	0H	Same as above for mailbox 7.
6	EN6	RW	0H	Same as above for mailbox 6.
5	EN5	RW	0H	Same as above for mailbox 5.
4	EN4	RW	0H	Same as above for mailbox 4.
3	EN3	RW	0H	Same as above for mailbox 3.
2	EN2	RW	0H	Same as above for mailbox 2.
1	EN1	RW	0H	Same as above for mailbox 1.
0	EN0	RW	0H	Same as above for mailbox 0.

PCI BUS MAILBOX READ INTERRUPT CONTROL REGISTER

Mnemonic: PCI_MAIL_IERD
 Offset: D2H
 Size: 16 bits

PCI_MAIL_IERD				
Bits	Mnemonic	Type	Reset Value	Description
15	EN15	RW	0H	1 = enable local interrupts on PCI bus read from mailbox 15 0 = disable local interrupts on PCI bus read from mailbox 15
14	EN14	RW	0H	Same as above for mailbox 14.
13	EN13	RW	0H	Same as above for mailbox 13.
12	EN12	RW	0H	Same as above for mailbox 12.
11	EN11	RW	0H	Same as above for mailbox 11.
10	EN10	RW	0H	Same as above for mailbox 10.
9	EN9	RW	0H	Same as above for mailbox 9.
8	EN8	RW	0H	Same as above for mailbox 8.
7	EN7	RW	0H	Same as above for mailbox 7.
6	EN6	RW	0H	Same as above for mailbox 6.
5	EN5	RW	0H	Same as above for mailbox 5.

Register Descriptions

Register Map

PCI_MAIL_IERD (cont'd)				
Bits	Mnemonic	Type	Reset Value	Description
4	EN4	RW	0H	Same as above for mailbox 4.
3	EN3	RW	0H	Same as above for mailbox 3.
2	EN2	RW	0H	Same as above for mailbox 2.
1	EN1	RW	0H	Same as above for mailbox 1.
0	EN0	RW	0H	Same as above for mailbox 0.

LOCAL BUS MAILBOX WRITE INTERRUPT CONTROL REGISTER

Mnemonic: LB_MAIL_IWR
 Offset: D4H
 Size: 16 bits

LB_MAIL_IWR				
Bits	Mnemonic	Type	Reset Value	Description
15	EN15	RW	0H	1 = enable PCI interrupts on local bus writes to mailbox 15 0 = disable PCI interrupts on local bus writes to mailbox 15
14	EN14	RW	0H	Same as above for mailbox 14.
13	EN13	RW	0H	Same as above for mailbox 13.
12	EN12	RW	0H	Same as above for mailbox 12.
11	EN11	RW	0H	Same as above for mailbox 11.
10	EN10	RW	0H	Same as above for mailbox 10.
9	EN9	RW	0H	Same as above for mailbox 9.
8	EN8	RW	0H	Same as above for mailbox 8.
7	EN7	RW	0H	Same as above for mailbox 7.
6	EN6	RW	0H	Same as above for mailbox 6.
5	EN5	RW	0H	Same as above for mailbox 5.
4	EN4	RW	0H	Same as above for mailbox 4.
3	EN3	RW	0H	Same as above for mailbox 3.
2	EN2	RW	0H	Same as above for mailbox 2.
1	EN1	RW	0H	Same as above for mailbox 1.
0	EN0	RW	0H	Same as above for mailbox 0.

LOCAL BUS MAILBOX READ INTERRUPT CONTROL REGISTER

Mnemonic: LB_MAIL_IERD
 Offset: D6H
 Size: 16 bits

LB_MAIL_IERD				
Bits	Mnemonic	Type	Reset Value	Description
15	EN15	RW	0H	1 = enable PCI interrupts on local bus reads from mailbox 15 0 = disable PCI interrupts on local bus reads from mailbox 15
14	EN14	RW	0H	Same as above for mailbox 14.
13	EN13	RW	0H	Same as above for mailbox 13.
12	EN12	RW	0H	Same as above for mailbox 12.
11	EN11	RW	0H	Same as above for mailbox 11.
10	EN10	RW	0H	Same as above for mailbox 10.
9	EN9	RW	0H	Same as above for mailbox 9.
8	EN8	RW	0H	Same as above for mailbox 8.
7	EN7	RW	0H	Same as above for mailbox 7.
6	EN6	RW	0H	Same as above for mailbox 6.
5	EN5	RW	0H	Same as above for mailbox 5.
4	EN4	RW	0H	Same as above for mailbox 4.
3	EN3	RW	0H	Same as above for mailbox 3.
2	EN2	RW	0H	Same as above for mailbox 2.
1	EN1	RW	0H	Same as above for mailbox 1.
0	EN0	RW	0H	Same as above for mailbox 0.

MAIL_WR_STAT: MAILBOX WRITE INTERRUPT STATUS

Mnemonic: MAIL_WR_STAT
 Offset: D8H
 Size: 16 bits

MAIL_WR_STAT				
Bits	Mnemonic	Type	Reset Value	Description
15	WR_STAT15	RW	0H	1 = Mailbox 15 has requested a PCI or local write interrupt 0 = Mailbox 15 has not requested a PCI or local write interrupt Cleared by writing '1'. Writing '0' has no effect
14	WR_STAT14	RW	0H	same as above for mailbox 14
13	WR_STAT13	RW	0H	same as above for mailbox 13
12	WR_STAT12	RW	0H	same as above for mailbox 12
11	WR_STAT11	RW	0H	same as above for mailbox 11
10	WR_STAT10	RW	0H	same as above for mailbox 10
9	WR_STAT9	RW	0H	same as above for mailbox 9
8	WR_STAT8	RW	0H	same as above for mailbox 8
7	WR_STAT7	RW	0H	same as above for mailbox 7
6	WR_STAT6	RW	0H	same as above for mailbox 6

Register Descriptions

Register Map

MAIL_WR_STAT (cont'd)				
Bits	Mnemonic	Type	Reset Value	Description
5	WR_STAT5	RW	0H	same as above for mailbox 5
4	WR_STAT4	RW	0H	same as above for mailbox 4
3	WR_STAT3	RW	0H	same as above for mailbox 3
2	WR_STAT2	RW	0H	same as above for mailbox 2
1	WR_STAT1	RW	0H	same as above for mailbox 1
0	WR_STAT0	RW	0H	same as above for mailbox 0

MAIL_RD_STAT: MAILBOX READ INTERRUPT STATUS

Mnemonic: MAIL_RD_STAT
 Offset: DAH
 Size: 16 bits

MAIL_RD_STAT				
Bits	Mnemonic	Type	Reset Value	Description
15	RD_STAT15	RW	0H	1 = Mailbox 15 has requested a PCI or local read interrupt 0 = Mailbox 15 has not requested a PCI or local read interrupt Cleared by writing '1'. Writing '0' has no effect
14	RD_STAT14	RW	0H	same as above for mailbox 14
13	RD_STAT13	RW	0H	same as above for mailbox 13
12	RD_STAT12	RW	0H	same as above for mailbox 12
11	RD_STAT11	RW	0H	same as above for mailbox 11
10	RD_STAT10	RW	0H	same as above for mailbox 10
9	RD_STAT9	RW	0H	same as above for mailbox 9
8	RD_STAT8	RW	0H	same as above for mailbox 8
7	RD_STAT7	RW	0H	same as above for mailbox 7
6	RD_STAT6	RW	0H	same as above for mailbox 6
5	RD_STAT5	RW	0H	same as above for mailbox 5
4	RD_STAT4	RW	0H	same as above for mailbox 4
3	RD_STAT3	RW	0H	same as above for mailbox 3
2	RD_STAT2	RW	0H	same as above for mailbox 2
1	RD_STAT1	RW	0H	same as above for mailbox 1
0	RD_STAT0	RW	0H	same as above for mailbox 0

QBA_MAP : LOCATING THE QUEUE IN LOCAL MEMORY^a

Mnemonic: QBA_MAP
 Offset: DCH
 Size: 32 bits

QBA_MAP																									
Bits	Mnemonic	Type	Reset Value	Description																					
31-20	BASE	RW	0H	Queue Base Address: These bits are used to generate bits 31-20 of all Inbound/Outbound pointers. Register bits so that lower bits of the decode are masked off.																					
19-10	-	R	0H	reserved																					
10-8	QSIZE	RW	0H	Queue Size: The size of the aperture is determined as follows: <table border="1" style="margin-left: 20px;"> <thead> <tr> <th>QSIZE</th> <th>Size</th> <th>Auto Increment Mask</th> </tr> </thead> <tbody> <tr> <td>000</td> <td>4K entries (16K Bytes)</td> <td>0x3FFC</td> </tr> <tr> <td>001</td> <td>8K entries (32K Bytes)</td> <td>0x7FFC</td> </tr> <tr> <td>010</td> <td>16K entries (64K Bytes)</td> <td>0xFFFC</td> </tr> <tr> <td>011</td> <td>32K entries (128K Bytes)</td> <td>0x1FFFC</td> </tr> <tr> <td>100</td> <td>64K entries (256K Bytes)</td> <td>0x3FFFC</td> </tr> <tr> <td>others</td> <td colspan="2">reserved</td> </tr> </tbody> </table>	QSIZE	Size	Auto Increment Mask	000	4K entries (16K Bytes)	0x3FFC	001	8K entries (32K Bytes)	0x7FFC	010	16K entries (64K Bytes)	0xFFFC	011	32K entries (128K Bytes)	0x1FFFC	100	64K entries (256K Bytes)	0x3FFFC	others	reserved	
QSIZE	Size	Auto Increment Mask																							
000	4K entries (16K Bytes)	0x3FFC																							
001	8K entries (32K Bytes)	0x7FFC																							
010	16K entries (64K Bytes)	0xFFFC																							
011	32K entries (128K Bytes)	0x1FFFC																							
100	64K entries (256K Bytes)	0x3FFFC																							
others	reserved																								
7-1	-	R	0H	reserved																					
0	ONLINE	RW	0H	I2O Device On Line: when clear (0), the VxxxEPC will return 0xFFFFFFFF when the inbound and outbound ports are read. This bit should only be enabled after the queues have been initialized by the local processor and it is ready to accept an inbound MFA. This bit must be clear in order for the local processor to modify the contents of the OFL_HEAD, OPL_TAIL, IPL_HEAD and IFL_TAIL registers.																					

I2O MESSAGE UNIT POINTERS

Offset: A0H-BFH
 Size: 32 bits each

Reg. Name	Offset	Description	Pointer Maintenance
OFL_HEAD	0xBC	Outbound Free List Head Pointer	PCI write of Outbound Port auto-increments
OFL_TAIL	0xB8	Outbound Free List Tail Pointer	Updated by local processor
OPL_HEAD	0xB4	Outbound Post List Head Pointer	Updated by local processor
OPL_TAIL	0xB0	Outbound Post List Tail Pointer	PCI read of Outbound Port auto-increments
IPL_HEAD	0xAC	Inbound Post List Head Pointer	PCI write of Inbound Port auto-increments
IPL_TAIL	0xA8	Inbound Post List Tail Pointer	Updated by local processor
IFL_HEAD	0xA4	Inbound Free List Head Pointer	Updated by local processor
IFL_TAIL	0xA0	Inbound Free List Tail Pointer	PCI read of Inbound Port auto-increments

Register Descriptions

Register Map



Glossary

This glossary contains terms used in EPC User's Manual. For reference to more information about a term, please refer to index and PCI 2.1 specification.

Burst - Bus protocol that allows a bus master to request more than one data transfer for an access. Typically, the data is sequential, sequential over a modulo boundary or cache line toggle (such as the 486).

Deadlock - The condition in which one processor (or master device) attempts to access a resource that is tightly coupled to a second processor (or master device) that is also in a state of attempting to access a resource local to the first processor.

DMA - Direct Memory Access. A DMA controller allows the CPU to continue operation while the EPC controls block transfer between Local and PCI address space.

Dynamic Bandwidth Allocation™ - A technique that allows a single FIFO to be dynamically shared for multiple purposes and also negates the need to require a bus retry each time a non-sequential address is begun.

Endian - The organization of sub-word data within the physical data bus. Little Endian processors address the first byte of a 32-bit word on the least significant data lines. Big Endian processors address the first byte of a 32-bit word on the most significant data lines. The PCI bus is strictly little endian and the internal registers of the EPC reflect this fact.

Wait State - A processor clock cycle in which no data transfer occurs although data transfer has been requested. Used to hold off a requesting master until data from memory or I/O is ready.

Word - The native bus size of the system. For this document, it is 32 bits which is the size of the PCI bus.

Glossary

Index

Symbols

'FR' 111, 115
'FRW' 111, 115
'R' 111, 115
'RW' 115
'W' 115

A

Address Translation 21
ADS 74, 77
Am29K 1, 4
Aperture 1, 8, 17, 25
Aperture Base Address 18

B

Back-to-Back 84
BIOS 23
 $\overline{\text{BLAST}}$ 74
 $\overline{\text{BOFF}}$ 78
Boundary 40, 45, 46
 $\overline{\text{BREQ}}$ 45
 $\overline{\text{BTERM}}$ 74, 77, 135
 $\overline{\text{BURST}}$ 74
Byte Order 25, 42

C

$\text{C}/\overline{\text{BE}}$ 23, 24, 47, 120
Chaining 38, 43
Chaining Descriptor 38
Configuration 83
Configuration Read 24, 45, 48
Configuration Write 24, 46, 49
Crosspoint Interrupt 97, 100

D

Data Byte Order 21
Deadlock 47
DEVSEL 47, 58
Disconnect 59
DMA 11, 27, 28
DMA Interrupt 40
DMA Programming 41
DMA Throttling 40
Doorbell 97
Doorbell Interrupt 11, 94
DOS Support 13, 18, 89

Index

Draining Strategy 29
Dual Address Cycle 24
Dynamic Bandwidth Allocation 1, 27

E

EEPROM 1, 47, 105, 107, 108, 109, 111
EEPROM Programming 111
Endian Conversion 21
EPROM 13, 14
ERR 135
Expansion ROM 23

F

FIFO 27
FIFO Architecture 30
FIFO Draining 29
FIFO Programming 31

G

$\overline{\text{GNT}}$ 29, 59

H

HOLD 45, 74, 77
HOLDA 74, 77

I

I/O Read 24, 45, 48
I/O Space 89
I/O Write 24, 46, 49
I2C 111
i960 1, 4
i960 Processor Configuration Note 110
IDSEL 47, 83, 85, 110
Initialization 105
 $\overline{\text{INTA}}$ 100, 102, 103
 $\overline{\text{INTB}}$ 102, 103
 $\overline{\text{INTC}}$ 102, 103
 $\overline{\text{INTD}}$ 100, 102, 103
Internal Register 13
Internal Registers 14
Interrupt 97
Interrupt Acknowledge 24, 45, 48
Interrupt Crosspoint 101
 $\overline{\text{INTx}}$ 100, 122
IO Cycle 13
 $\overline{\text{IRDY}}$ 48

L

Latency Timer 59
 $\overline{\text{LBGRT}}$ 74

LBREQ 74
LICU 95, 97, 99
LINT 74, 103
Little-Endian 38
Local-from-PCI Read 9
Local-to-PCI Write 9
LPARx 80
LREQ 75
LRESET 105, 111

M

Mailbox Register 1, 11, 93
Mailbox Registers Programming 95
Masking 98
Master 1, 7
Memory Read 24, 45, 48
Memory Read Line 24, 45, 48
Memory Read Multiple 24, 45, 48
Memory Write 24, 46, 49
Memory Write and Invalidate 24, 46, 49

O

Overlapping 22

P

PC/DOS 89
PCI 2.1 1, 24, 45
PCI Command 24
PCI Configuration Cycle 13, 83, 109
PCI Configuration Space 110
PCI Disconnect 45, 46, 47, 58
PCI EPROM 13
PCI I/O Space 21, 113
PCI Interrupt Acknowledge Cycle 103
PCI Master Abort 58, 85
PCI Memory Space 21, 113
PCI Operation 7
PCI Retry 47, 58
PCI Special Interest Group 4, 45
PCI Target Abort 58
PCI Target Disconnect 59
PCI Target Retry 59
PCI Type 0 Configuration Cycle 85
PCI-from-Local Read 10
PCI-to-Local Write 10
PCI-to-PCI Bridge 85
PICU 95, 99
Posted Read 47
PowerPC 11
Prefetch 25, 28, 103
PRST 105, 111

Index

R

\overline{RDY} 73, 74, 135
Read Aperture 28
Read FIFO 31
Read Memory 38
Read Prefetching 17, 22, 25
 \overline{READY} 61, 77, 135
Recovery 58
Register
 DEVICE ID 112
 DMA_CSR0 136
 DMA_CSR1 136
 DMA_CSRx 39, 40, 42, 43
 DMA_CTLB_ADDR0 137
 DMA_CTLB_ADDR1 137
 DMA_CTLB_ADDRx 38, 39, 43
 DMA_LENGTH0 136
 DMA_LENGTH1 136
 DMA_LENGTHx 42
 DMA_LOCAL_ADDR0 135
 DMA_LOCAL_ADDR1 135
 DMA_LOCAL_ADDRx 41
 DMA_PCI_ADDR0 135
 DMA_PCI_ADDR1 135
 EPROM_BASE 86
 FIFO_CFG 29, 46, 47, 129
 FIFO_PRIORITY 130
 FIFO_STAT 131
 LB_BASE0 127
 LB_BASE1 127
 LB_BASEx 23, 25, 103
 LB_CFG 135
 LB_IMASK 97, 98, 133
 LB_IO_BASE 14, 93, 108, 109, 128
 LB_ISTAT 95, 97, 98, 132
 LB_MAIL_IERD 95, 139, 140
 LB_MAIL_IWR 95, 139
 LB_MAP0 128
 LB_MAP1 128

LB_MAPx 23, 24, 103
LB_STAT 102
LB_WR_PCI 49
MAIL_DATA 137
MAIL_RD_STAT 94, 95, 141
MAIL_WR_STAT 94, 95, 140
PC_IO_BASEx 45
PCI_BASE0 120
PCI_BASE1 121
PCI_BASEx 17, 18, 21, 22, 86, 89, 91
PCI_BPARAM 100, 102, 122
PCI_CC_REV 108, 119
PCI_CMD 22, 113, 117, 118
PCI_DEVICE 117
PCI_HDR_CFG 119
PCI_INT_CFG 100, 126, 127
PCI_INT_STAT 95, 100, 102, 125
PCI_IO_BASE 14, 47, 93, 113, 120
PCI_MAIL_IERD 95, 138
PCI_MAIL_IEWR 95, 137, 138
PCI_MAP0 123
PCI_MAP1 124
PCI_MAPx 17, 18, 21, 47, 89, 91
PCI_ROM 122
PCI_STAT 58, 59, 118
PCI_STATUS 85
PCI_SUB_ID 122
PCI_SUB_VENDOR 121
PCI_VENDOR 85, 112, 117
SYSTEM 107, 111, 134

Register Descriptions 115

Register Map 115

REQ 29, 49, 118

Reset 105

ROM 23

S

SCL 111

SDA 111

SERR 117, 118

Index

Special Cycle 24, 46, 49
Swap Mode 22

T

Target 1
Throttling 40
TRDY 10, 45, 46, 47, 49

V

V96SSC 1, 7
VGA 91
VxBMC/CMC 1, 7

W

Wait State 45, 48
Write FIFO 17, 27, 28
Write Memory 38
Write Posting 27

X

x86 93