EZDMA2 IP for Xilinx Hard IP

Reference Manual

Version 1.4.6 December 2012

EZDMA2 IP

Reference Manual

Documentation Change History

Date	Version Number	Changes
December 2011	1.4.6	• Corrected documentation bugs.
December 2011	1.4.5	• Added support for Kintex 7.
January 2011	1.4.4	• Updated interrupts information and Master Module signals. • Added more information about completions.
April 2010	1.4.3	• Updated INT_ACK description.
December 2009	1.4.2	• Updated specification versions.
October 2009	1.4.1	• Corrected documentation bugs.
September 2009	1.4.1	• Added support for Virtex-6 and Spartan-6.
June 2009	1.3.2	• Updated slv_bytevalid signal description. • Added app_dma_fifocnt and dma_fifocnt signals to Scatter-Gather.
March 2009	1.2.2	• Updated DMA Controller signal names.
October 2008	1.2.1	• No change to Reference Manual.
July 2008	1.2.0	• Changed specification standards. • Updated Xilinx Hard IP Interface Signals. • Modified maximum payload, message count, and traffic class.
May 2008	1.1.2	• Updated Xilinx Hard IP Interface Signals
February 2008	1.1.1	• Corrected minor errors in transaction examples
December 2007	1.1	• Updated description of SLAVE BYTE COUNT.
June 2007	1.0	• First release

Proprietary Notice

Table of Contents

List of Tables

List of Figures

Preface

About this Document

This document has been written for design managers, system engineers, and designers who are evaluating or using the PLDA EZDMA2 IP.

Additional Reading

This section lists additional resources from PLDA and third-parties.

PLDA periodically updates its documentation. Please contact PLDA Technical Support or check the Web site at http://www.plda.com for current versions.

PLDA Publications

Please refer to the following documents for further information:

- *EZDMA2 IP Getting Started*: The *Getting Started* guide provides information to enable designers to integrate the PLDA EZDMA2 IP into their design flow as quickly as possible (installing, customizing, integrating, and simulating the Core).
- *Software Tools for PCI/PCI-X and PCI Express IP Cores*: The *Software Reference Manual* describes PLDA's Software Development Kit (SDK).
- *Bus Functional Model Reference Manual*: The *BFM Reference Manual* provides the complete functional description of the PLDA testbench.
- *Build History*: The *Build History* lists changes made to the packaging of each build.
- *Revision History*: The *Revision History* lists changes made to the RTL of the Core.

Other Publications

Please refer to the following documents for information on specification standards:

- *PCI Express™ Base Specification Revision 2.0*
- *LogiCORE IP 7 Series FPGAs Integrated Block v1.1 for PCI Express User Guide UG477*
- *Virtex-6 LogiCORE Integrated Block v1.4 for PCI Express Designs User Guide UG517*
- *Spartan-6 LogiCORE Integrated Block v1.2 for PCI Express Designs User Guide UG654*
- *Virtex-5 LogiCORE Endpoint Block Plus v1.13 for PCI Express Designs User Guide UG341*

Feedback and Contact Information

Feedback about this Document

PLDA welcomes comments and suggestions pertaining to this documentation. Please contact PLDA Technical Support and provide the following information:

- the title of the document
- the page number to which your comments refer
- a description of your comments

Contact information

Corporate Headquarters

PLDA
Parc club du golf - Bât. 11a
Rue Guillibert
13856 Aix-en-Provence Cedex 3 - France

Tel: USA +1 408 273 4528 - International +33 442 393 600
Fax: +33 442 394 902

Sales

For sales questions, please contact sales@plda.com.

Technical Support

For technical support questions, please contact PLDA Support at http://www.plda.com/plda_login.php using the Support Center if you have a PLDA online account.

If you don’t have a PLDA account, contact http://www.plda.com/support_enquiry.php.

Chapter 1 EZDMA2 IP Core Features

PLDA's EZDMA2 IP for Xilinx Virtex-5, Virtex-6, Spartan-6, and Kintex-7 FPGAs is designed for those with little or no experience with PCI Express or experienced designers looking for an easy-to-use yet robust PCI Express 2.0 or 1.1 interfacing solution for Xilinx FPGAs.

The EZDMA2 IP wraps around Xilinx's PCI Express Endpoint Blocks and adds the renowned PLDA EZDMA interface, providing multi-channel DMA capability with scatter-gather support.

The following table describes the features of the EZDMA2 IP Core:

Table 1: Features of the EZDMA2 IP Core

General	• 32, 64, or 128-bit data path • *PCI Express Base Specification Revision 2.0* compliant • Supports up to 32 MSI messages using 64-bit addressing • Supports maximum payload size of 512 bytes • Supports Virtex-5, Virtex-6 and Spartan-6, and 7-Series Integrated Blocks for PCI Express (Gen2).
Customization	• Easy customization with the IP Wizard • Unused features not implemented in silicon
DMA Data transfer	• Supports up to 8 built-in DMA channels and up to 8 external DMA ports • Supports up to 4 GB DMA data transfer • Supports up to 8 outstanding requests

Chapter 2 EZDMA2 IP Architecture

2.1 Architecture of the Master Module DMA Interface

The EZDMA2 IP can implement up to 8 independent DMA channels that may be used simultaneously to manage 8 separate data flows, as illustrated below:

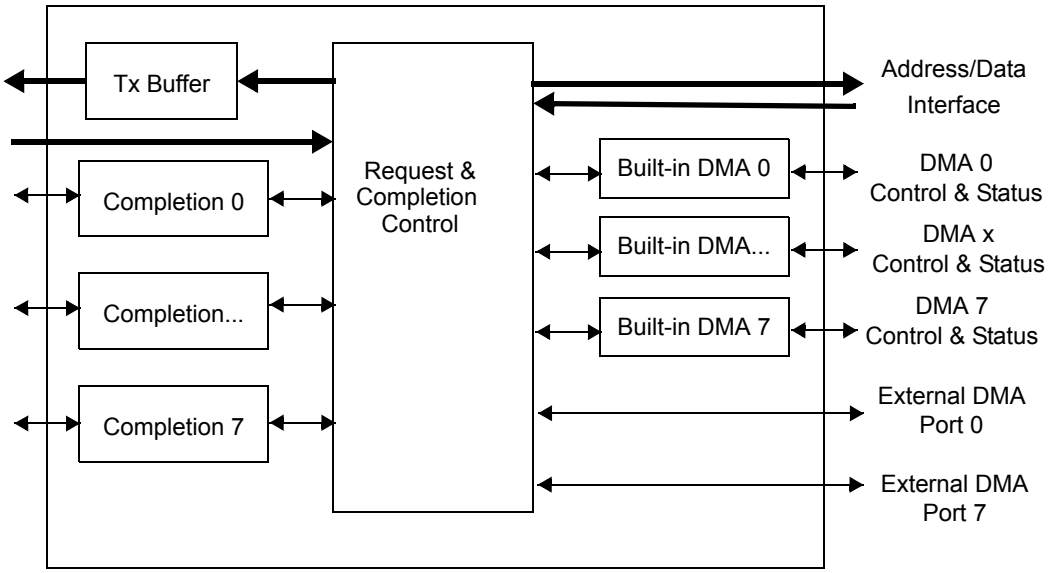

Figure 1: Architecture of the Master Module

Per-channel control and status signals make it possible to program, monitor, and control each DMA channel independently.

The Address/Data Interface is common to all channels and enables data to be read and written through this memory-like interface.

2.1.1 Request & Completion Control

The Request & Completion Control is responsible for:

- Sending transfer requests issued by DMA channels to the Transmit interface of the Hard IP, according to available Completion resources
- Receiving Completions from the Receive interface of the Hard IP and forwarding them to the appropriate channel

2.1.2 Master Request Module

The following figure illustrates the behavior of the Master Request Module:

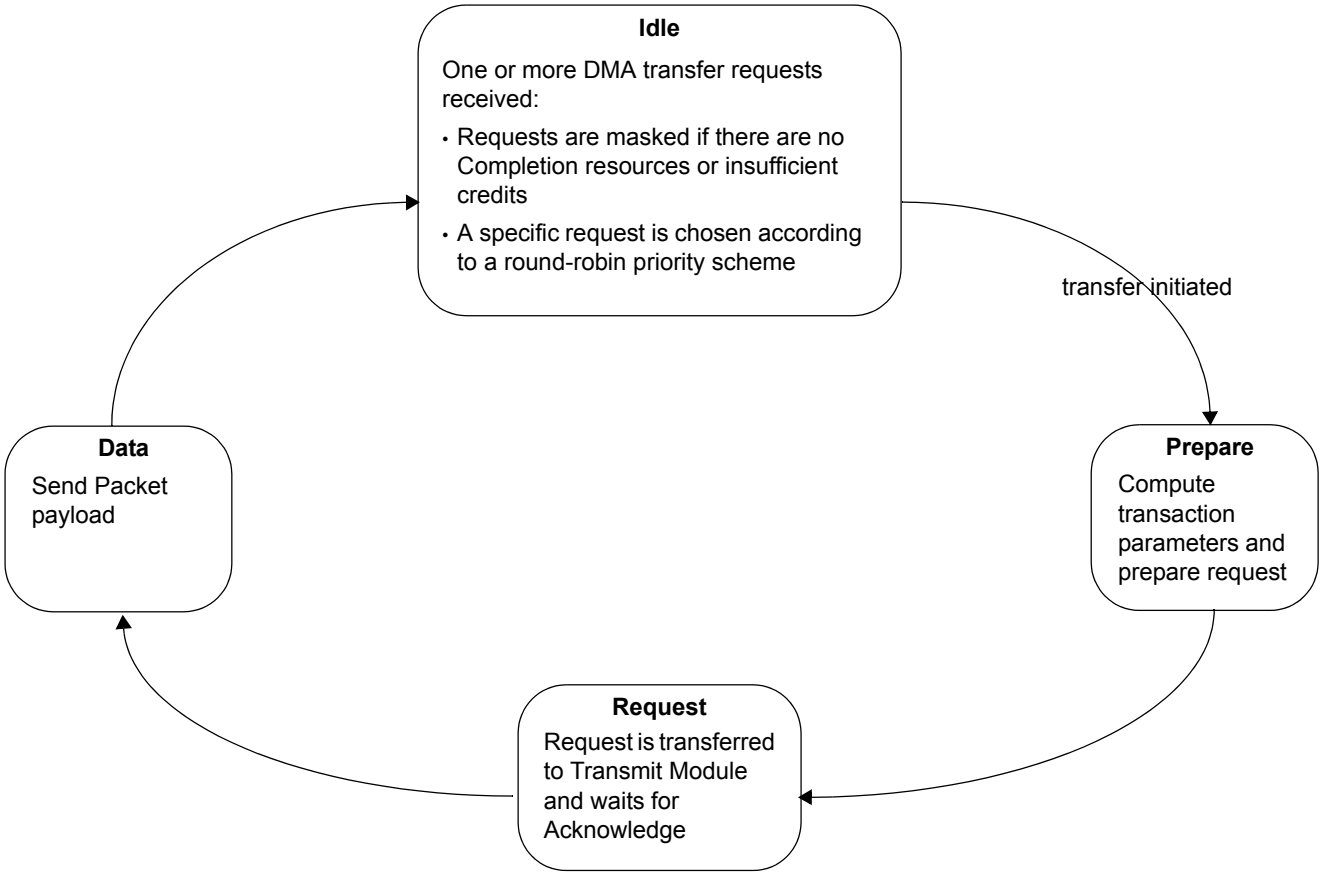

Figure 2: Behavior of Master Request Logic

The DMA Request logic receives and transmits transfer requests to the Transmit Module. When one or more channel requests are detected, the Master Request Module masks requests for which there are no Completion resources or not enough credits. It then selects requests according to a round-robin priority scheme.

Data is read from the local interface without any DMA channel interaction.

2.1.3 Behavior of Master Completion Logic

The following figure illustrates the behavior of Master Completion logic:

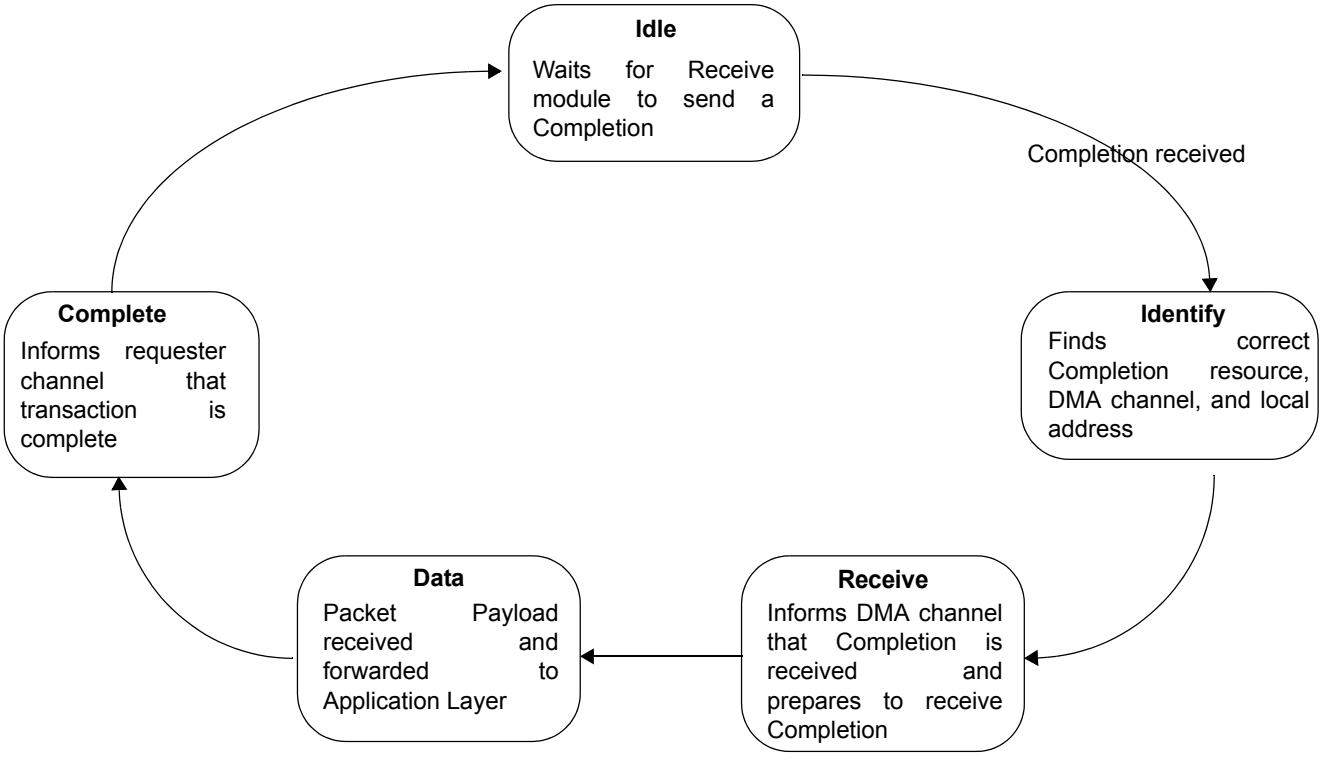

Figure 3: Behavior of Master Completion Logic

Master Completion logic receives Completions from the Receive module. It identifies the appropriate completion resource, DMA channel, and local address for the Completion and informs the requesting channel.

Data is transmitted to the local interface without any DMA channel interaction. The Master Completion Module receives Completions from the Receive module and informs the requesting channel.

2.1.4 Completion Resources

Up to 8 Completion resources manage Non-Posted requests and maintain necessary information to complete the transfer.

A timeout error is signalled if no Completion is received after a given amount of time, and the transfer is aborted.

Each completion resource is assigned a fixed tag, from 00000000 to 00000111. Completion resources are allocated dynamically and several handlers can serve the same DMA channel simultaneously.

2.2 Architecture of the Slave Module

The Slave Module is responsible for:

- Handling I/O, Memory Read, and Memory Write requests
- Decoding TLPs and providing data about the Slave Module
- Sending Completer Abort Completions when necessary

The following figure illustrates the logical flow of the Slave Module.

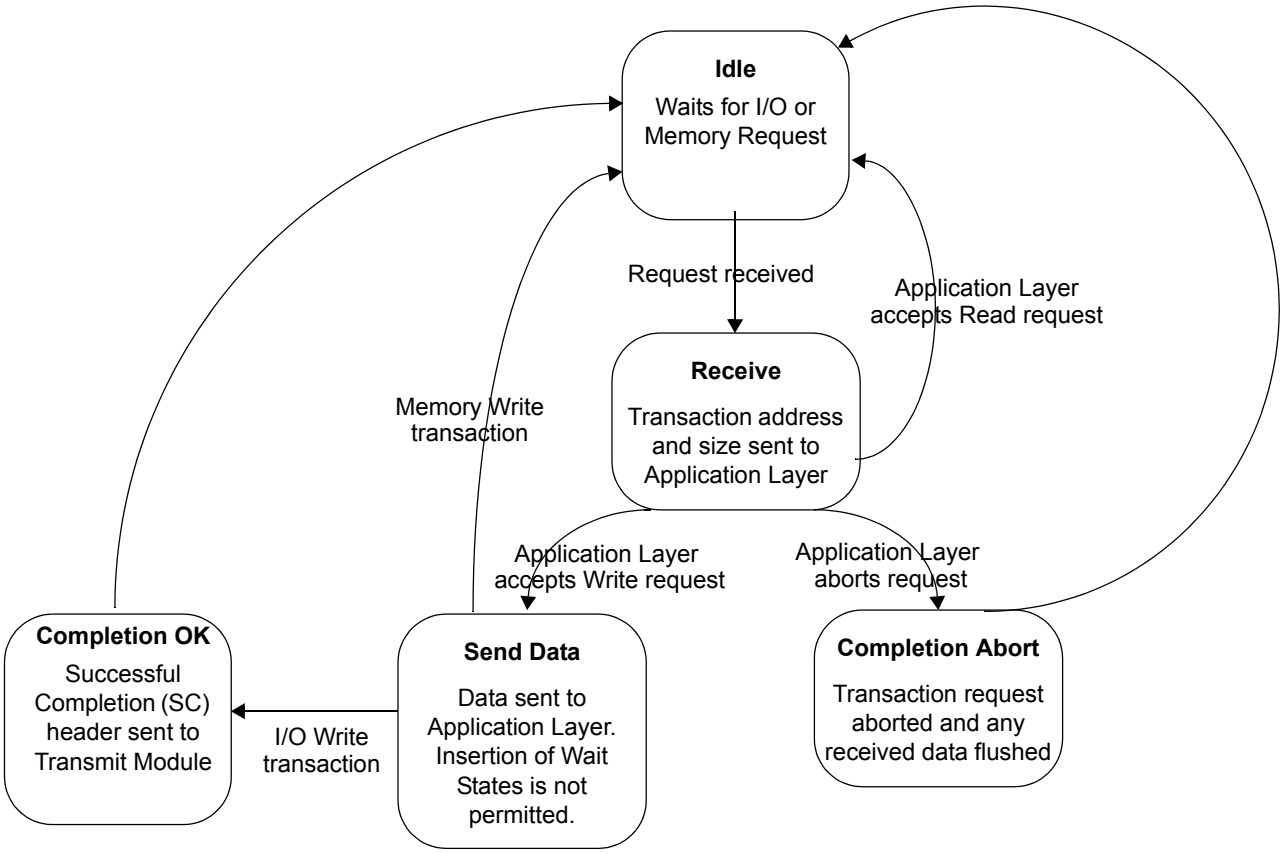

Figure 4: Slave Module Behavior

The Slave Module receives request transactions from the Xilinx Hard IP Core and provides the Application Layer with the start address, targeted BAR (Base Address Register), and size of transfer of a given transaction.

Having received a packet from the Slave Module, the Application Layer determines whether to accept or abort the transaction with the following consequences:

- **Write transaction accepted**: Data is immediately processed and insertion of a wait-state is not permitted. For I/O Write transactions, a successful Completion is automatically sent.
- **Read transaction accepted**: The request is immediately forwarded to the Application Layer, which is responsible for storing necessary information and performing the Completion.
- **Aborted transaction**: Any data is discarded and an Abort Completion is sent automatically, if appropriate.

Note: Application logic is allowed any reasonable amount of time before accepting or aborting an incoming transaction.

There is no rule in the PCI Express specification that indicates a maximum delay for a device to respond to a read request. However, a completion timeout will usually occur if no completion is received within a few milliseconds. Moreover, no other request or completion transactions can be processed by the Receive Module until the incoming transaction is accepted or aborted. This can prevent the Master Module from receiving Completions, thus delaying DMA transfers, and dramatically reducing system performance.

To avoid this, application logic should always try to accept or abort transactions as quickly as possible, and never exceed a few hundred clock cycles to respond.

Completions

- **I/O Write and aborted Read transactions**: I/O Write transactions and aborted Read requests are automatically completed by the Slave Module, which sends a completion packet with appropriate status to the Transmit Module.
- **Accepted I/O and Memory Read transactions**: The Application Layer is responsible for sending Completion transactions in response to I/O and Memory Read requests using a DMA channel:
  1. The Application Layer accepts a transfer with assertion of `SLV_ACCEPT` and stores the values of `SL_BYTECOUNT[]`, `SLV_CPLADDR[]`, and `SLV_CPLPARAM[]`.
  2. The Application Layer programs a DMA channel to perform the Completion.
  3. The Application Layer finishes DMA transfer as with other types of transfers.

Note: You should implement a separate DMA channel for DMA completions that is not used for other types of transfer. Sharing DMA between completions and Read/Write transfers can lead to dead locks.

Example 1: Immediate Programming of DMA Registers

In this example, the Application Layer chooses to immediately program the DMA registers and DMA Parameters, thereby avoiding the need to temporarily store Byte Count, Completion Parameters, and Completion Address values elsewhere.

Note in particular:

- **clock cycle 4**: `DMA_REGIN[]` and `DMA_PARAM[]` can be programmed on the same clock cycle as assertion of `SLV_ACCEPT`.

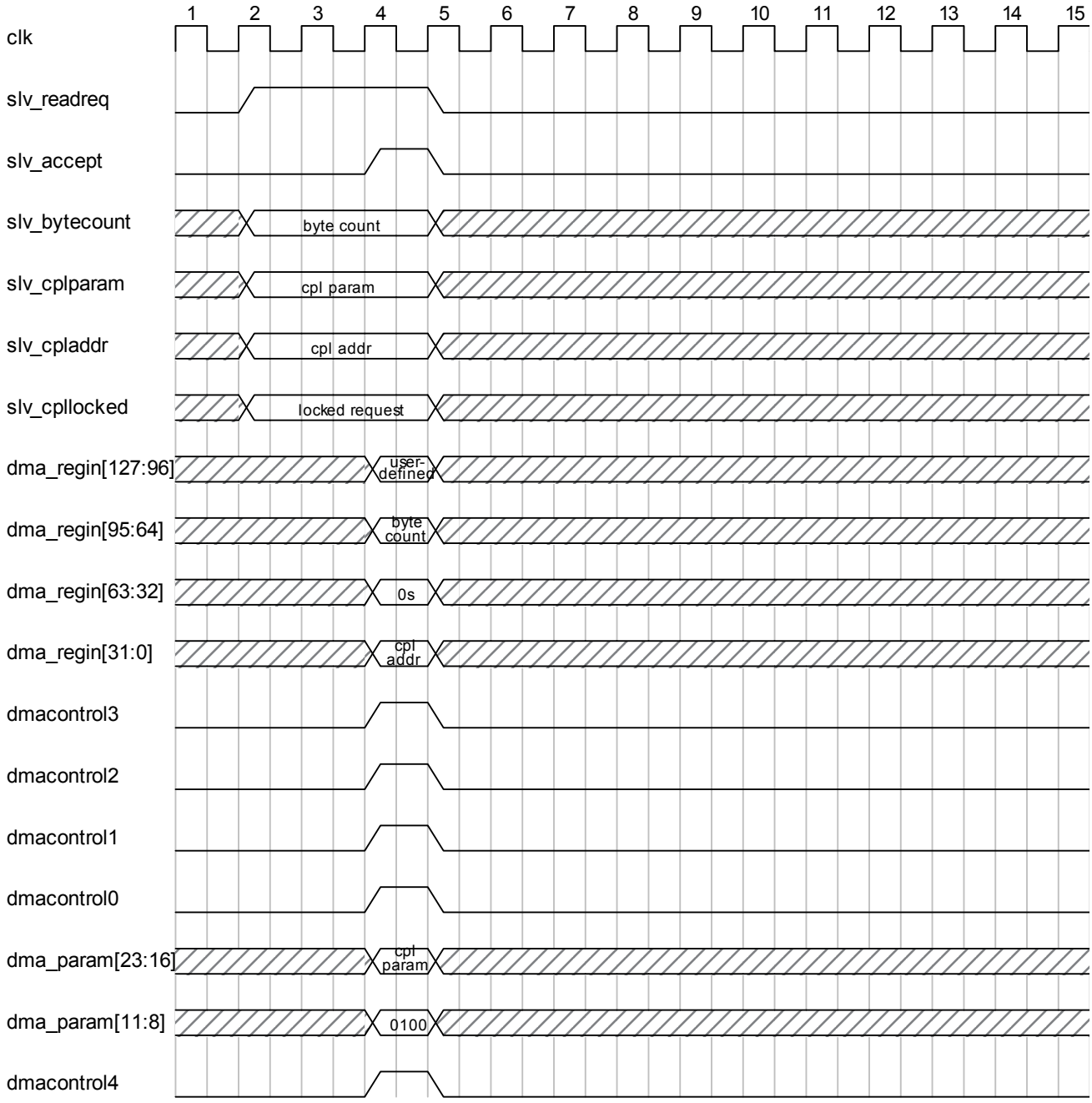

Figure 5: Waveform illustrating immediate programming of the DMA registers

Example 2: Delayed programming of DMA Registers

This example is identical to the previous example, except that the Application Layer chooses to delay programming of the DMA Registers and DMA Parameters by N clock cycles. In this case, the Application Layer is required to temporarily store Byte Count, Completion Parameters, and Completion Address values elsewhere.

Note in particular:

- **clock cycle N+1**: `DMA_REGIN` can be programmed simultaneously or at any time after assertion of `SLV_ACCEPT`.
- **clock cycle N+4**: `DMA_PARAM` can be programmed simultaneously or at any time after `DMA_REGIN` has been programmed.

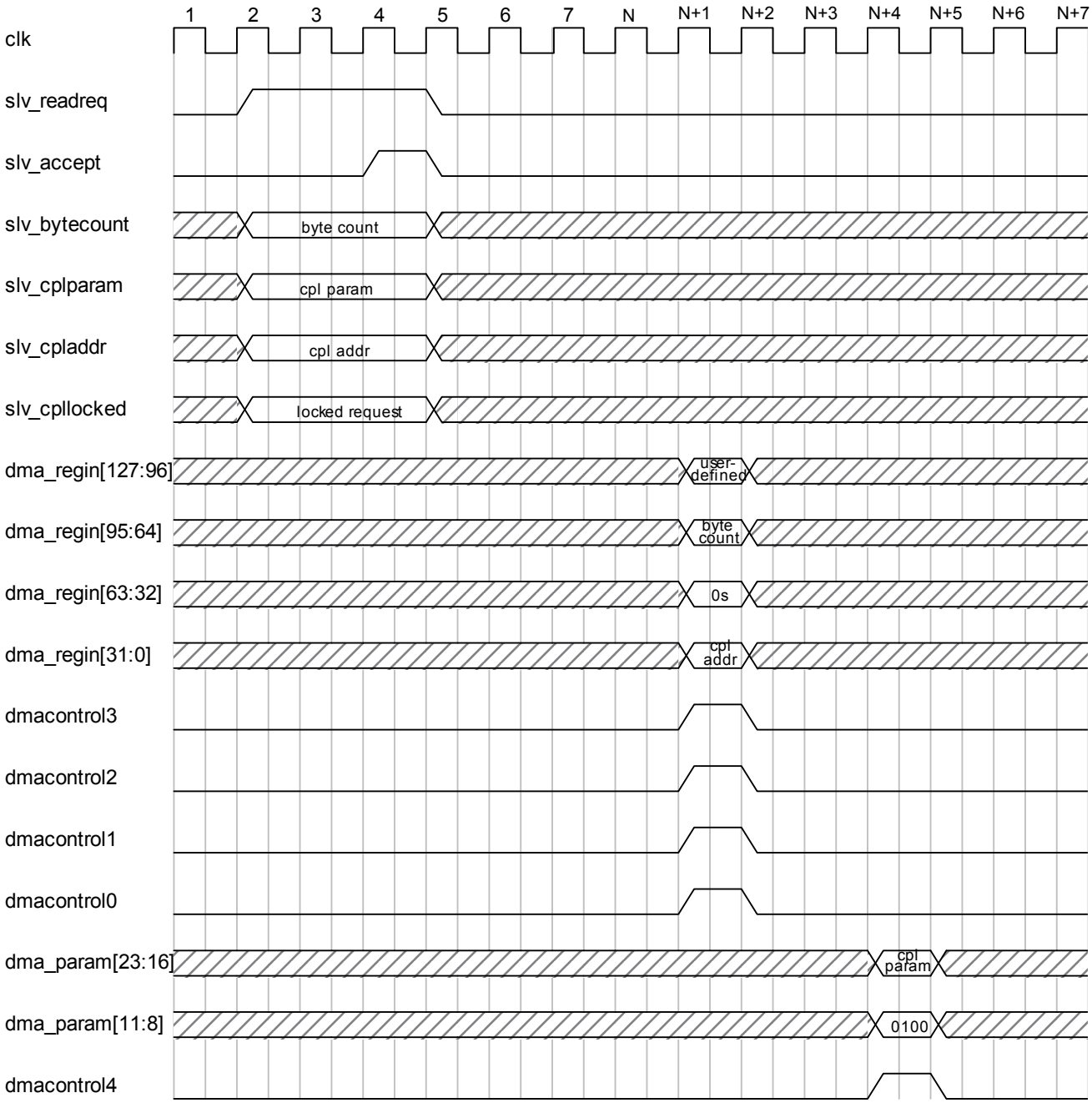

Figure 6: Waveform illustrating delayed programming of the DMA registers

2.3 Configuration Space

The Configuration Space, located within the Transaction Layer, implements all configuration registers and associated functions. See Appendix B: Register Content of the Configuration Space or the *PCI Express Specifications* for the complete content of these registers:

- Type 0 Configuration Space
- PCI Power Management Capability Structure
- Message Signaled Interrupt (MSI) Capability Structure
- PCI Express Capability Structure
- Virtual Channel Capabilities

The configuration space also generates all messages (such as PME#, INT, and Error), MSI requests, and Completion packets from Configuration requests that flow in the direction of the Root Complex, which are generated by a downstream port in the direction of the PCI Express Link. All such transactions are dependent upon the content of the PCI Express Configuration Space as described in the *PCI Express Specifications*.

Chapter 3 Integrating the EZDMA2 IP Core into your Design

This chapter provides information to help you integrate the Core into your design.

Note: Some of the signals described in the following sections may not appear in your implementation of the Core, depending on the configuration choices you make with the EZDMA2 IP Wizard.

The diagram below illustrates how the EZDMA2 IP Core can be integrated with the Application Layer for a typical PCI Express design using **FIFO mode**.

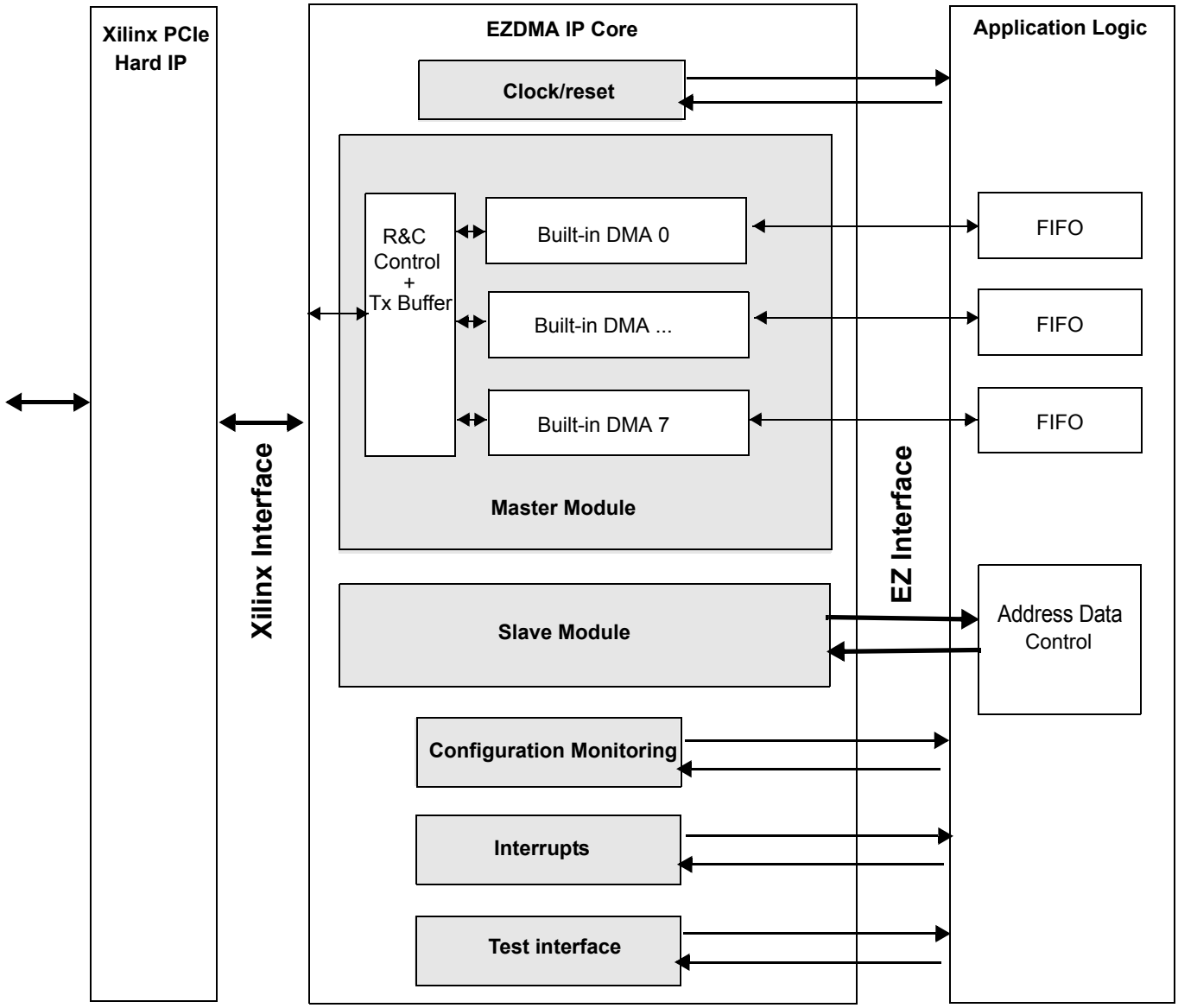

Figure 7: Integration of the Core with the Application Layer

Note: Power Management is not handled by the EZDMA2 IP; you must connect to the Hard IP Core directly in order to use power management features. See Section 3.11: Connecting the EZDMA2 IP to the Xilinx Hard IP for more information.

3.1 EZ Interface

The following table describes signals used to configure the EZDMA2 IP Core. You can set values for these signals using the EZDMA2 IP Wizard.

Table 2: EZ Interface Signals

Signal	I/O	Description
k_ez[63:0]	in	**EZ Interface-specific settings**: • k_ez[5:0]: Completion timeout in ms (3...60) • k_ez[7:6]: reserved • k_ez[15:8]: Implemented bit for each DMA channel • k_ez[25:16]: reserved • k_ez[28:26]: Number of MSI messages (000 =1, ..., 101= 32) • k_ez[31:29]: Designed Max payload size (000 = 128 bytes, ..., 101 = 2KB) • k_ez[37:32]: reserved • k_ez[40:38]: Designed Max read request size (000 = 128 bytes, ..., 010 = 512 bytes) • k_ez[47:41]: reserved • k_ez[63:48]: Implemented bit for each completion resource

3.2 Clocks and Resets

The following table describes the clock and reset signals used by the EZDMA2 IP Core:

Table 3: Clock and Reset Signals

Signal	I/O	Description
clk	in	**Clock**: This signal is the user clock for all Core logic and is provided by the Xilinx Hard IP. See Section 3.11: Connecting the EZDMA2 IP to the Xilinx Hard IP for more information.
rstn	in	**Asynchronous Reset**: This signal is the active-low reset asynchronous signal. This signal resets all Core registers and should be asserted whenever the device is reset.

3.3 Master Module Interface

Note: In the following sections, when dma0_xxx signals are described, these also apply to DMA Channels 1 to 7. For example, for the signal DMA0_REGIN[127:0] described in the table below, there exists an equivalent DMA1_REGIN[127:0], DMA21_REGIN[127:0], etc, for each DMA Channel up to DMA7_REGIN[127:0].

3.3.1 Master Module Signals

The following table describes signals used to communicate between the Application Layer and the Master module of the EZDMA2 IP Core, including status, control, and data signals.

Table 4: Master Application Layer Interface Signals

Signal	I/O	Description
dma0_regin[127:0] (also for DMA 1 - 7)	in	**Built-in DMA Register In**: This signal indicates the values to be written to the Address and Size DMA registers when DMA0_CONTROL[3:0] is asserted.
dma0_regout[127:0] (also for DMA 1 - 7)	out	**Built-in DMA Register Out**: This signal reflects the state of the DMA register. It can be read back in order to check the current channel address, remaining data count, and other settings.
dma0_param[23:0] (also for DMA 1 - 7)	in	**Built-in DMA Parameter**: This signal indicates the value to write to the DMA Parameters register when m_dma0_control[] bit 4 is asserted.

dma0_control[5:0] (also for DMA 1 - 7)	in	**Built-in DMA Control**: This signal is used to program a DMA channel, as follows:
		• dma0_control[0]: write enable for PCI address LSB register
		• dma0_control[1]: write enable for PCI address MSB register
		• dma0_control[2]: write enable for size register
		• dma0_control[3]: write enable for local address register
		• dma0_control[4]: write enable for DMA parameters register
		• dma0_control[5]: used to abort a currently running DMA transfer
		  • If asserted and the channel still has outstanding requests, all requests are handled before the transfer is aborted, otherwise the transfer is immediately aborted.
		  • If asserted, and the current transfer is a Completion, than a Completion with Completion Abort (CA) status is sent and the channel stops

Table 4: Master Application Layer Interface Signals

Signal	I/O	Description

dma0_status[3:0] (also for DMA 1 - 7)	out	**Built-in DMA Channel Status**: This signal reports details about the channel state.
		• 0000: (idle state): Last transfer ended successfully
		• 0001: (idle state): Last transfer was stopped by backend
		• 0010: (idle state): Last transfer ended because of CPL timeout
		• 0011: (idle state): Last transfer ended because of CPL UR error
		• 0100: (idle state): Last transfer ended because of CPL CA error
		• 0xxx: (idle state): reserved
		• 1000: (busy state): Channel is busy processing: DMA channel computes next action
		• 1001: (busy state): Requesting transfer: DMA channel attempts to send a request to the PCI Express Link
		• 1010: (busy state): Processing request:
		  • Read DMA: The DMA channel is waiting for Completion(s).
		  • Write DMA: The DMA channel is waiting for all write data to be posted.
		• 1011: (busy state): Waiting for backend to provide/accept data: DMA is in FIFO mode and `DMA0_FIFOCNT` is not large enough to allow data transfer
		• 1xxx: (busy state): reserved

dma0_fifocnt[12:0] (also for DMA 1 - 7)	in	**Built-in DMA FIFO Count**: This signal indicates how many bytes of data the Application Layer is currently able to provide/accept. This signal is used in FIFO mode only (and can be tied to 0s in RAM mode). If a FIFO device is connected to a DMA channel, then this signal should indicate in real-time how many bytes of data can be read/written to the FIFO. Designs that can always provide/receive data must tie this signal to 4096. In order to meet PCI Express protocol requirements, the DMA channel breaks transfers into smaller PCIe read/write requests when necessary; each time a DMA channel is ready to send a new request, it computes the request size depending on the programmed maximum read request size/ maximum payload size and the current value indicated by DMA FIFO Count. If the DMA FIFO count is smaller than the minimum request size (either the number of bytes to complete a transfer or the number of bytes to reach the next 128-byte address boundary), then transfer is paused until the DMA FIFO count value is large enough.
dma_rd	out	**DMA Read**: This signal is asserted when the Master module reads data from the Application Layer, during which time the Application Layer provides data located at the address specified by `DMA_ADDR[]` on `DMA_DATA_IN[]`.
dma_rdaddr[31:0]	out	**DMA Read Address**: This signal indicates the local address from which data should be read. The specified data must be present on `DMA_RDDATA[]`. **Note**: You can specify a value between 13 and 32 bits for the size of this signal using the Wizard. See the *Getting Started* for more information.
dma_rdchannel[15:0]	out	**DMA Read Channel**: This signal indicates which DMA channel is currently reading data: • Bits[7:0]: Built-in DMA channel 7..0. • Bits[15:8]: External DMA port 7..0.

Table 4: Master Application Layer Interface Signals

Signal	I/O	Description
dma_rddata[31:0]/ [63:0]/[127:0]	in	**DMA Read Data**: This signal is the data bus that provides data for DMA Write transactions. Data must be naturally aligned on byte lanes as follows: With a 32-bit data path: • Bits[7:0]: Data for addresses similar to xx00 • Bits[31:24]: Data for addresses similar to xx11 With a 64-bit data path: • Bits[7:0]: Data for addresses similar to xx000 • Bits[63:56]: Data for addresses similar to xx111 With a 128-bit data path: • Bits[7:0]: Data for addresses similar to x0000 • Bits[127:120]: Data for addresses similar to x1111
dma_wr	out	**DMA Write**: This signal is asserted when the Master Module writes data to the Application Layer. Application Layer logic stores current data on `DMA_WRDATA[]` at the address specified by `DMA_WRADDR[]`. Valid bytes are indicated by `DMA_WRBYTEVALID[]`.
dma_wraddr[31:0]	out	**DMA Write Address**: This signals indicates the location at which data present on `DMA_WRDATA[]` should be written. **Note**: You can specify a value between 13 and 32 bits for the size of this signal using the Wizard. See the *Getting Started* for more information.
dma_wrchannel[15:0]	out	**DMA Write Channel**: This signal indicates which DMA channel is currently writing data: • Bits[7:0]: Built-in DMA channel 7..0. • Bits[15:8]: External DMA port 7..0.
dma_wrdata[31:0]/ [63:0]/[127:0]	out	**DMA Write Data**: This signal is the data bus for receiving data for DMA Read transactions. Data must be naturally aligned on byte lanes. `DMA_WRBYTEVALID[]` indicates which byte lanes are valid.
dma_wrbytevalid[3:0]/ [7:0]/[15:0]	out	**DMA Write Byte Valid**: This signal indicates which bytes of `DMA_WRDATA[]` are valid during transmission: With a 32-bit data path: • dma_wrbytevalid[0] enables data bits 3:0 With a 64-bit data path: • dma_wrbytevalid[0] enables data bits 7:0 • dma_wrbytevalid[7] enables data bits 63:56 With a 128-bit data path: • dma_wrbytevalid[15:8] enables data bits 127:120

Table 4: Master Application Layer Interface Signals

Signal	I/O	Description

edma_maxmask[9:0]	out	Current maximum payload size & maximum read request information for DMA channels:
		• Bits [9:5]: maximum read request size mask:
		  • 11111 = 128 bytes
		  • 11110 = 256 bytes
		  • 11100 = 512 bytes
		  • 11000 = 1KB
		  • 10000 = 2KB
		  • 00000 = 4KB
		• Bits [4:0]: maximum payload mask
		  • 11111 = 128 bytes
		  • 11110 = 256 bytes
		  • 11100 = 512 bytes
		  • 11000 = 1KB
		  • 10000 = 2KB
		  • 00000 = 4KB

edma0_req (also for DMA 1 - 7)	in	**Transfer Request**: This signal is asserted to request the Core to perform a transfer. The signals `EDMA_NDESC`, `EDMA_ARBHINT`, and `EDMA0_LOCADDR` must be valid and stable as long as this signal is asserted. The `EDMA0_CTRL(0)` signal is asserted for one clock cycle to acknowledge the request; the request must be deasserted on the following clock cycle.
edma0_desc[127:0] (also for DMA 1 - 7)	in	**Request TLP Header Information**: • Bits [127:96]: TLP header first DWORD. • Bits [95:94]: DMA read latency on local interface. • Bits [93]: Set if request length is 4KB, otherwise cleared. • Bits [92:83]: Reserved, must be 0s. • Bits [82:80]: Requester function number. • Bits [79:64]: TLP header second DWORD bits [15:0]. • Bits [63:32]: TLP header fourth DWORD. • Bits [31:0]: TLP header third DWORD.

edma0_arbhint[12:0]
(also for DMA 1 - 7)
	in	**Information for DMA Arbiter**:
		• Bits [12:11]: Indicate packet type:
		  • 00 = non posted
		  • 10 = posted
		  • 11 = completion
		• Bits [10:0]: reserved

edma0_locaddr[31:0] (also for DMA 1 - 7)	in	**Local Address**: Indicates the local address at which transfer starts.

Table 4: Master Application Layer Interface Signals

Signal	I/O	Description

edma0_ctrl[6:0]
(also for DMA 1 - 7)
	out	**DMA Control**:
		• Bit [0]: Request acknowledge: asserted for one clock cycle to indicate that the transfer request has been taken into account
		• Bit [1]: Write request sent: asserted for one clock cycle after a write request has been acknowledged, in order to indicate that all data has been read from the local interface.
		• Bit [2]: Completion received: indicates that a completion has been received
		• Bit [3]: Completion timeout: indicates that no or only partial completion has been received after the timeout delay
		• Bits [6:4]: Completion status: reports completion status code when bit 2 is asserted
		  • 000=successful
		  • 001=Unsupported request
		  • 010=completer retry status
		  • 100=completer abort

3.3.2 Transaction Examples using Master signals

Example 3: Typical transfer of a Read request

The waveform below illustrates a typical transfer of a Read request:

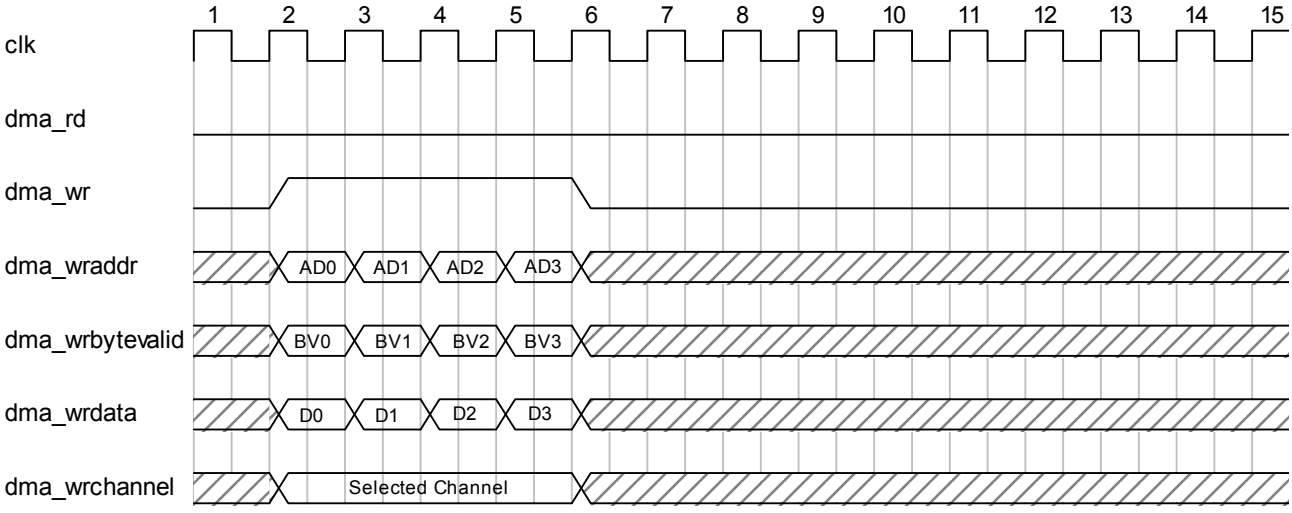

Figure 8: Waveform illustrating a typical transfer of a Read Request

Data read from the EZDMA2 IP Core is written to the local interface using dma_wr... signals.

The Application Layer may not insert wait states and must store data immediately.

Example 4: DMA Write transaction in RAM mode with 1 clock cycle latency

In this example, the Core transmits a 4 Data Phase DMA Write transaction.

Note in particular:

- **clock cycle 3**: Data transfer doesn't begin until clock cycle 4 due to system latency.

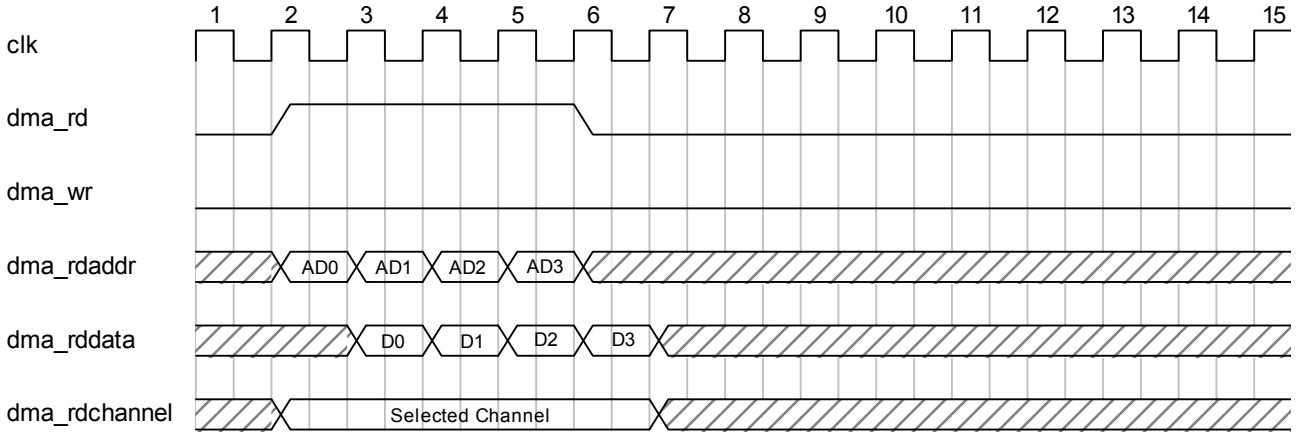

Figure 9: Waveform illustrating 4 Data Phase DMA Write transaction with 2 clock cycles latency

In order to prepare PCI Express Write transactions, the DMA Control logic reads data from the local interface using dma_rd... signals.

In this example, the Application Logic has programmed DMA latency to 00b, which means the address is presented one clock cycle in advance.

Example 5: DMA Write transaction with 2 clock cycles latency

This example is identical to the one above except that data transfer is delayed for 2 clock cycles instead of 1.

In this example, the Core transmits a 4 Data Phase DMA Write transaction.

Note in particular:

- **clock cycle 4**: Data transfer doesn't begin until clock cycle 4 due to system latency.

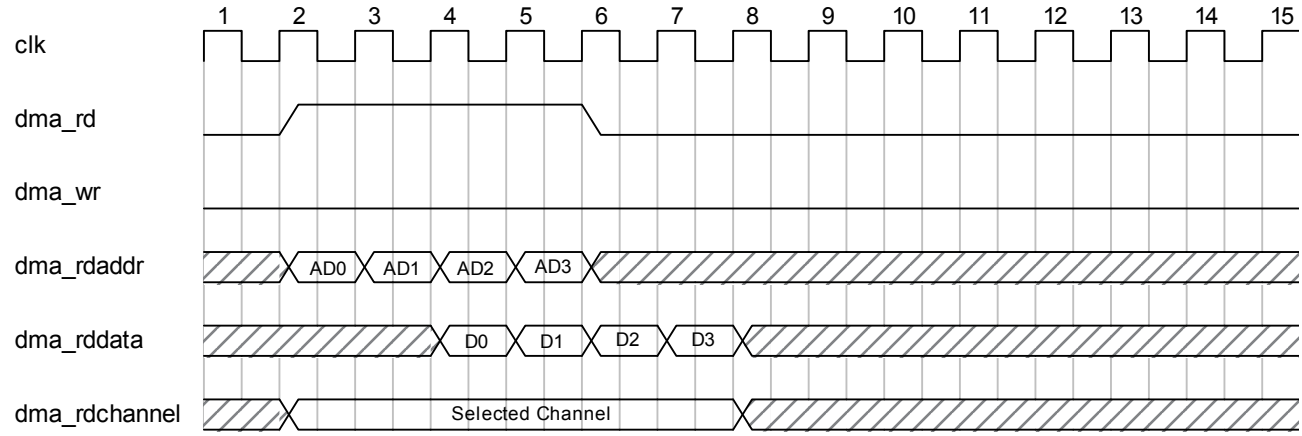

Figure 10: Waveform illustrating 4 Data Phase DMA Write transaction with 2 clock cycles latency

In order to prepare PCI Express Write logic, the DMA Control reads data from the local interface using dma_rd... signals.

In this example, the Application Logic has programmed DMA latency to 01b, which means the address is presented 2 clock cycles in advance.

Example 6: Writing to and Reading from the DMA Registers

This example illustrates writing to and reading from the DMA registers.

Note in particular:

- **clock cycle 3**: `DMA_CONTROL0` must be asserted in order for Address 1 to be written to the DMA register.
- **clock cycle 5**: `DMA_REGOUT[31:0]` now contains Address 1.

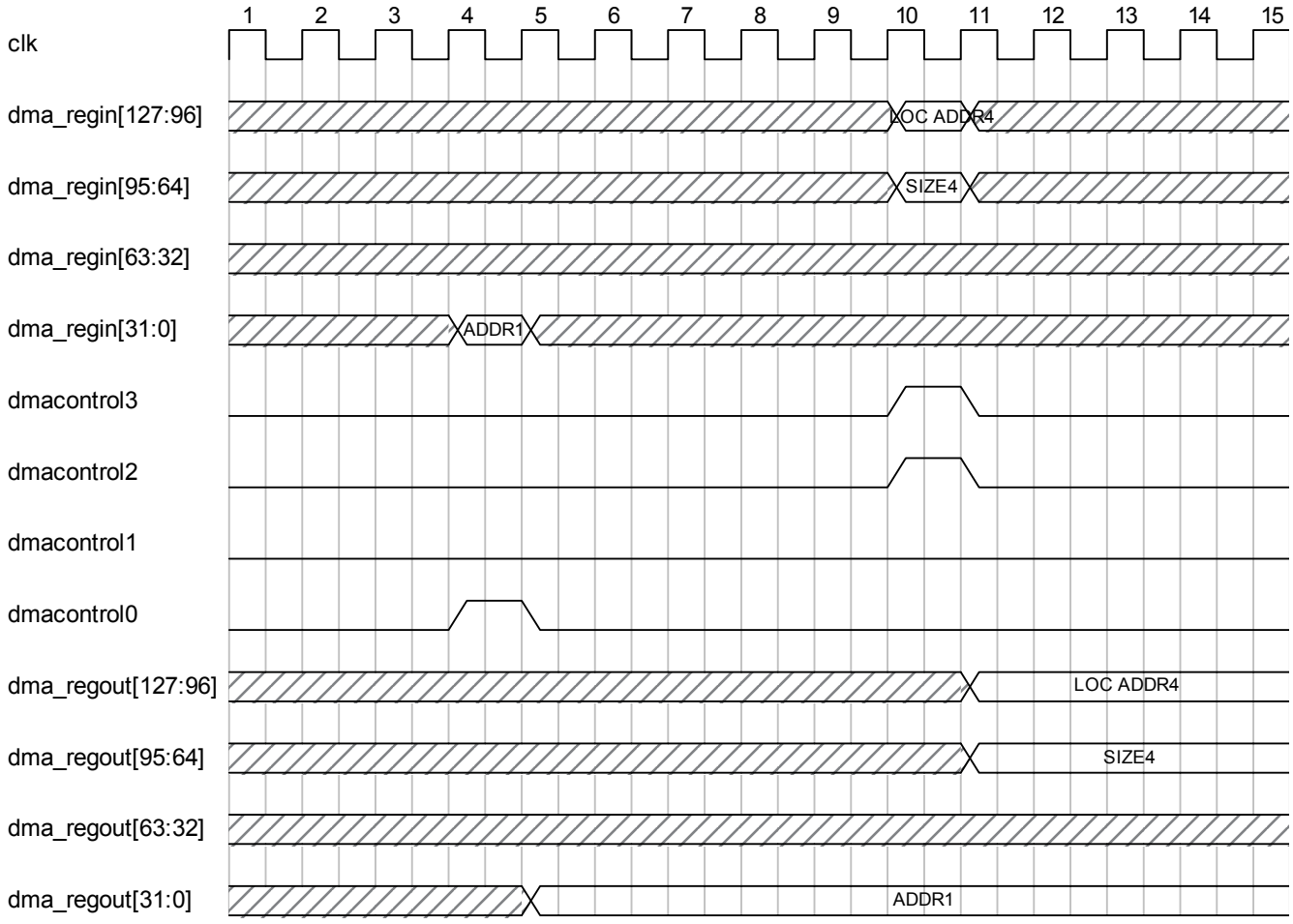

Figure 11: Waveform illustrating Writing to and Reading from the DMA registers

3.4 Setting DMA Channel Registers

The DMA registers, which are programmed by the Application Layer, are illustrated below:

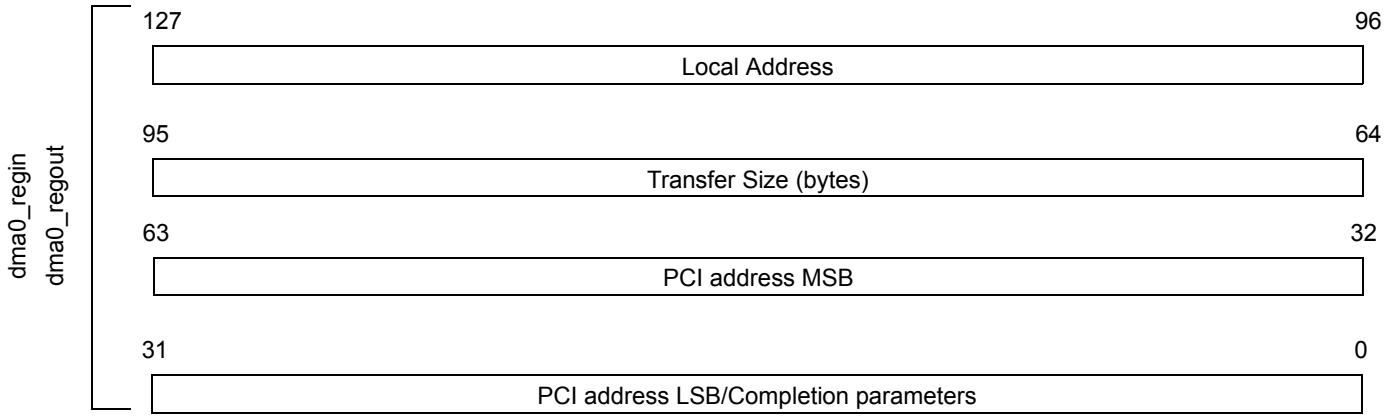

Figure 12: DMA registers

The following table describes each field of the DMA Channel Register:

Table 5: DMA Channel Register Fields

Name	Description
PCI Address[63:0]	**If the transfer is a completion, then**: the MSB field must be 0, and the LSB field must be programmed with completion parameters obtained from slv_cpladdr[]. **If the transfer is not a completion, then**: the Application Layer programs the PCI start address of a DMA transfer in this register. The MSB indicates addressing as follows: • if MSB is 0: 32-bit addressing used • if MSB is not 0: 64-bit addressing is used (if DMA is a memory transfer) Note that a transfer must not cross a 4GB addressing boundary. The address is automatically incremented and indicates the address of the last requested data + 1. On successful completion of a transfer, this register should contain the value: pci_start_address + transfer_size.
Transfer Size[95:64]	This register is used by the Application Layer to record the total transfer size (in bytes) for burst transfers. The value of this register must be programmed to 4 bytes for DW transfers. The size is automatically decremented and this register indicates the remaining data size that can be requested. Upon successful completion of a DMA burst transfer, the value of this register should be 0.
Local address[127:96]	This register is used by the Application Layer to record the local start address of a DMA transfer. The local address is automatically incremented during RAM mode transfers and indicates the address of the last transferred data + 1. Upon successful completion of a transfer, the value of this register should be local_start_address + transfer_size. Note that the local address must be aligned with the PCI address (bits 2:0 should be identical). The value of this address is 0 in FIFO mode.

3.4.1 Setting DMA Channel Options

The EZDMA2 IP Wizard enables you to set the Local Address size, the maximum DMA transfer size, and the number of Outstanding Requests, allowing you to choose between increased system performance or reduced Core logic.

3.4.1.1 Local Address Size

The Local Address Size register can be set with the EZDMA2 IP Wizard to between 13 and 32 bits.

If, for example, the Application Layer only needs 23 bits to map all peripherals connected to the DMA channels, this signal can be set to 23. This ties unused bits of the Local Address Size register to 0, thus saving logic.

Note that a design that uses DMA exclusively in FIFO mode should not need to use the Local Address register, and this setting can be set to its minimum value.

3.4.1.2 Maximum DMA Transfer Size

The Transfer Size register can be set with the EZDMA2 IP Wizard to between 8KB - 1 (13 bits) and 4GB - 1 (32 bits). For example, if the Application Layer never performs DMA transfers of more than 1MB, this signal can be set to 2MB - 1 (21 bits). This ties unused bits of the DMA Transfer register to 0, thus saving logic.

3.4.1.3 Number of Outstanding Requests

The number of Outstanding Requests implemented in a design can be set with the EZDMA2 IP Wizard to between 1 and 16. Outstanding Requests are only necessary for Non Posted (Memory Read and I/O) DMA transfers. The more Outstanding Requests in a design, the greater the potential performance. However, this will involve increased logic. (See the FAQ Article entitled “About DMA Mode” for more information.)

The following figure illustrates three example Outstanding Request configurations.

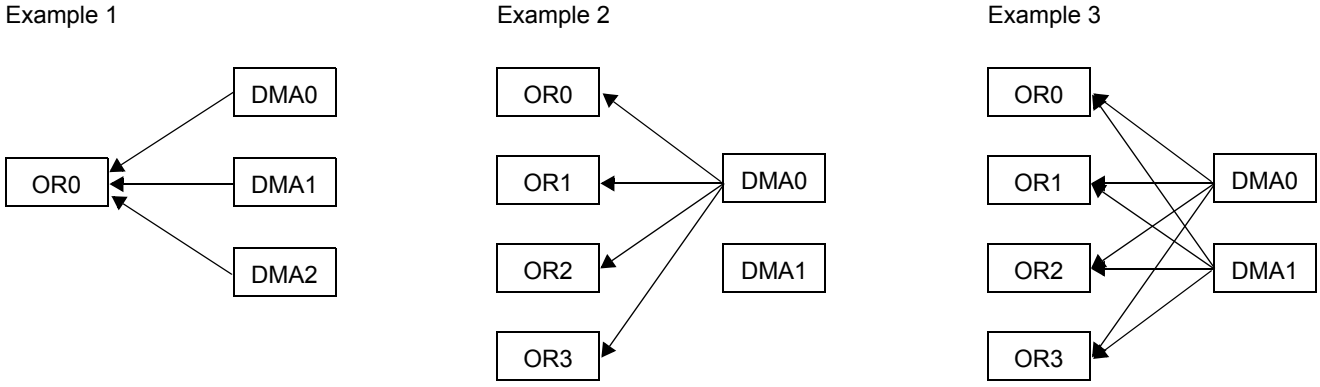

Figure 13: Potential Outstanding Requests / Channel configurations

The following table describes the examples illustrated above:

Table 6: Description of Outstanding Request examples

Example	Description
1	You may choose to implement one Outstanding Request if: • Your system only performs Posted (Memory Write) DMA transactions. • System performance is not a high priority.
2	In this example, DMA1 performs Write transfers and has no need for any Outstanding Requests. DMA0 performs Read transfers and can have up to 4 outstanding requests at any given time.
3	In this example, both DMA0 and DMA1 perform DMA Read and Write transfers. Each DMA channel can have from 1 to 4 outstanding requests at any given time, depending on the number of Outstanding Requests already used by the other DMA channel.

3.4.2 Address/Data Interface

Data for DMA transfers is read from or written to the registers through a local interface based on a designer-defined 32-bit addressing scheme and simple read/write controls.

Write transfers:

- data and the data's location address are presented simultaneously
- the Application Layer can pipeline Write cycles if necessary

Read transfers: The start address is presented 0 to 3 clock cycles before data. This latency is programmed for each DMA channel and permits maximum flexibility in order to interface with a wide range of peripherals.

Note that data for transfers programmed in RAM mode can be received in a non-linear order.

3.4.3 DMA Channel Control

DMA channels are controlled by the Channel Controller Module:

- Each channel can have up to 8 outstanding requests. In FIFO mode, however only one request is permitted at a time (in which case the channel waits for a Completion before making another request).
- A channel must wait for all Completions before stopping if all requests have been issued, the Application Layer stops DMA, or an error is detected.

The following figure illustrates behavior of the DMA Channel Control:

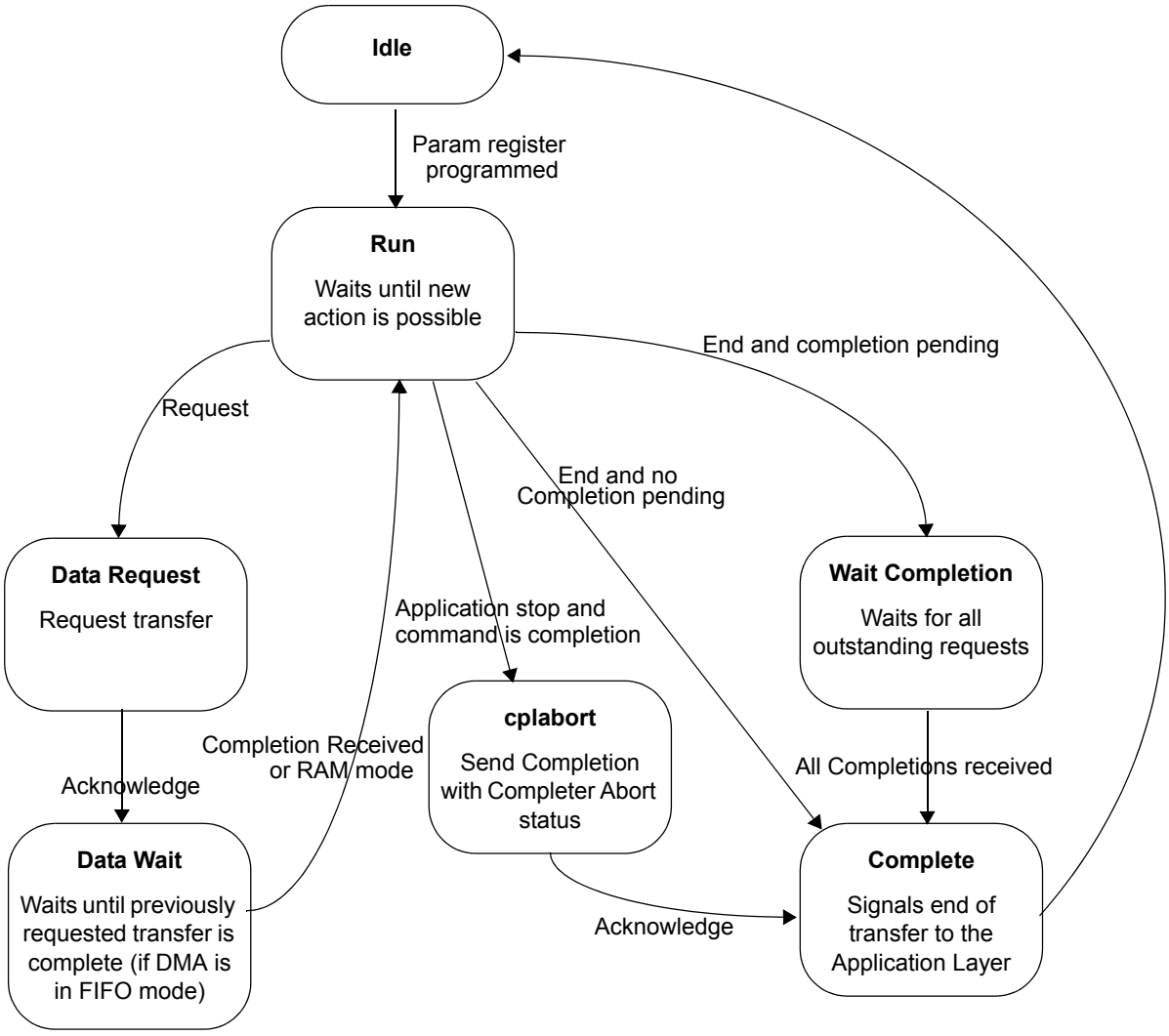

Figure 14: DMA Channel control behavior

3.5 Setting DMA Parameters

The DMA Parameters, which are programmed by the Application Layer, are illustrated below:

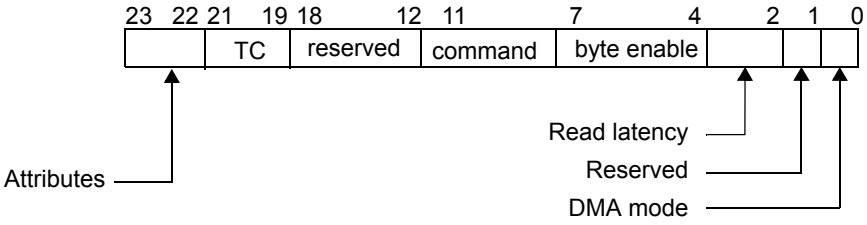

Figure 15: DMA Parameters

The following table describes each field of the DMA Parameters:

Table 7: DMA Channel Parameters

Name	Description
DMA Mode[0]	**FIFO Mode** (bit set to 0): In this mode, only one PCI Express request is permitted at a time. `DMA_FIFOCNT[]` determines whether a transfer is possible. Data is always received in order and a FIFO can be safely used. **RAM Mode** (bit set to 1): In this mode, multiple PCI Express requests are permitted at a time and the Application layer must be constantly ready to provide/accept all data. Read DMA data may be received in any order. A FIFO may not be safely used. See Section 3.5.1: Connecting the DMA Module to the Application Layer for more information.
Reserved[1]	Bit 1 of the DMA register is reserved and must be tied to 0.
Read Latency[3:2]	This register indicates the number of clock cycles the Application Layer needs to provide data to `DMA_ADDR[]` after a read is requested on the local interface by `DMA_READ`. • 00: data is available on next clock cycle • 01: data is available 2 clock cycles later • 10: data is available 3 clock cycles later • 11: data is available 4 clock cycles later **Note:** By default, the maximum supported read latency is 3. A larger read latency requires larger buffering logic. You can set a lower read latency value (2,1 or 0) to decrease logic usage if your application does not require a large latency.
Byte Enable[7:4]	The Application Layer programs this register with the appropriate byte-enable for DW transfers. The value of this register is ignored for burst transfers.
Command[11:8]	The Application Layer programs this register with a DMA command in order to specify which PCI Express command should be used for a transfer. See Section 3.5.1: Connecting the DMA Module to the Application Layer for more information.
Reserved[18:12]	--
TC[21:19]	Traffic class for a transfer.
Attributes[23:22]	• Bit 23 indicates if relaxed ordering is enabled for a transfer. The default is 0. • Bit 22 indicates if No-Snoop is enabled for a transfer. The default is 0.

3.5.1 Connecting the DMA Module to the Application Layer

3.5.1.1 Connecting RAM Devices to DMA

The following figure shows the connection of a RAM or RAM-like device to DMA:

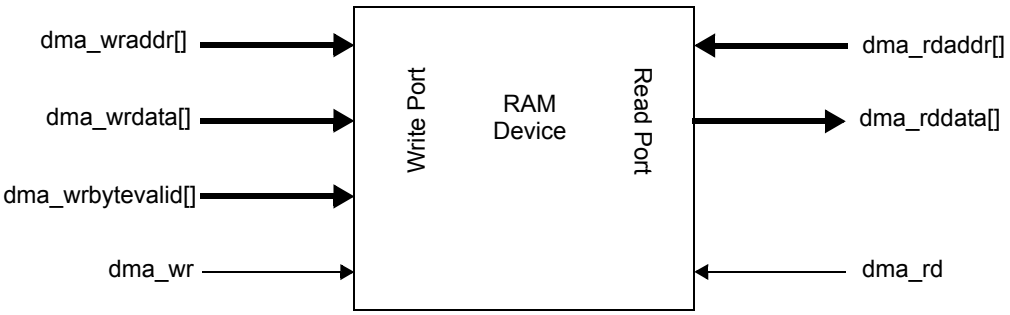

Figure 16: RAM connection to the DMA interface

When programming a DMA channel, the read latency is adjusted depending on the number of register levels between the Read Address port and the DMA Data Input port.

3.5.1.2 Connecting FIFO Devices to DMA

The following figure shows the connection of a FIFO device to DMA:

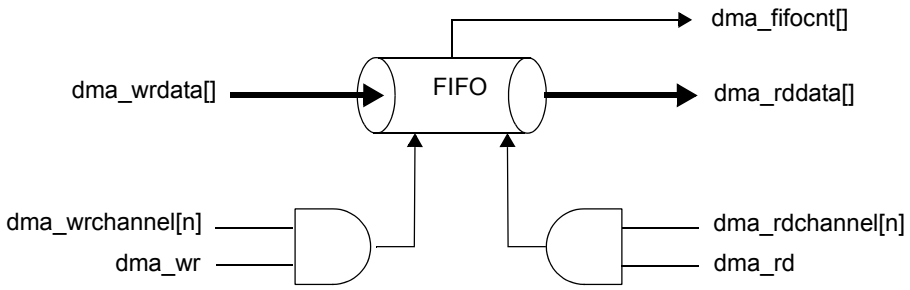

Figure 17: FIFO connection to the DMA interface

Do not use FIFO devices in DMA-RAM Mode as data can arrive in a non-linear order.

You must adjust read latency when programming DMA channels to take into account the number of register levels between the Read Enable port and the DMA Data Read port.

Note: In FIFO mode, performance can be significantly lower than that of RAM mode, and may provide less than 20% of RAM mode throughput.

3.5.2 DMA Commands

- **DW commands** have the following characteristics:
  - Restricted to a single data phase
  - The Transfer Size register is ignored after a transfer
  - Byte-enable bits specified in the Byte Enable register are used
- **Burst commands** have the following characteristics:
  - Can be any size from 1 byte to 2^{32} - 1 bytes
  - The Address register is automatically incremented after a transfer
  - The appropriate byte-enable bits are automatically computed by the DMA engine for all other commands

The following table describes DMA Commands:

Table 8: DMA Commands

Command	Code	Type	Allowed Transfer Size	Allowed Value for TC & Attributes
I/O read	0000	DW	4 bytes	0s
I/O write	0001	DW	4 bytes	0s
Memory read DW	0010	DW	4 bytes	any
Memory write DW	0011	DW	4 bytes	any
Completion with data	0100	Burst	same as request	same as request
reserved	0101			
Memory read burst	0110	Burst	any	any
Memory write burst	0111	Burst	any	any
reserved	1xxx			

3.6 Implementing Scatter-Gather DMA

The EZDMA2 IP default DMA Transfer mode is Direct DMA Transfer. However, you can implement Scatter-Gather DMA mode using the external Scatter-Gather Controller module. You can use Scatter-Gather DMA Transfer to perform large data transfers to a buffer that is fragmented in system memory.

3.6.1 Direct and Scatter-Gather DMA Transfer

In **Direct DMA Transfer** mode, the DMA start address is a pointer to a contiguous data buffer mapped in the PCI bus address space. Data is read to and written from the buffer in a sequential order:

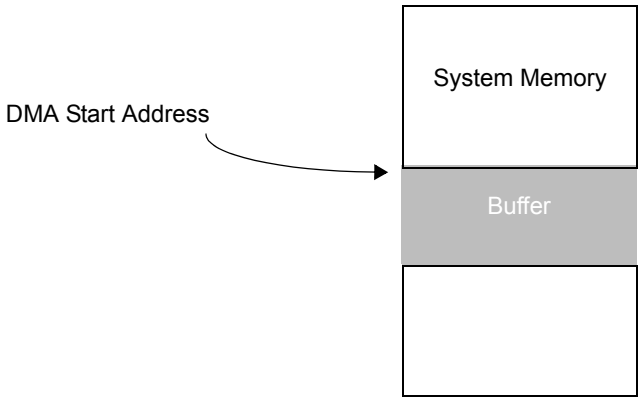

Figure 18: Direct DMA Transfer

In **Scatter-Gather DMA Transfer** mode, the DMA start address is a pointer to a chained list of page descriptors. Each descriptor contains the address and size of a data block (page), as well as a pointer to the next descriptor block to enable circular buffers:

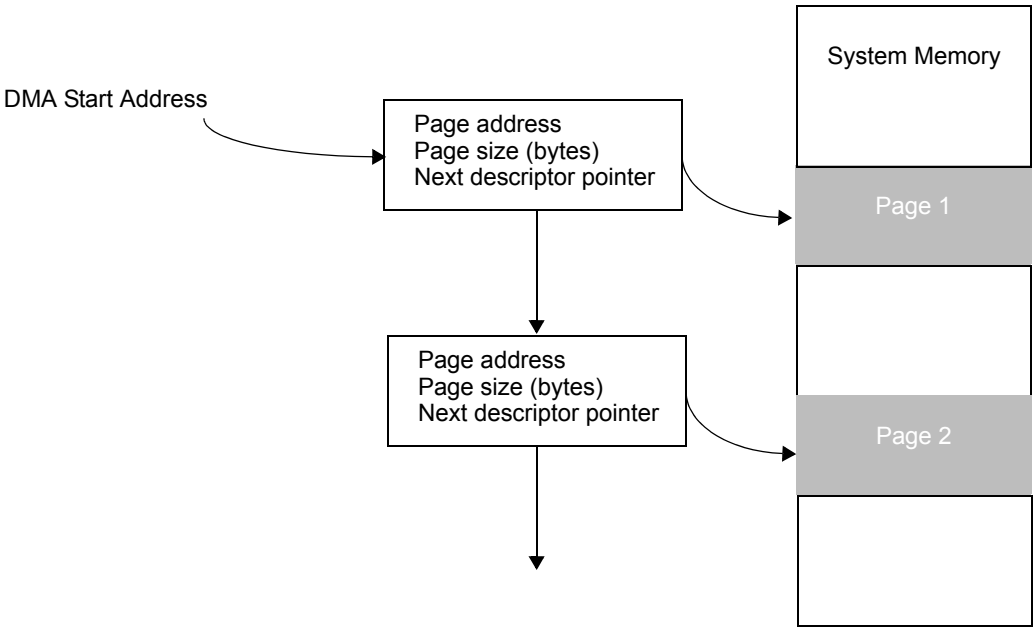

Figure 19: Scatter-Gather DMA Transfer

Note: Scatter-Gather DMA mode is independent of either RAM or FIFO mode.

The following table describes a sample Scatter-Gather implementation using descriptors blocks of 20 bytes:

Table 9: Receive Buffer Memory Resources

Offset	Field	Description
00h	Page address [31:3]	Indicates start address of memory page that must be aligned on an 8-byte boundary (bits 2:0 must be "000") for the 64-bit data path, or a 16-byte boundary for the 128-bit data path: • If a page is located in the 32-bit addressing space, then bits [63..32] must be set to 0. • If a page is located in the 64-bit addressing space, then a full 64-bit address must be initialized. Note that a page must not cross a 4GB address boundary.
04h	Page address [63:32]	
08h	Page Size	Indicates the size of the memory page in units of bytes. This value must be a multiple of 8 bytes .
0Ch	Next descriptor pointer [31:2] & end of chain bit	Specifies the address of the next page descriptor, which must be aligned on a 4-byte boundary (bits 1:0 must be 00). Setting bit 0 to 1 indicates that the current descriptor is the last descriptor in the chain and the DMA engine will not attempt to fetch other descriptors.
10h	Next descriptor pointer [63:32]	

3.6.2 Scatter-Gather Controller

The Scatter-Gather Controller module is an add-on to the EZDMA2 IP Core that is inserted between the Application Logic and each DMA channel that requires Scatter-Gather capabilities.

The following figure illustrates a sample implementation in which DMA0 and DMA1 have scatter-gather capabilities and DMA2 functions in direct mode:

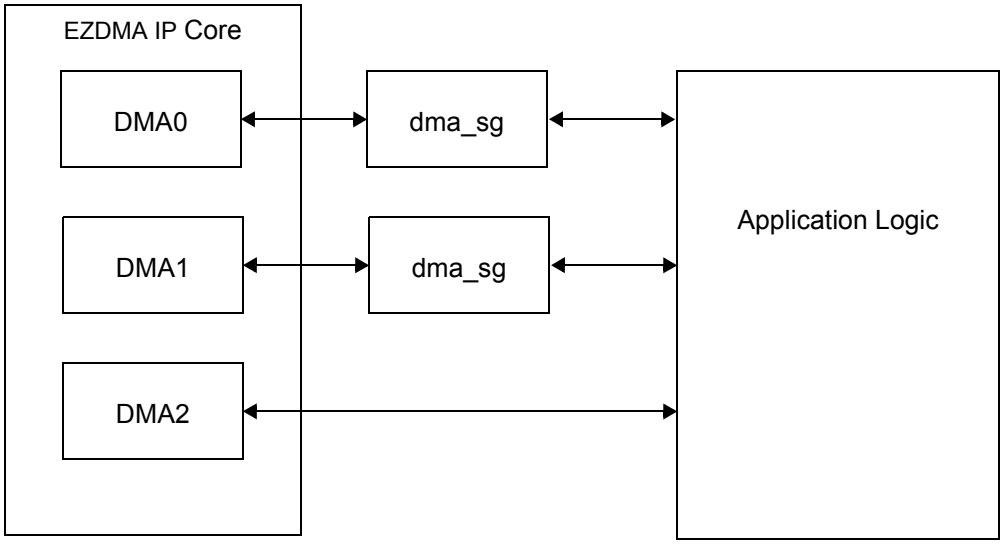

Figure 20: DMA Sample Implementation

The following figure shows scatter-gather controller signals:

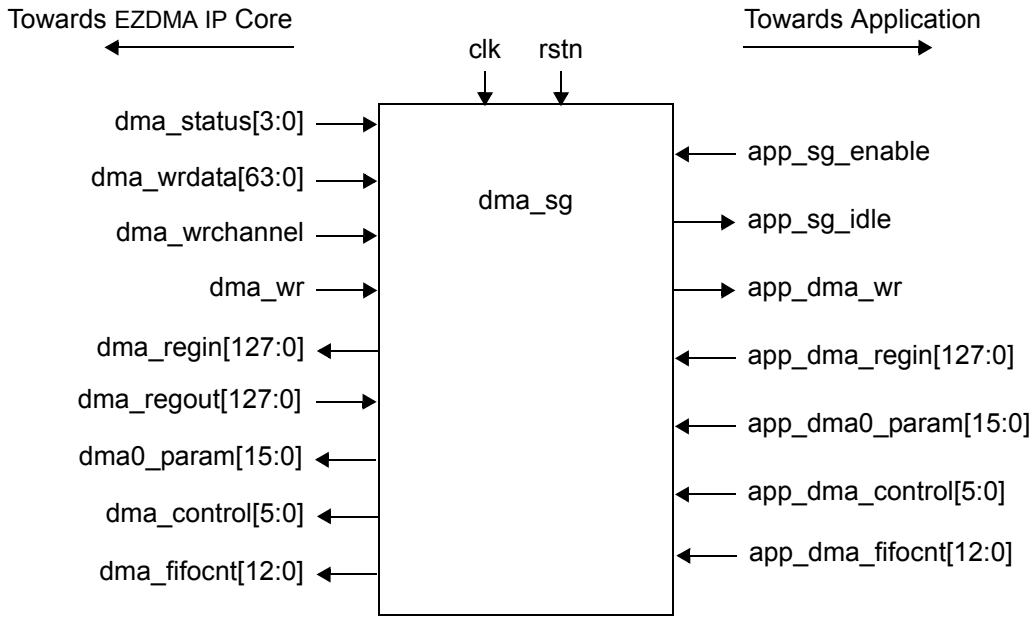

Figure 21: Controller signals

The table below describes scatter-gather controller signals:

Table 10: Controller Signals

Signal	Description
clk	125 or 250 MHz clock (same as PCIe Core).
rstn	Active-low reset (same as PCIe Core).
app_sg_enable	This signal must be asserted when a DMA transfer is programmed in order to use scatter-gather mode, otherwise direct mode is used.
app_dma_wr	**Write enable for application logic**: this signal indicates that data present on dma_wrdata must be stored.
app_dma_regin[127:0]	This signal indicates the address and size of transfer to be programmed (same as dma_regin input of PCIe Core).
app_dma0_param[23:0] (also for DMA 1 - 7)	This signal indicates parameters to be programmed (same as dma_param input of PCIe Core).
app_dma_control[5:0]	This signal is used to program and abort DMA transfer (same as dma_control input of PCIe Core).
dma_status [3:0]	DMA status from DMA channel.
dma_wrdata[63:0]/[127:0]	DMA write data from PCIe Core.
dma_wrchannel	DMA channel write enable for DMA channel.
dma_wr	DMA write enable from PCIe Core
dma_regin[127:0]	DMA channel registers input.
dma_regout[127:0]	DMA channel registers output.
dma0_param[23:0] (also for DMA 1 - 7)	DMA channel parameters.
dma_control[5:0]	DMA channel control.
app_dma_fifocnt[12:0]	This signal indicates the DMA FIFO count from application logic.
dma_fifocnt[12:0]	DMA FIFO count.
app_sg_idle	This signal indicates when the scatter-gather controller is idle.

3.7 Slave Module Interface

3.7.1 Slave Module Signals

The following table describes the signals used to communicate between the Application Layer and the Slave module of the EZDMA2 IP Core:

Table 11: Slave Module Signals

Signal	I/O	Description
slv_dataout[31:0]/ [63:0]/[127:0]	out	**Slave Data Out**: This signal is the data bus for output in slave mode. It reads data received from the PCI Express bus during Slave Write transactions.
slv_bytevalid[3:0]/ [7:0]/[15:0]	out	**Slave Byte Valid**: This signal indicates which bytes of `SLV_DATAOUT[]` are valid during each data phase of a write transaction, when `SLV_WRITE` is asserted. • slv_bytevalid[0]: enables data bits [3:0] or [7:0] • slv_bytevalid[7]: enables data bits [63:56] • slv_bytevalid[15]: enables data bits [127:120] During a read transaction, this signal indicates the value of the last/first DWORD byte enable field of the TLP header: • slv_bytevalid[3:0]: first DWORD byte enable • slv_bytevalid[7:4]: last DWORD byte enable
slv_bytecount[12:0]	out	**Slave Byte Count**: This signal indicates the size of a transaction from first enabled byte to last enabled byte. Possible values are from 1 to 4096 bytes.
slv_dwcount[10:0]	out	**Slave DW Count**: This signal indicates the transaction's DW count. Possible values are from 1 to 1024 DW.
slv_addr[63:0]	out	**Slave Address**: This signal acts as a memory address counter. Address bits can be directly connected to internal or external static memory devices. The Address is initialized with the PCI start address of a transaction and then automatically incremented whenever a DW is written.
slv_bar[6:0]	out	**Slave BAR**: This signal indicates which space is targeted during a slave access: • slv_bar[5:0]: set when BAR0... BAR5 are targeted • slv_bar[6]: set when expansion ROM BAR is targeted
slv_readreq	out	**Slave Read Request**: This signal indicates when a read request is received. Application logic must store the address, target BAR and size (as necessary), as well as `SLV_CPLADDR[31:0]`, and `SLV_CPLPARAM[4:0]` in order to complete the transaction.
slv_cpladdr[31:0]	out	**Slave Completion Address**: This signal indicates the DMA address required to perform a completion: • slv_cpladdr[6:0]: Low address • slv_cpladdr[7]: reserved • slv_cpladdr[15:8]: Requester tag • slv_cpladdr[31:16]: Requester ID
slv_cplparam[7:0]	out	**Slave Completion Parameter**: This signal contains the DMA parameters required to perform a completion: • slv_cplparam[7:6]: Request attributes • slv_cplparam[5:3]: Request traffic class • slv_cplparam[2:0]: reserved
slv_writereq	out	**Slave Write Request**: This signal indicates when a write request is received.

Table 11: Slave Module Signals

Signal	I/O	Description
slv_write	out	**Slave Write**: This signal indicates that data present on SLV_DATAOUT must be written at the address specified by SLV_ADDR.
slv_lastwrite	out	**Slave Last Write**: This signal is asserted for 1 clock cycle when the last DW of a Memory or I/O Write transfer is received.
slv_io	out	**Slave IO**: This signal indicates whether a request is IO. It can be used to distinguish IO and memory requests.
slv_accept	in	**Slave Accept**: The Application asserts this signal when SLV_READREQ or SLV_WRITEREQ are asserted in order to accept the corresponding transaction. If the transaction is an I/O Write, then a completion with Successful Completion (SC) status is issued. For Slave read requests, SLV_ACCEPT needs to be asserted before completion timeout is reached on the Host system. For Slave write requests, the application can assert SLV_ACCEPT at any time, however, all TLPs received will be blocked until SLV_ACCEPT is asserted.
slv_abort	in	**Slave Abort**: The Application asserts this signal when SLV_WRITEREQ or SLV_READREQ are asserted in order to abort the corresponding transaction. If the transaction is not posted, then a Completion with Completion Abort (CA) status is issued.
slv_ur	in	**Slave Unsupported**: The Application asserts this signal while SLV_WRITEREQ or SLV_READREQ are asserted in order to treat the corresponding transaction as an unsupported request. If the transaction is not posted, then a Completion with Unsupported Request (UR) status is issued.

3.7.2 Transaction Examples using Slave Signals

Example 7: Typical Transfer of a Write Request

In this example, the Core transmits a Write request. If the request is an I/O Write transaction, the Core automatically sends a Completion with Successful Completion (SC) status.

Note in particular:

- **clock cycle 5**: Data transmission begins 2 clock cycles after assertion of `SLV_ACCEPT`.

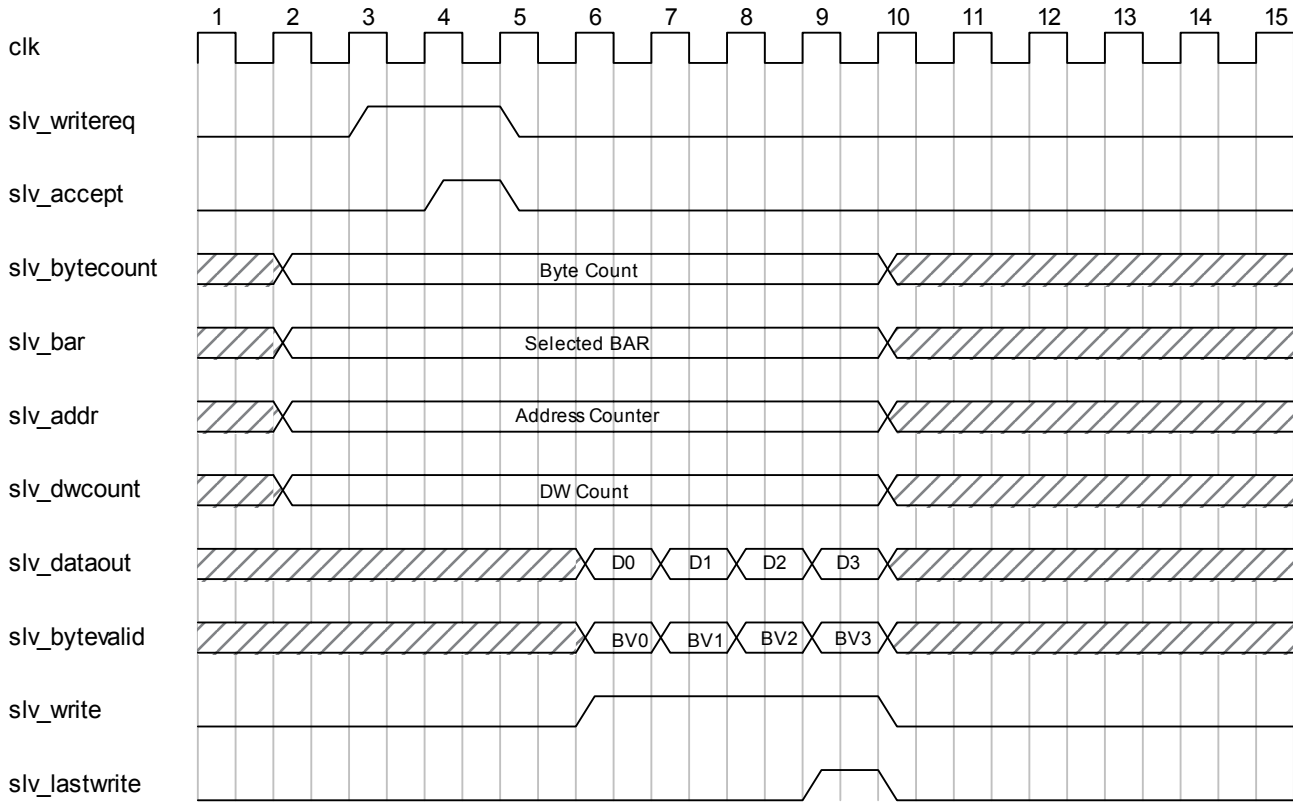

Figure 22: Waveform illustrating a typical Write request

Example 8: Typical Transfer of a Read Request

In this example, the Core transmits a Read request. The Application Layer is responsible for sending a Completion.

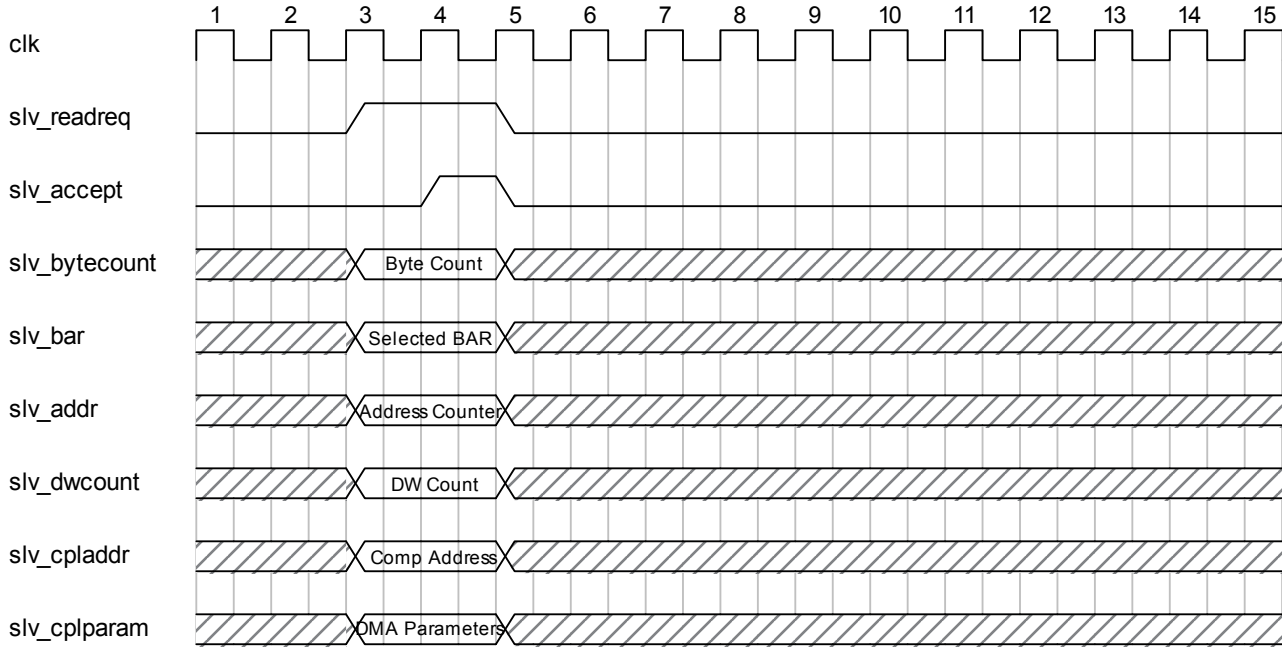

Figure 23: Waveform illustrating a typical Read request

Example 9: Aborting a Write request

Application logic might need to abort a transaction if a permanent error occurs.

In this example, the Core transmits a Write request, which is then aborted. The Core automatically sends a Completion with Completion Abort (CA) status.

Note in particular:

clock cycle 5: `SLV_WRITEREQ` is deasserted 1 clock cycle after the assertion of `SLV_ABORT`.

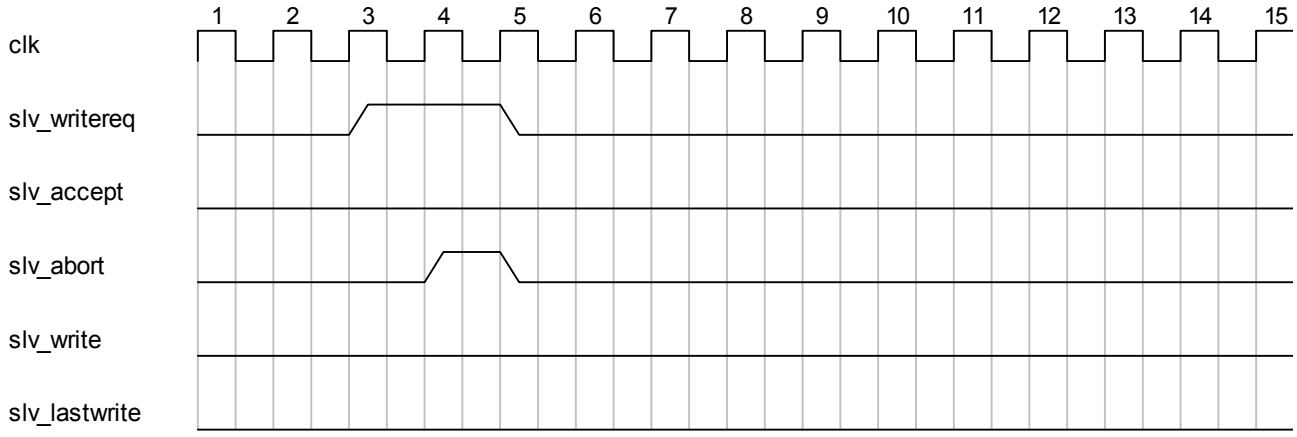

Figure 24: Waveform illustrating an aborted Write request

Example 10: Aborting a Read request

In this example, the Core transmits a Read request of 4 DW, which is then aborted. The Core automatically sends a Completion with Completion Abort (CA) status.

Note in particular:

- **clock cycle 5**: SLV_READREQ is deasserted 1 clock cycle after the assertion of SLV_ABORT.

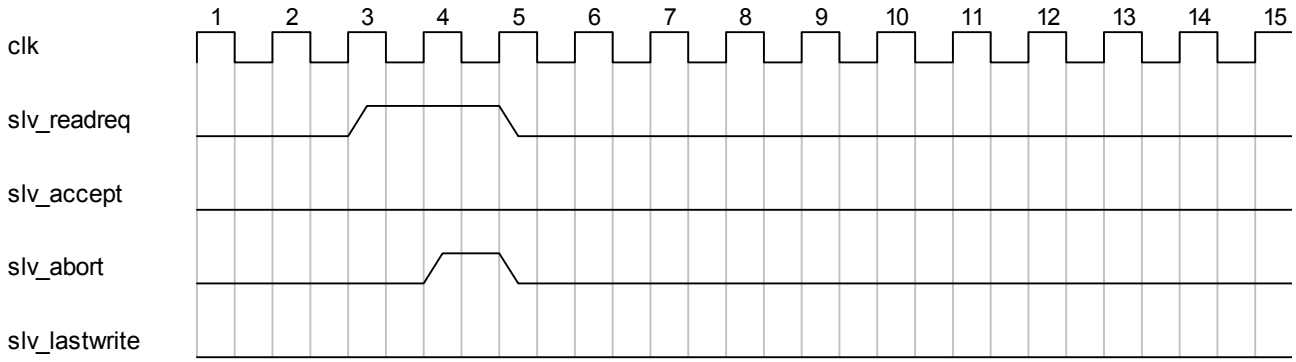

Figure 25: Waveform illustrating an aborted Read request

3.8 Handling Interrupts

The following table describes interrupt signals:

Table 12: Interrupt Signals

Signal	I/O	Description
int_request	in	**Interrupt Request**: This signal is asserted by the Application Layer to request an interrupt, and deasserted when the interrupt has been serviced by software. If MSI is enabled (bit 0 of CFG_MSICSR is set), MSI is used to signal interrupts, otherwise interrupt pin A signaling is used.
int_ack	out	**Interrupt Acknowledge**: This signals is asserted by the Core following a rising edge of INT_REQUEST to indicate that information has been sent to the PCIe link, and is deasserted when INT_REQUEST is deasserted. Note that the value of INT_REQUEST must not change before the previous change is acknowledged.
int_msgnum[4:0]	in	**Interrupt MSG Number**: This signal indicates the MSI message number and must be valid when INT_REQUEST is asserted. The CFG_MSICSR[6:4] signal (see Section 3.9) indicates how many messages are allocated. If multiple MSI messages are not used, this signal should be hardwired to 0.

Example 11: Sending Interrupts

The Application asserts INT_REQUEST to generate an interrupt and provides information about the interrupt source through a dedicated interrupt register. The application layer keeps INT_REQUEST asserted until the software has acknowledged the interrupt using one of the following methods:

- **ISR_RW**: Software reads and writes the value read to the interrupt register. INT_REQUEST is de-asserted when the driver writes to the interrupt register
- **ISR_READ**: Software reads the interrupt register. INT_REQUEST is de-asserted when the interrupt register is read.
- **ISR_WRITE**: Software reads and writes a specific value (set by the Software) to the interrupt register. INT_REQUEST is de-asserted when the interrupt register is read.

MSI is used to signal an interrupt if this mechanism has been enabled by the host system:

- An MSI message with message number specified by INT_MSGNUM is sent when INT_REQUEST is asserted and INT_ACK confirms that it has been sent to the PCIe Link. Note that host system might allocate less message numbers than requested: application must check CFG_MSICSR to know which messages are available.
- No information is sent to the PCIe Link when interrupt request is deasserted.

If MSI is not enabled, then interrupt A pin is used instead:

- Int A assert message is sent to PCIe Link when Int request is asserted and INT_ACK confirms that it has been sent.
- Int A deassert message is sent when Int request is deasserted and INT_ACK confirms that it has been sent.

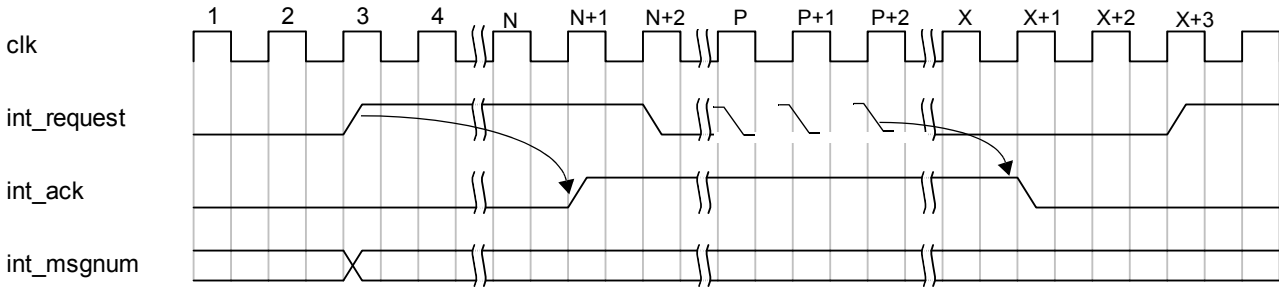

Figure 26: Waveform illustrating an interrupt sent by the application

3.9 Configuration Interface

The following table describes configuration signals:

Table 13: Configuration Signals

Signal	I/O	Description

cfg_msicsr[15:0]	out	**Configuration MSI Control Status Register**: This output provides essential fields of this register; it is not an accurate image of RSI N(A) counts, however:
		• **cfg_msicsr[15:7]**: reserved
		• **cfg_msicsr[6:4]**: Multiple Message Enable: This field indicates permitted values for MSI signals. For example, if "100" is written to this field 16, 8, 4, 2, or 1 MSI signals are allocated.
		  • 000: 1 MSI allocated
		  • 001: 2 MSI allocated
		  • 010: 4 MSI allocated
		  • 011: 8MSI allocated
		  • 100: 16 MSI allocated
		  • 101: 32MSI allocated
		  • 110: reserved
		  • 111: reserved
		• **cfg_msicsr[3:1]**: Multiple Message Capable: This is the value set using the Wizard.
		  • 000: 1 MSI requested
		  • 001: 2 MSI requested
		  • 010: 4 MSI requested
		  • 011: 8 MSI requested
		  • 100: 16 MSI requested
		  • 101: 32 MSI requested
		  • 110: reserved
		  • 111: reserved
		cfg_msicsr[0]: MSI enable: 0 = this component cannot use MSI.

cfg_prmcsr[31:0]	out	**Configuration Primary Control Status Register**: See *PCI Express Specifications* for details
cfg_devcsr[31:0]	out	**Configuration Dev Control Status Register**: See *PCI Express Specifications* for details.
cfg_linkcsr[31:0]	out	**Configuration Link Control Status Register**: See *PCI Express Specifications* for details.
cfg_ltssm[4:0]	out	**Configuration Link Training Module**: Reports Link Training Module state: • 00000: Initialization • 01111: L0 • 10101: L0s • 10111: L1 • 11111: In transition

3.10 Test Mode

The following table describes the test mode signal used for simulation and debugging:

Table 14: Test Mode Signals

Signal	I/O	Description
test_mode[15:0]	in	**Test Mode**: Default values are all 0s. • test_mode[0]: reserved • test_mode[1]: Set this bit to 1 to force the Receive Module to treat all requests as Unsupported Requests. • test_mode[2]: Set this bit to 1 to force the Master module to consider timeout expired for all pending Completions. • test_mode[3]: Set this bit to 1 to disable warning assertions. • test_mode[15:4]: reserved

3.11 Connecting the EZDMA2 IP to the Xilinx Hard IP

The following signals are used to connect the EZDMA2 IP Core to the XIlinx Hard IP. Refer to the *FPGA User Guides (see* Other Publications) for a complete description of these signals.

Table 15: Xilinx Hard IP Interface Signals

Virtex/Spartan Signals	7-Series Signals	I/O
Transmit Interface: These signals must be directly connected to the Xilinx Hard IP Core.		
trn_td[31:0]/[63:0]/[127:0] **trn_trem_n[7:0]** (Virtex-5) **trn_trem_n[1:0]** (Virtex-6) **trn_tsof_n** **trn_teof_n** **trn_tsrc_dsc_n** **trn_tsrc_rdy_n** **trn_terrfwd_n** **trn_terr_drop_n** (Virtex-6 & Spartan-6) **trn_tstr_n**(Virtex-6 & Spartan-6)	**m_axis_rx_tdata[31:0]/[63:0]/[127:0]** **m_axis_rx_tkeep** **m_axis_rx_tlast** **m_axis_rx_tvalid** **m_axis_rx_tuser**	out
trn_tdst_dsc_n **trn_tdst_rdy_n** **trn_tbuf_av[3:0]** (Virtex-5) **trn_tbuf_av[5:0]** (Virtex-6 & Spartan-6)	**m_axis_rx_tready** (Kintex 7) **rx_rnp_ok** (Kintex 7)	in
Receive Interface: These signals must be directly connected to the Xilinx Hard IP Core.		
trn_rdst_rdy_n **trn_rnp_ok_n** **trn_rcpl_streaming_n** (Virtex-5)	**s_axis_tx_tready** **tx_err_drop** **tbuf_av**	out

Table 15: Xilinx Hard IP Interface Signals

Virtex/Spartan Signals	7-Series Signals	I/O
trn_rd[31:0]/[63:0]/[127:0] **trn_rrem_n[7:0]** (Virtex-5) **trn_rrem_n[1:0]** (Virtex-6) **trn_rsof_n** **trn_reof_n** **trn_rsrc_dsc_n** **trn_rsrc_rdy_n** **trn_rbar_hit_n [6:0]** **trn_rerrfwd_n**	**s_axis_tx_tdata[31:0]/[63:0]/[127:0]** **s_axis_tx_tkeep** **s_axis_tx_tlast** **s_axis_tx_tvalid** **s_axis_tx_tuser**	in
Error Reporting: These signals must be directly connected to the Xilinx Hard IP Core.		
cfg_err_cor_n **cfg_err_cpl_abort_n** **cfg_err_cpl_timeout_n** **cfg_err_cpl_unexpect_n** **cfg_err_ecrc_n** **cfg_err_posted_n** **cfg_err_tlp_cpl_header[47:0]** **cfg_err_ur_n** **cfg_err_locked_n** **cfg_trn_pending_n**	**cfg_err_cor** **cfg_err_cpl_abort** **cfg_err_cpl_timeout** **cfg_err_cpl_unexpect** **cfg_err_ecrc** **cfg_err_posted** **cfg_err_tlp_cpl_header[47:0]** **cfg_err_ur** **cfg_err_locked** **cfg_trn_pending**	out
cfg_err_cpl_rdy_n	**cfg_err_cpl_rdy**	in
Interrupt Control: These signals must be directly connected to the Xilinx Hard IP Core.		
cfg_interrupt_n **cfg_interrupt_assert_n** **cfg_interrupt_di**	**cfg_interrupt** **cfg_interrupt_assert** **cfg_interrupt_di**	out
cfg_interrupt_rdy_n **cfg_interrupt_do** **cfg_interrupt_mmenable** **cfg_interrupt_msienable**	**cfg_interrupt_rdy** **cfg_interrupt_do** **cfg_interrupt_mmenable** **cfg_interrupt_msienable**	in
Configuration Management: These signals must be directly connected to the Xilinx Hard IP Core.		
cfg_pcie_link_state _n[2:0] **cfg_bus_number [7:0]** **cfg_device_number[4:0]** **cfg_function_number[2:0]** **cfg_status[15:0]** **cfg_command[15:0]** **cfg_dstatus[15:0]** **cfg_dcommand[15:0]** **cfg_lstatus [15:0]** **cfg_lcommand [15:0]**	**cfg_pcie_link_state [2:0]** **cfg_bus_number [7:0]** **cfg_device_number[4:0]** **cfg_function_number[2:0]** **cfg_status[15:0]** **cfg_command[15:0]** **cfg_dstatus[15:0]** **cfg_dcommand[15:0]** **cfg_lstatus [15:0]** **cfg_lcommand [15:0]**	in
Miscellaneous: These signals must be directly connected to the Xilinx Hard IP Core.		
trn_lnk_up_n	**trn_lnk_up**	in

Appendix A: PCI Express System Performance

The EZDMA2 IP Core is highly customizable, so choices you make during configuration impact on how the Core functions. Factors affecting system performance include:

- Latency
- Maximum effective bandwidth
- Actual link usage

A.1 Latency

Latency is the delay in transferring a packet between two points. Global latency, that is, the total latency within a system, is the sum of several factors:

- Data payload size of a TLP
- Core latency between the Application Layer and the Link (varies between Receive and Transmit transactions)
- Switch latency
- Completer latency, such as SDRAM read latency and SRAM mailbox read latency
- PCI Express to PCI/PCI-X bridge latency and latency inherent to a PCI component

You can avoid high global latency by carefully designing your system's architecture (limiting the number of Switches, for example) and by modifying the width of critical Links.

The following table shows typical latency values using a x4 lane for various types of Read Request transactions and for different types of memory (mailbox or SDRAM):

Table 16: Typical latency values of Read Request transactions

Fabric Environment	Data Payload Size and Latency		
	8B	32B	256B
Link bandwidth usage	64 ns	88 ns	312 ns
Point-to-point, mailbox	240 ns	264 ns	464 ns
Point-to-point, SDRAM	272 ns	296 ns	496 ns
Point-to-point, PCI peripherals	448 ns	472 ns	904 ns
Through switch, mailbox	528 ns	552 ns	776 ns
Through switch, SDRAM	560 ns	584 ns	812 ns
Through switch, PCI peripherals	672 ns	720 ns	1144 ns

The type of transaction also affects latency:

- **Read transactions (high latency potential)**: Most systems generate more than one completion transaction per Read Request. More transactions means greater potential latency.
- **Write transactions (low latency potential)**: Write operations are posted and do not require completions. Fewer transactions means lower potential latency.
- **Small transactions (high latency potential)**: Many small packets will result in greater latency than one large packet, because data payload transfer time contributes to overall latency.

A.2 Maximum Effective Bandwidth

Bandwidth is a measure of the rate at which data is transferred at a specific point of a Link. Maximum Effective Bandwidth is the rate at which “valuable” data is transferred at a particular point. It doesn’t include transaction «overhead», such as headers, sequence numbers, CRCs, ECRCs, and other packets like DLLPs and SKIP advanced sets.

Maximum Effective Bandwidth = data / (data + overhead)

The table below shows Maximum Effective Bandwidth values for Completion transactions sent in response to consecutive Read or Write requests on a x4 Link. For Read Requests, the Completion transaction size can be split into packets of 64 DW, 32 DW, or 16 DW. An overhead of 5 DW for each packet is presumed, with an additional 2 DW of DLLP for Read Request transactions:

Table 17: Sample Maximum Effective Bandwidth Values

Completion transaction size (bytes / DW)	Completion packet(s) in response to a...			
	Write Request	Read Request 64 DW Packet	Read Request 32 DW Packet	Read Request 16 DW Packet
4 / 1	16%	12%	12%	12%
8 / 2	28%	22%	22%	22%
32 / 8	61%	53%	53%	53%
64 / 16	76%	69%	69%	69%
128 / 32	86%	82%	82%	72%
256 / 64	92%	90%	84%	74%
512 / 128	96%	91%	85%	75%
1024 / 256	98%	92%	86%	75%
2048 / 512	99%	92%	86%	75%
4096 / 1024	99%	93%	86%	76%

For example, to calculate the Maximum Effective Bandwidth for a Completion packet of 128 DW, divided into four packets of 32 DW, generated in response to a Read Request:

- 4 = number of Completion Packets
- 5 = number of DW of overhead per packet
- 2 = number of DW per DLLP for the complete Read Request transaction
- **Overhead** = (4 X 5) + 2 = 22
- **Maximum Effective Bandwidth** = 128 / (128 + 22) = 85%

Note: A Maximum Payload Size of 64 DW or 128 DW provides between 92% and 96% of effective bandwidth for Write transactions and practically the saturation limit for Read transactions. Maximum payloads in this range can be a good trade-off between latency, congestion, and Core size. Systems that require higher bandwidth might need higher maximum data payloads for effective bandwidth values of up to 99%.

A.3 Actual Link Usage

Actual Link Usage is a measure of how much a Link is actually used in a given period of time, and is defined by the following equation:

Actual Link Usage = active time / (active time + idle time)

A.4 System Performance Illustrated

You can optimize system performance by defining your objectives (is maintaining a small Core size more important than maximum throughput?) and balancing the following factors:

- Maximum packet size
- Global system latency
- Number of outstanding requests per component

In general, you should consider increasing the number of outstanding requests with small data payloads and in high-latency systems.

By default, the maximum value of outstanding requests possible per component is 32, but by setting the Extended tag and the Phantom Function Number Enable bit, you can set outstanding requests up to 2,048. However, once Link Usage has reached 100%, increasing the number of outstanding requests will not improve system performance.

The following examples use a Maximum Effective Bandwidth of 82%.

A.4.1 One Outstanding Request, Small Packet

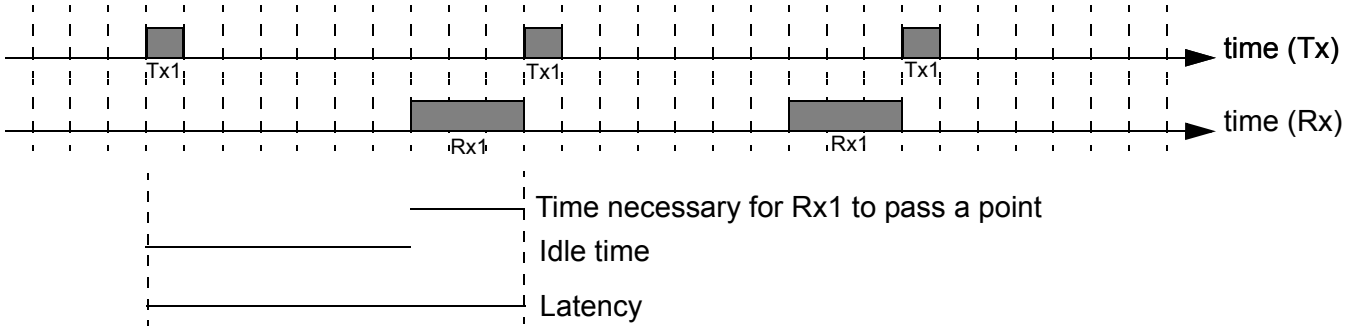

Figure 27: One outstanding request, small packet

In this example, Tx1 represents a Read Request of 32 DW. Rx1 represents the corresponding Completion packet. One period (the latency of the transaction) is the time it takes for a Request transaction to be sent to a Completer and have a Completion packet returned. The Actual Link Usage is:

Actual Link Usage = Rx1 / (Rx1 + idle time)

Actual Link Usage = 3 / (3 + 7) = 30%

For a 32 DW Read Request, the Maximum Effective Bandwidth is 82% (see Table 17). If the actual link usage is 30%, calculate 30% of the Maximum Effective Bandwidth to obtain effective (actual) bandwidth:

Effective Bandwidth = .3 X .82 = 25%

The effective bandwidth of this Link is only 25%.

A.4.2 Two Outstanding Requests, Small Packet

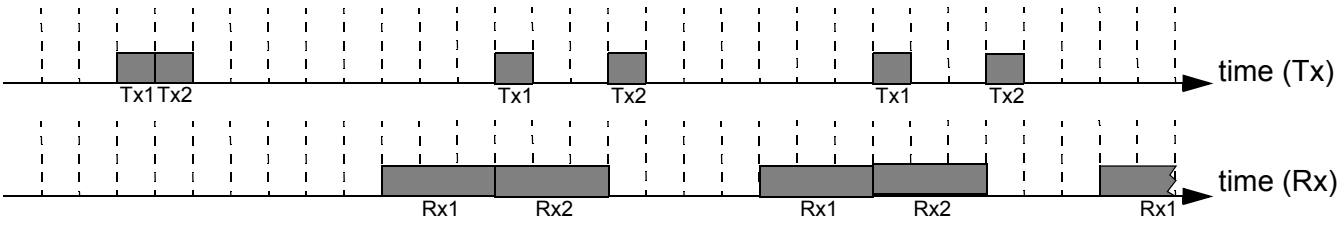

Figure 28: Two outstanding requests, small packet

Tx1 and Tx2 represent Read Requests of 32 DW. By increasing the number of outstanding requests to two and leaving all other variables the same, effective bandwidth is:

- Actual Link Usage = 6 / (6 + 4) = 60%
- Effective Bandwidth = .6 X .82 = 49%

A.4.3 Four Outstanding Requests, Small Packet

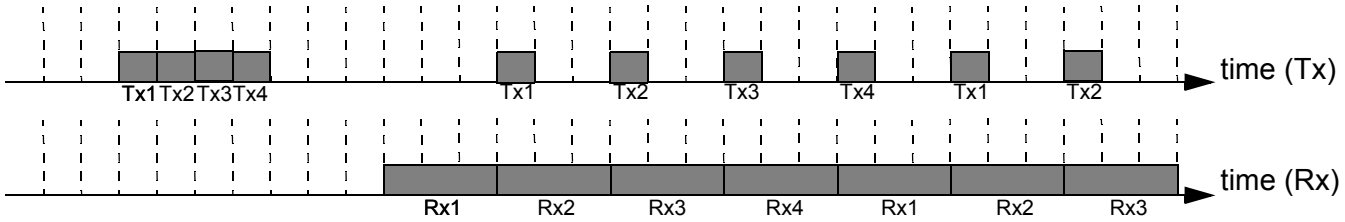

Figure 29: Four outstanding requests, small packet

Tx1, Tx2, Tx3, and Tx4 represent Read Requests of 32 DW. Outstanding requests are set to four and the Actual Link Usage = 100%.

Actual Link Usage = 10 / (10 + 0) = 100%

Effective Bandwidth = 1 X .82 = 82%

This Link is performing at Maximum Effective Bandwidth. Increasing the number of outstanding requests will not improve system performance and will needlessly increase the size of the Core.

A.4.4 Two Outstanding Requests, Large Packet

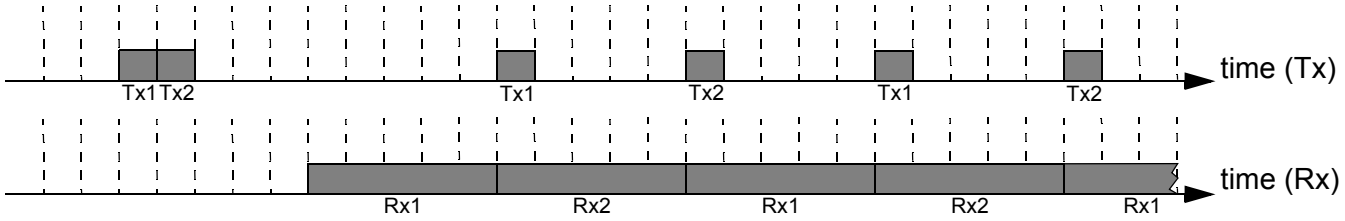

Figure 30: Two outstanding requests, large packet

Tx1 and Tx2 represent Read Requests of 32 DW. There are only 2 outstanding requests, but increasing the Maximum Packet Size (Max_Payload_Size) enables Maximum Effective Bandwidth.

Actual Link Usage = 10 / (10 + 0) = 100%

Effective Bandwidth = 1 X .82 = 82%

A.4.5 High Latency System

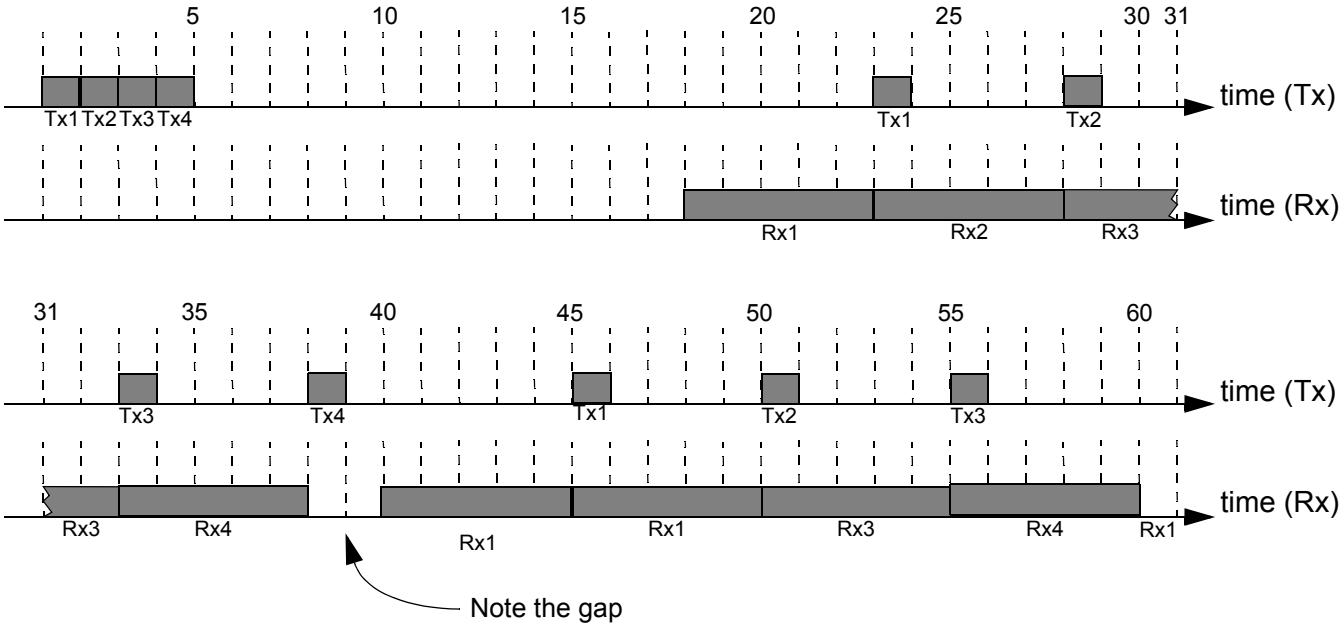

Figure 31: High latency system

Tx1, Tx2, Tx3, and Tx4 represent Read Requests of 32 DW. The time it takes for a Request to be sent and for the component to receive the corresponding Completion TLP is relatively large, 22 in this case.

Actual Link Usage = 22 / (22 + 2) = 92%

Effective Bandwidth = .92 X .82 = 75%

Even with a high number of outstanding requests and large packets (Max_Payload_Size), this Link will never reach its Maximum Effective Bandwidth because of high global latency.

Appendix B: Register content of TLPs

The following tables describe register content for all types of descriptors (TLPs):

- Register content for TLPs without a data payload
- Register content for TLPs with a data payload

This Appendix is provided in order to help you debug potential problems.

For more information about the register content of TLPs, see the PCI Express™ Base Specification Revision 2.0.

B.1 Register Content for a TLP without a Data Payload

Table 18: Memory Read Request 32-bit addressing descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr			0	0	Length								
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Table 19: Memory Read Request-Locked 32-bit addressing descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	0	1	0	TC			0	0	0	0	TD	EP	Attr			0	0	Length								
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	Reserved																															

Table 20: Memory Read Request 64-bit addressing descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	1	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr			0	0	Length								
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[63:32]																															
Byte 12	Address[31:2]																														0	0

Table 21: Memory Read Request-Locked 64-bit addressing descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	1	0	0	0	0	1	0	TC			0	0	0	0	T	EP	Attr		0	0	Length									
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[63:32]																															
Byte 12	Address[31:2]																														0	0

Table 22: I/O Read Request descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	R																															

Table 23: Type 0 Configuration Read Request descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Bus Number								Device Nb.								Func				0	0	0	0	Ext. Reg.				Register Nb.			
Byte 12	R																														0	0

Table 24: Type 1 Configuration Read Request descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Bus Number								Device Nb.								Func				0	0	0	0	Ext. Reg.				Register Nb.			
Byte 12	R																															

Table 25: Message (without data) descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	1	1	0	r₂	r₁	r₀	0	TC			0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Byte 4	Requester ID																Tag								Message Code							
Byte 8	Vendor defined or all zeros																															
Byte 12	Vendor defined or all zeros																															

Table 26: Completion (without data) descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	1	0	1	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Completer ID																Status			B	Byte Count											
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	R																															

Table 27: Completion Locked (without data) descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	0	0	0	1	0	1	1	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Completer ID																Status			B	Byte Count											
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	R																															

B.2 Register Content for a TLP with a Data Payload

Table 28: Memory Write Request 32-bit addressing descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	R																															

Table 29: Memory Write Request 64-bit addressing descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	1	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr			0	0	Length								
Byte 4	Requester ID																Tag								Last BE				First BE			
Byte 8	Address[63:32]																															
Byte 12	Address[31:2]																														0	0

Table 30: I/O Write Request descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Address[31:2]																														0	0
Byte 12	R																															

Table 31: Type 0 Configuration Write Request descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Bus Number								Device Nb.					Func			0	0	0	0	Ext. Reg.				Register Nb.						0	0
Byte 12	R																															

Table 32: Type 1 Configuration Write Request descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	TD	EP	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Byte 4	Requester ID																Tag								0	0	0	0	First BE			
Byte 8	Bus Number								Device Nb.					Func			0	0	0	0	Ext. Reg.				Register Nb.						0	0
Byte 12	R																															

Table 33: Completion (with data) descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	1	0	1	0	0	TC			0	0	0	0	TD	EP	Attr			0	0	Length								
Byte 4	Completer ID																Status				B	Byte Count										
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12	R																															

Table 34: Completion Locked (with data) descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	0	0	1	0	1	1	0	TC			0	0	0	0	TD	EP	Attr			0	0	Length								
Byte 4	Completer ID																Status				B	Byte Count										
Byte 8	Requester ID																Tag								0	Lower Address						
Byte 12																																

Table 35: Message (with data) descriptor format

	+0								+1								+2								+3							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Byte 0	0	1	1	1	0	r₂	r₁	r₀	0	TC			0	0	0	0	TD	EP	0	0	0	0			Length							
Byte 4	Requester ID																Tag								Message Code							
Byte 8	Vendor defined or all zeros for Slot Power Limit																															
Byte 12	Vendor defined or all zeros for Slots Power Limit																															

Appendix C: PCI Express Fundamentals

This Appendix describes some of the basic features of the PCI Express protocol for users that are not familiar with the protocol.

For a full description, see the *PCI Express Specifications.*

C.1 About PCI Express

PCI Express is a third generation, high-bandwidth, low voltage, differential serial interconnect technology that maintains compatibility with existing PCI infrastructures. It is derived from the PCI and PCI-X protocols, however PCI Express represents an architectural leap beyond its sister technologies rather than simply an extension of the existing norms.

PCI Express includes the following features:

- Increased bandwidth: high-speed links can currently transfer up to 2.5 gigabits per second, and future implementations of PCI Express will support as much as 10 gigabits per second
- Isochronous transactions: With the introduction of Virtual Channels (VCs) and Traffic Classes (TCs), the designer can prioritize traffic flow
- Serial point-to-point interface
- High bandwidth per pin
- Scalability
- Support for differentiated services, that is, different Qualities of Service (QoS)
- Power Management and budgeting
- Hot-Plug and Hot-Swap support
- Ability to maintain Link-level and end-to-end data integrity
- Advanced Error handling

C.2 PCI Express Lanes and Links

A PCI Express lane is the fundamental connection between two devices. Each lane is defined by a pair of differential transmit signals and a pair of differential receive signals, as illustrated below.

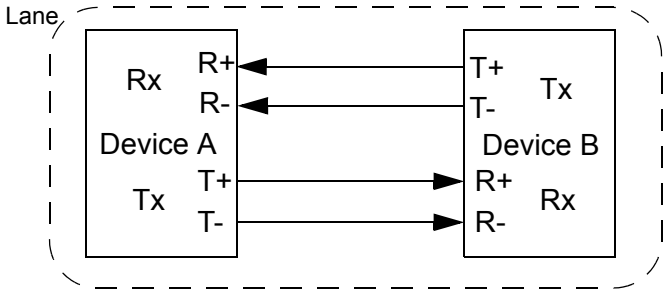

Figure 32: A PCI Express Lane

With a PCI Express lane, there is no arbitration for a shared Bus, as the Transmitting device is always directly connected with a Receiving device and each device is always both a transmitting device and a receiving device.

A PCI Express Link is defined as the connection between two devices and includes one or more lanes. The *PCI Express Specifications* define x1, x2, x4, x8, x12, x16, and x32 lanes for serial Links. The following figure illustrates a x4 (four-lane) Link.

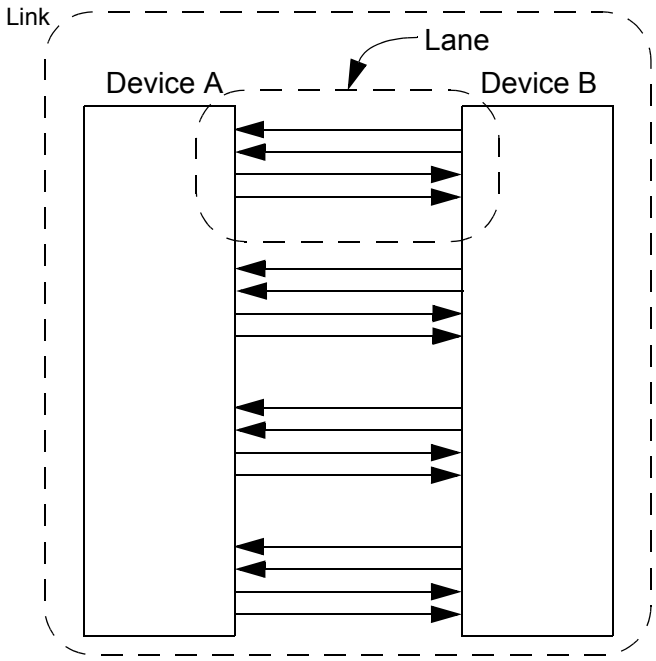

Figure 33: PCI Express x4 Link

PCI Express maintains compatibility with PCI software by simulating a PCI bus across each Link.

C.3 The PCI Express Fabric

The following figure illustrates a typical PCI Express fabric (or topology).

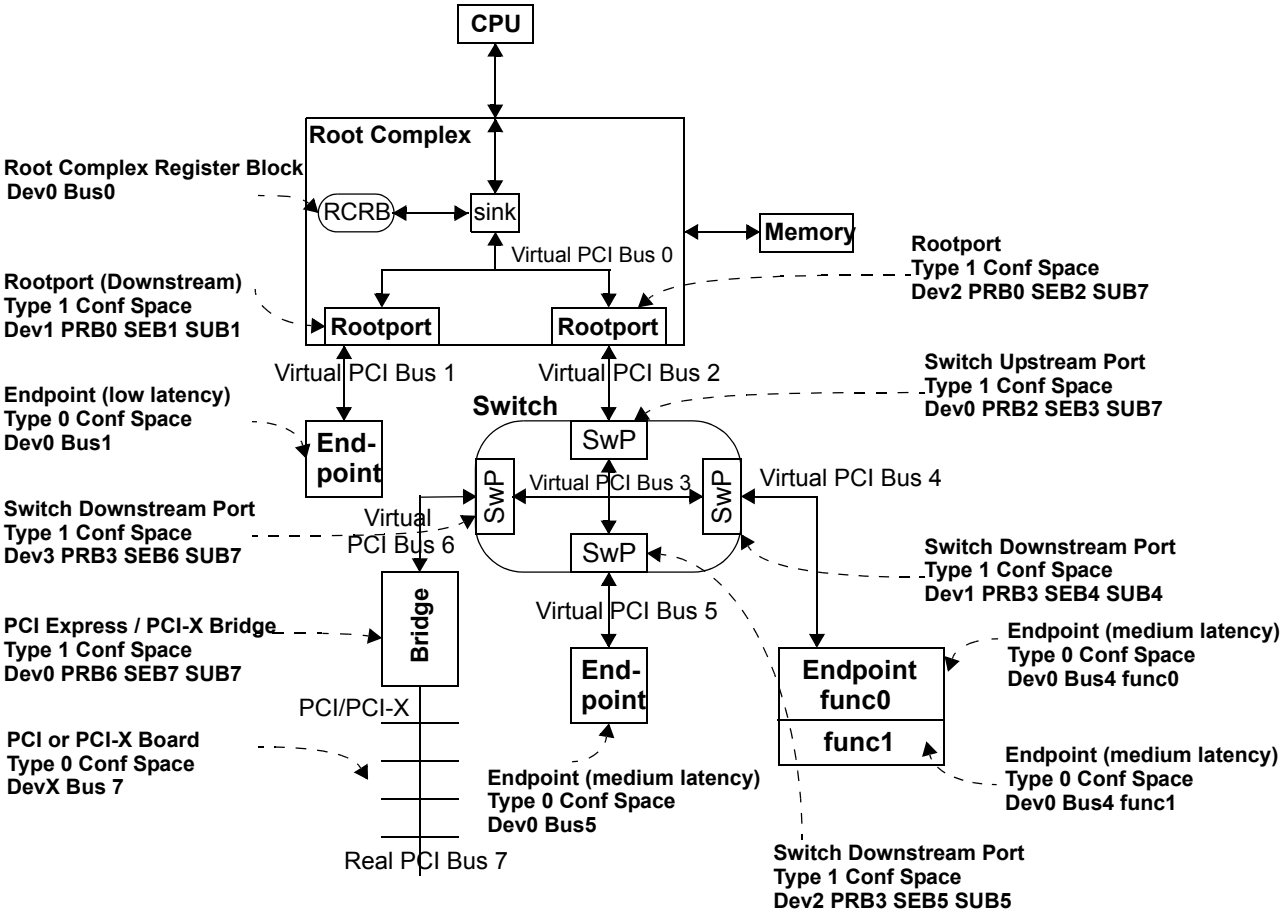

Figure 34: A Typical PCI Express Fabric

C.3.1 Root Complex

In the example above, the Root Complex is composed of one Root Complex Register Block (RCRB) associated with the sink unit (entry to the PCI Express fabric hierarchy) and two Rootports. The sink unit and the two Rootports constitute a virtual PCI bus segment that has Bus Number 0. This means that the Root Complex implements a virtual embedded switch between the Rootports and transfers are allowed directly between them (this is an optional feature of the Root Complex).

The PCI configuration software programs each Rootport as a Type 1 Configuration space. In the example, the first Rootport is Bus 0, Device 1 and the second Bus 0, Device 2.

The sink unit represents the start of the PCI Express fabric and is equivalent to a virtual PCI host port that can generate all transaction types (Memory, I/O, Completion, and Message) and also relay Request and Completion transactions used to access the central memory.

Rootport Device 1 is connected to an Endpoint through virtual PCI Bus Segment 1 and has the following Type 1 Configuration Space parameters:

- Primary Bus Number 0
- Secondary Bus Number 1
- Subordinate Bus Number 1 (only one bus number is located downstream of the Rootport)

Rootport device 2 is connected to a PCI Express Switch and has the following configuration parameters:

- Primary Bus Number 0
- Secondary Bus Number 2
- Subordinate Bus Number 7 (bus segments 2 through 7 are located behind the Rootport)

The Endpoint component located behind Rootport Device 1 is a low latency Endpoint, for example, a graphic

board where latency is critical for optimal performance. An Endpoint device number is always 0 and its Bus number is programmed by the PCI configuration software (Bus 1, in this example).

C.3.2 Endpoint

All PCI Express Endpoints implement a Type 0 Configuration Space and have Device number 0. In the example (Figure 34), the two Endpoints located behind the Switch have Bus Numbers 4 and 5. The Endpoint located behind Bus 4 implements two different functions (func0 and func1). The Endpoints located behind Bus Numbers 4 and 5 are considered medium latency components due to the fact that they are behind a Switch component and not directly connected to the Root Complex. In general, a PCI Express Endpoint component is considered to have high latency if two or more Switch components lie between it and the Root Complex.

The PCI Endpoint components located behind the Bridge are also implemented as Type 0 Configuration space components and respond only to Type 0 Configuration access requests (if their PCI_IDSEL pin is asserted).

C.3.3 Switch

In the example (Figure 34), the PCI Express Switch component implements four PCI Express ports, one upstream and three downstream. Each port is equivalent to a PCI Bridge and implements a Type 1 Configuration space. Internal routing between ports is accomplished by a virtual PCI Bus segment.

The following table describes the configuration of each Switch port in Figure 34.

Table 36: Configuration of Switch ports

Device Number	0 (upstream)	1 (downstream)	2 (downstream)	3 (downstream)
Primary Bus Number	2	3	3	3
Secondary Bus Number	3	4	5	6
Subordinate Bus Number	7	4	5	7

C.3.4 Bridge

The PCI Express / PCI Bridge component implements a common Type 1 Configuration space for both sides of the bridge: the PCI Express Core and the PCI component Core. The mechanism to handle transactions between the two is described in the *PCI Express Base Specification Revision 2.0* (as well as *Revisions 1.1 and 1.0a*). In Figure 34, the Bridge has the following parameters:

- Device number 0 (mandatory)
- Primary Bus Number 6 (PCI Express Core Link)
- Secondary Bus Number 7 (PCI bus)
- Subordinate Bus Number 7 (no other PCI / PCIe Bridges are located on the PCI bus)

C.4 Types of Transactions

The following table summarizes the different types of transactions possible in the PCI Express fabric:

Table 37: PCI Express transaction types and characteristics

Transaction Type	How request is handled	Routing method	Notes
Memory Write	Posted (no Completion is required)	Address	Memory Write Requests use Posted TLPs and can have data payloads of up to 4 Kbytes (depending on the Max_Payload_Size parameter of the Configuration Space). Memory Write Requests can use 32-bit address formatting or 64-bit address formatting.
Memory Read	Non-Posted (Completion is required)	Address	Memory Read Requests use Non-Posted TLPs and have no payload. A dedicated register of the configuration space defines the maximum Read Request size. Please see the following section for details concerning Completion transactions. Memory Read Requests can use 32-bit address formatting or 64-bit address formatting.
I/O Write	Non-Posted	Address	I/O Request transactions use Non-Posted TLPs. I/O Write Request transactions have a data payload of 1DW. I/O Requests always require a Completion packet, whether the request is returned successful or aborted. Please see the following section for details concerning Completion transactions.
I/O Read	Non-Posted	Address	
Configuration Write	Non-Posted	ID	Configuration Requests use Non-Posted TLPs. Configuration Write requests have data payloads of 1 DW. Each Configuration request requires a Completion regardless of whether the request was accepted. Please see the following section for details concerning Completion transactions.
Configuration Read	Non-Posted	ID	
Message	Posted	Address	In addition to the three PCI transactions (Memory, I/O, and Configuration), PCI Express introduces a fourth type of transaction for Messages. Message transactions support power management requests and emulate PCI legacy virtual pins (such as INT, PME, PERR, and SERR).

A Completion Transaction is a distinct package generated in the PCI Express fabric in response to a Read or Write Request. Like Configuration Requests, Completion transactions are routed by Transaction ID.

Completion transactions vary in nature depending on the reason for their generation. The following table describes differences in Completion transactions.

Table 38: Completion transaction characteristics

Completion transactions generated in response to a...	have these characteristics
Memory Request	Completion transactions are generated for Memory Read Requests but not Memory Write requests, since no completion is required for the latter. Completions can have data payloads of up to 4 Kbytes, depending on the Maximum Payload Size. Note that large Read Request transactions (transactions of 4 Kbytes, for example) might be divided into several smaller Completion Packets (64, 128, or 256 bytes on a 64 or 128 address boundary) in order to reduce overall latency and to optimize data flow within each Switch. Completion transactions may only be divided if they have an end address corresponding to the Read Completion Boundary (RCB) parameter, which is 64 or 128 bytes for Endpoints, Root Complexes, and Bridges, and 64 bytes for Switches. Note that the Core does not check for violations of the Read Completion Boundary (RCB).
I/O Request	Completion transactions are generated for both Read and Write I/O requests. Completion packets generated in response to I/O Write requests have no data payload, whereas a completion packet in response to an I/O Read request has a data payload of 1 DW. I/O Requests always require a Completion transaction regardless of return status (accepted, successful, unsupported, aborted, or retry).
Configuration Request	Completion transactions are generated for both Read and Write Configuration requests. Completion packets have data payloads of 1 DW if the request is accepted. Configuration Requests always require a Completion transaction regardless of return status (accepted, successful, unsupported, aborted, or retry).

C.5 Routing Rules

The PCI Express fabric uses the following routing rules:

- All Memory and I/O transactions are routed by address (BAR decoding for Endpoint Type 0 configuration spaces, and Memory Mapped I/O for Rootport, Switch, and Bridge Type 1 Configuration spaces).
- Configuration, Message, and Completion transactions are routed by the Transaction ID of the destination component (Bus, Device, and Function number) and comply with Type 1 Configuration Space routing specifications.

Message transactions are independent of the PCI configuration software and have different types of routing. See Section C.5.3 for details.

The TLP header includes information about the type of the transaction and whether the transaction is routed by address or ID.

C.5.1 Routing by Address

Transactions that are routed by address compare the destination address (included in the TLP header) with the registers of the component.

Table 39: Routing by Address Rules

Component Type	Routing
Type 0 Components (Endpoint)	If the address of the transmitted TLP lies within one of the component's implemented Base Address Registers (BARs), the component will accept the TLP. If the address of the TLP does not lie within the component's BARs, the TLP is rejected and an error is generated.
Type 1 Components (Rootport, Switch, or Bridge)	If a device receives a request on its Primary Bus and the destination address is located within one of the defined address windows (as defined by its registers), the request is forwarded to the Secondary Bus. If a device receives a request on its Secondary Bus and the destination address is located outside of the defined address window, the request is forwarded to the Primary Bus.

C.5.2 Routing by ID

Regardless of the type of PCI Express component, when a Type 1 Configuration Space port receives a TLP on its Primary Bus, the following rules apply:

- If the selected Bus Number is equal to its Secondary Bus number, the Type 1 Configuration request is transformed into a Type 0 Configuration request according to the standard PCI rules.
- In all other cases, the Type 1 Configuration request remains a Type 1 Configuration request and is transferred to the Secondary Bus (internal or external).
- If the selected Bus Number does not fall between the Secondary Bus number and the Subordinate Bus number, the transaction is discarded.

C.5.3 Implicit Routing

Messages do not use Address or ID routing, but rely on other mechanisms referred to as "implicit" because the destination is implied by the routing type (see section 2.2.8 of the *PCI Express Base Specification Revision 2.0,1.1 or 1.0a* for more information). Implicit routing depends upon the inherent knowledge a PCI Express component has relating to upstream and downstream transmissions within the fabric.

"Implicit" Routing types include the following:

- Routed to Root Complex
- Broadcast from Root Complex
- Local (Terminate at Receiver)
- Gathered and routed to Root Complex
- reserved (Terminate at Receiver)

C.5.4 Routing Examples

The following section outlines two hypothetical transactions that correspond to Figure 34.

C.5.4.1 Configuration Write Transaction

Configuration transactions use ID routing (routing based on the Bus, Device, and Function numbers of the destination component).

Table 40: Configuration Write transaction steps

Step	Component	ID	Description
1	Sink Unit / CPU	Dev0 Bus0	The CPU instructs the sink unit to generate a Configuration Write transaction directed to Bus Number 3, Device 2.
2	Rootport	Dev2 PRB0 SEB2 SUB7	The Rootport receives the transaction and determines that the destination Bus (3) is located within its Bus Address window (the range defined by its Secondary Bus number and Subordinate Bus number, in this case 2-7). It accepts the transaction in order to forward it to its Secondary Bus. The target Bus (3) is not equal to the Secondary Bus (2), and the transaction remains a Configuration Type 1 transaction.
3	Switch Upstream Port	Dev0 PRB2 SEB3 SUB7	The Switch's Upstream Port receives the transaction and determines that the destination Bus (3) is located within its Bus Address window and accepts the transaction in order to forward it to its Secondary Bus (the Switch's internal virtual PCI bus). The target Bus (3) is equal to the Secondary Bus (3), and the Type 1 Configuration transaction is transformed into a Type 0 Configuration transaction.
4	Switch Downstream Port	Dev2 PRB3 SEB5 SUB5	Each downstream port in the Switch receives the Type 0 Configuration transaction on virtual Bus 3 and decodes the Device number (2). Port Device 2 recognizes itself as the Completer, accepts the transaction, and generates a Completion TLP using a Transaction ID (Bus 0, Device 0).
5	Switch Upstream Port	Dev0 PRB2 SEB3 SUB7	The Completion TLP is accepted by the Secondary Bus of the Switch's upstream port, which determines that the target Bus Number (0) does not lie within its Bus Address window, and the transaction is forwarded to its Primary Bus.
6	Rootport	Dev2 PRB0 SEB2 SUB7	Likewise, the Rootport forwards the transaction back to Virtual Bus 0.
7	Sink Unit / CPU	Dev0 Bus0	Finally, the sink unit accepts the packet and passes it back to the CPU.

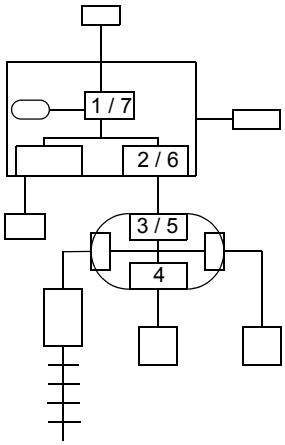

Figure 35: Tracing a Write Transaction through the fabric

C.5.4.2 Memory Read Transaction

Steps 1-6 of the following example describe a Memory Read request, which uses address routing (routing based on either a 64-bit format associated with 4 DW headers or a 32-bit format associated with 3 DW headers). For this example, it is assumed that address windows are correctly defined for each Type 1 Configuration Space component and that the Completer Endpoint Base Address Register (BAR) is correctly defined.

Steps 7-14 describe routing of the Completion transaction generated in response to the Memory Read request. As in the first example (Configuration transaction), Completion transactions use ID routing (routing based on the Bus,

Device, and Function numbers of the destination component).

Table 41: Memory Read transaction steps

Step	Component	ID	Description
1	Endpoint	Dev0 Bus1	The Endpoint component located behind Rootport Device 1 generates a Memory Read request directed to the Endpoint located behind the Switch's Port device 2.
2	Rootport	Dev1 PRB0 SEB1 SUB2	The Secondary Bus of the Rootport (Device 1) receives the transactions, determines that the address lies outside of its address window, and forwards the transaction through its Primary Bus to the Virtual PCI Bus 0.
3	Rootport	Dev2 PRB0 SEB2 SUB7	The Primary Bus of the Rootport (Device 2) determines that the destination address lies within its prefetchable address window and forwards the transactions through its Secondary Bus to the Switch.
4	Switch Upstream Port	Dev0 PRB2 SEB3 SUB7	The Primary Bus of the upstream port of the Switch determines that the destination address lies within its address window and forwards the transactions through its Secondary Bus to the virtual PCI Bus 3.
5	Switch Downstream Port	Dev2 PRB3 SEB5 SUB5	The Primary Bus of downstream port Device 2 of the Switch determines that the destination address lies within its address window and forwards the transactions through its Secondary Bus to the virtual PCI Bus 5.
6	Endpoint	Dev0 PRB5	The Endpoint (the Completer, in this example) receives the transaction and determines that the address corresponds to one of its BARs and accepts the request. The Endpoint responds to the request and generates one or more Completion transactions to send data back to the requester.
Steps 7-14 describe routing of the Completion transaction generated in response to the Memory Read request.			
7	Switch Downstream Port	Dev2 PRB3 SEB5 SUB5	Completion transactions use ID routing, and in this example target the requesting component (Endpoint Bus1 Dev0). The Secondary Bus of the downstream port Device 2 determines that the destination Bus (1) lies outside of its Bus window (5-5) and forwards the transaction through its Primary Bus to the virtual PCI Bus 3.
8	Switch Upstream Port	Dev0 PRB2 SEB3 SUB7	The Secondary Bus of the upstream port determines that the destination Bus (1) lies outside of its Bus window (3-7) and forwards the transaction through its Primary Bus to the Rootport.
9	Rootport	Dev2 PRB0 SEB2 SUB7	The Secondary Bus of the Rootport determines that the destination Bus (1) lies outside of its Bus window (2-7) and forwards the transaction through its Primary Bus to the virtual PCI bus 0.
10	Rootport	Dev1 PRB0 SEB1 SUB2	The Primary Bus of the Rootport determines that the destination Bus (1) lies within its Bus window (1-1) and forwards the transaction through its Secondary Bus to the Endpoint.
11	Endpoint	Dev0 Bus1	The Endpoint recognizes itself as the destination component, verifies the tag of the transaction, and processes the transmitted information.

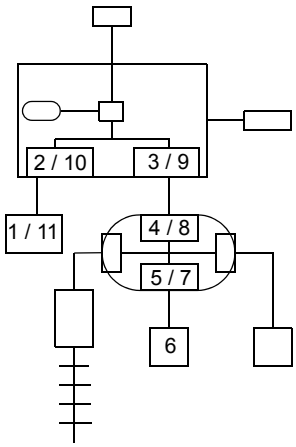

Figure 36: Tracing a Read Transaction through the fabric

C.6 Flow Control

C.6.1 Traffic Classes (TCs)

TCs allow for differentiated services, permitting you to prioritize the flow of certain data through the fabric. A Link must implement at least one TC (TC0) and can implement up to eight TCs, depending on your design.

C.6.2 Virtual Channels (VCs)

VCs allow for multiple independent paths of data flow over a single Link. A Link must implement at least one VC (VC0) and can implement up to eight VCs, according to your design.

The number of VCs initialized across a Link has no relation to the number of lanes implemented by the Link. For example, a x1 component could have eight VCs, and a x4 component could have one VC.

If two components on either side of a Link implement a different number of VCs, only the number of VCs the two components have in common are initialized (see Figure 38).

C.6.3 TCs and VCs

The concept of TCs associated with VCs facilitate data flow in the fabric, allowing you to determine what percentage of a particular Link should be devoted to a particular kind of data transfer. This, in turn, helps to avoid congestion and permits isochronous traffic.

The following figure illustrates the function of Virtual Channels and Traffic Classes in the fabric.

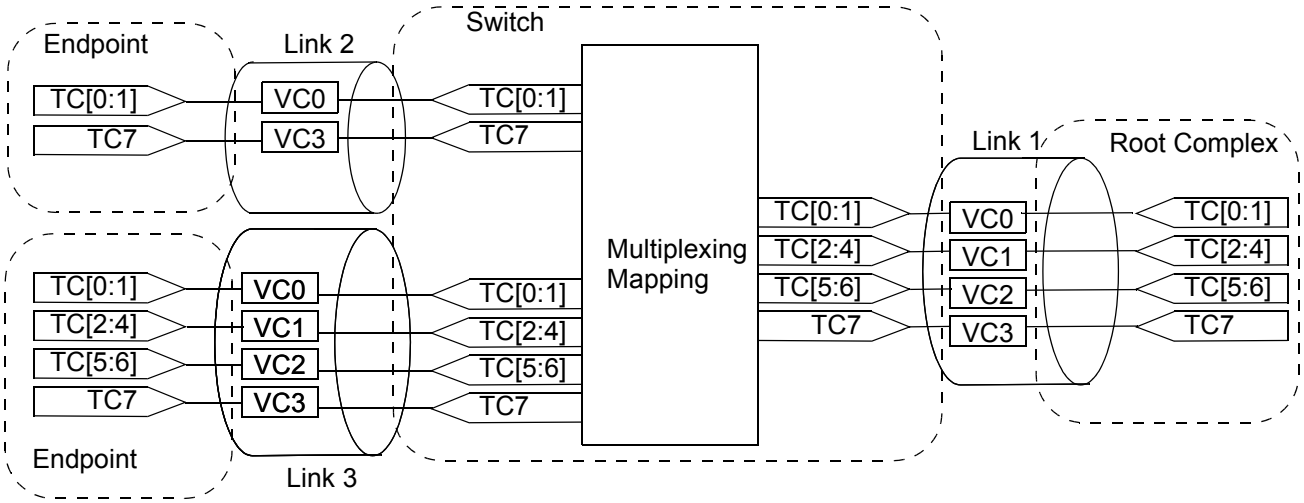

Figure 37: Flow Control through Virtual Channels (VCs) and Traffic Classes (TCs)

Note that the figure above does not specify how many VCs are implemented per component. Consider Link 2, enlarged in the following figure:

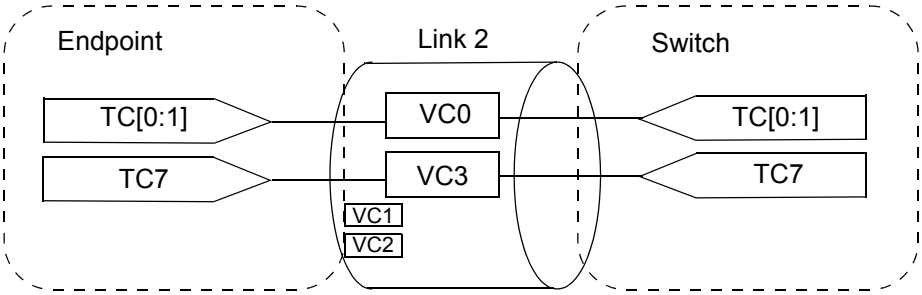

Figure 38: Flow Control through a single Link

The Endpoint component might also have VC1 and VC2 implemented, but the Link only initializes the number of common VCs shared across a Link.

C.6.4 Receive Buffer

Each component commits a certain number of resources (determined by the user) to the Receive buffer. The Receive buffer is located in the Transaction Layer and accepts incoming TLPs from the Link and then sends them to the Application Layer for processing. Receive buffer resources are either implemented per VC or per Link. The following figure enlarges VC3 from Figure 37 and illustrates the various buffers that, taken together, make up the

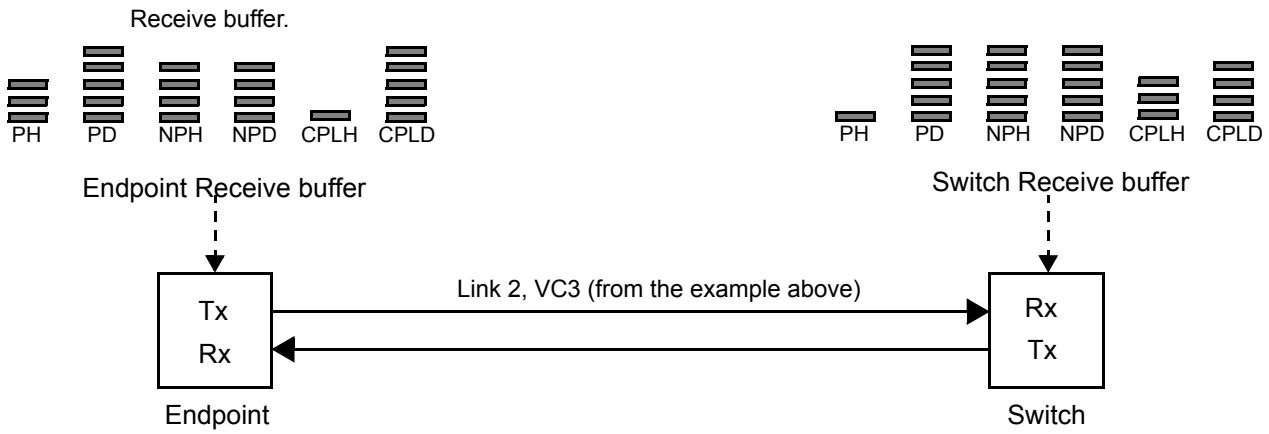

Figure 39: Receive buffers for a Virtual Channel

The Receive buffer stores TLPs based on the type of a transaction, not the TC of a transaction. Types of transactions include Posted transactions, Non-Posted transactions, and Completion transactions.

A transaction always has a header but does not necessarily have data. The Receive buffer accounts for this distinction, maintaining separate resources for the header and data of each type of transaction. To summarize, distinct buffer resources are maintained per initialized VC for each of the following elements:

- Posted transactions, header (PH)
- Posted transactions, data (PD)
- Non-Posted transactions, header (NPH)
- Non-Posted transactions, data (NPD)
- Completion transactions, header (CPLH)
- Completion transactions, data (CPLD)

Note that the Receive buffer levels on one side of a Link have no relation to the Receive buffer levels on the other side of a Link.

The size of the Receive buffer has a significant impact on system performance. The smallest possible size is the Maximum Payload Size (Max_Payload_Size), but the Receive buffer is typically set to at least four times the Maximum Payload Size for the following reasons:

- so that it can store a maximum-sized TLP in the buffer and forward a second TLP to the Application Layer
- so that it can handle multiple Completions of minimum payload size. For example, a component set to handle 16 outstanding requests, each of which might have four corresponding Completion packets, would need a minimum Receive buffer size of 16 X 4 = 64 DW.

C.6.5 Flow Control Credits

A component advertises Buffer space availability with Flow Control credits. FC credits are maintained for each of the six Receive buffers and are transferred using FC Packets, a type of Data Link Layer Packet (DLLP).

The transmitting side of a Link will not send a transaction if the receiving side hasn't advertised enough FC credits (header and data credits) for that particular type of transaction.

The following table offers an example of advertised FC credits for an Endpoint component.

Table 42: Example of an Endpoint's advertised credits at and after Link initialization and the effect on Flow Control

Type of FC credit	Advertised Credits at Link initialization	Advertised Credits at Link initialization + n clock cycles	Transmission permitted...
Posted Header	16	0	No: Sufficient credits for both header and data must be advertised before a packet is transmitted.
Posted Data	128	96	

Table 42: Example of an Endpoint's advertised credits at and after Link initialization and the effect on Flow Control

Type of FC credit	Advertised Credits at Link initialization	Advertised Credits at Link initialization + n clock cycles	Transmission permitted...
Non-Posted Header	16	16	Yes
Non-Posted Data	16	16	
Completion Header	infinite	infinite	Yes: Please see the following section, Deadlock avoidance, for an explanation of infinite credits.
Completion data	infinite	infinite	

The unit of a single FC credit differs between header and data:

- Header (maximum-sized header + digest[1])
  - 4 DWs for Completion transactions
  - 5 DWs for Request transactions
- Data: 4 DWs (16-bytes aligned)

FC Credits are initialized for each VC with maximum credits and then updated periodically as TLPs are extracted from the Receive buffer of the receiving side of a Link. Note that one DLLP can update FC Credits associated with one or more TLPs.

C.6.6 Deadlock Avoidance

As in all Switch and Bridge architectures, deadlock occurs when a component can only proceed when one of its internal resources is freed. PCI Express prevents deadlock with two complementary mechanisms:

- **Reordering of TLPs**: Component resources are freed when Posted Write requests are executed or when a component receives the final Completion TLP in response to a Read request. More specifically, Non-Posted requests must give priority to Posted and Completion transactions.
- **Infinite credits**: Endpoints and Rootports must advertise infinite Completion credits in order to prevent deadlock. To do so, they are not allowed to initiate Read Request transactions if they do not have sufficient Completion buffer space in order to store the maximum number of Completion transactions that might be generated in response to the Read Request.

Note: Other reordering rules govern interactions between Posted requests, Non-Posted requests, and Completion transactions depending on the Traffic Class (TC) and on the "relaxed ordering" bit of the TLP. These additional reordering rules have no effect on deadlock avoidance but can facilitate the flow of global traffic within a fabric.

C.7 Error Handling

C.7.1 Error Recovery

The Data Link Layer handles error recovery. It is based on 32-bit Cyclic Redundant Check (CRC) error detection, TLP sequence number, Replay buffer, and ACK/NAK DLLP exchange. A typical error check might transpire as follows:

- Step 1 Transmitter: On the transmit side of the Link, the Data Link Layer adds a sequence number to the TLP and generates a 32-bit CRC in order to protect the complete packet. The transmitter also stores the sent packet in its Replay buffer.

1. The TLP digest is the end-to-end CRC located just before the LCRC at the end of the TLP. It is signaled with the TD bit set to 1 in the header.

- Step 2 Receiver: On the receive side of the Link, the Data Link Layer verifies the CRC and the sequence number and generates an Acknowledgement / Negative Acknowledgement (ACK/NAK) DLLP in order to report to the transmitter if the packet has been correctly received.
- Step 3 Transmitter: If the received ACK/NAK DLLP is negative (NAK), the transmitter must end its current transmission and re-send the TLP that caused the NAK response. If, on a second try, the packet is successfully transmitted (ACK DLLP), the TLP is purged from the Replay buffer. If the transmitter has not received an ACK reply after three retries, the Link is directed to recovery, that is retraining of the Link.

Note: DLLPs are also protected by a 16-bit CRC, but no acknowledge or error recovery mechanism exists for this check. Instead, DLLPs are periodically generated in order to ensure that the proper information is transmitted.

C.7.2 End-to-End Cyclic Redundancy Check (ECRC)

PCI Express maintains data integrity across two Endpoints of the fabric with the TLP digest (also called ECRC). The TD bit (TLP Digest bit) of the TLP header indicates whether the TLP includes a TLP digest.

Switch devices change packet sequence numbers between ingress and egress ports and can introduce errors as they recalculate the CRC. The TLP digest is an optional feature used in high reliability systems to check for introduced errors.

C.7.3 Replay buffer

The Replay buffer, located in the Data Link Layer and common to all VCs, stores a copy of a transmitted TLP until the transmitted packet is acknowledged by the receiving side of the Link. Each stored TLP includes the Header, an optional data payload (of which the maximum size is determined by the Maximum Payload Size parameter), an optional ECRC, the sequence number, and the LCRC field.

The receiving side of the Link acknowledges reception of a TLP with transmission to the transmitting side of an ACK DLLP. In the case of CRC error on the receiving side, a NAK DLLP is sent to the transmitting side, which retrieves the TLP from the Replay buffer and sends it again.

The user is responsible for setting the size of the Replay buffer, which should be of a sufficient size so that TLP transmission is not delayed because of a full buffer. Replay buffer resources are only freed upon reception of an ACK DLLP, which means that Link latency (associated with DLLP transmission and/or implementation of the Physical Layer) affects the ideal size of the Replay buffer. In general, the Replay buffer size should be at least twice the maximum TLP size.

C.7.4 Completion Timeout

Endpoint and Rootport components use a timeout mechanism (which is design-specific) for failed Read Request transactions in order to report errors to the Rootport (using Error Messages) and to free Completion resources.