

LogiCORE™ IP Endpoint Block Plus v1.9 for PCI Express®

User Guide

UG341 September 19, 2008





Xilinx is disclosing this Specification to you solely for use in the development of designs to operate on Xilinx FPGAs. Except as stated herein, none of the Specification may be copied, reproduced, distributed, republished, downloaded, displayed, posted, or transmitted in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Any unauthorized use of this Specification may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.

Xilinx does not assume any liability arising out of the application or use of the Specification; nor does Xilinx convey any license under its patents, copyrights, or any rights of others. You are responsible for obtaining any rights you may require for your use or implementation of the Specification. Xilinx reserves the right to make changes, at any time, to the Specification as deemed desirable in the sole discretion of Xilinx. Xilinx assumes no obligation to correct any errors contained herein or to advise you of any correction if such be made. Xilinx will not assume any liability for the accuracy or correctness of any engineering or technical support or assistance provided to you in connection with the Specification.

THE SPECIFICATION IS PROVIDED "AS IS" WITH ALL FAULTS, AND THE ENTIRE RISK AS TO ITS FUNCTION AND IMPLEMENTATION IS WITH YOU. YOU ACKNOWLEDGE AND AGREE THAT YOU HAVE NOT RELIED ON ANY ORAL OR WRITTEN INFORMATION OR ADVICE, WHETHER GIVEN BY XILINX, OR ITS AGENTS OR EMPLOYEES. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE SPECIFICATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NONINFRINGEMENT OF THIRD-PARTY RIGHTS.

IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOST DATA AND LOST PROFITS, ARISING FROM OR RELATING TO YOUR USE OF THE SPECIFICATION, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THE TOTAL CUMULATIVE LIABILITY OF XILINX IN CONNECTION WITH YOUR USE OF THE SPECIFICATION, WHETHER IN CONTRACT OR TORT OR OTHERWISE, WILL IN NO EVENT EXCEED THE AMOUNT OF FEES PAID BY YOU TO XILINX HEREUNDER FOR USE OF THE SPECIFICATION. YOU ACKNOWLEDGE THAT THE FEES, IF ANY, REFLECT THE ALLOCATION OF RISK SET FORTH IN THIS AGREEMENT AND THAT XILINX WOULD NOT MAKE AVAILABLE THE SPECIFICATION TO YOU WITHOUT THESE LIMITATIONS OF LIABILITY.

The Specification is not designed or intended for use in the development of on-line control equipment in hazardous environments requiring fail-safe controls, such as in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support, or weapons systems ("High-Risk Applications"). Xilinx specifically disclaims any express or implied warranties of fitness for such High-Risk Applications. You represent that use of the Specification in such High-Risk Applications is fully at your risk.

©2006- 2008 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
10/23/06	1.1	Initial Xilinx release.
2/15/07	2.0	Update core to version 1.2; Xilinx tools 9.1i.
5/17/07	3.0	Update core to version 1.3; updated for PCI-SIG compliance.
8/8/07	4.0	Update core to version 1.4; Xilinx tools 9.2i, Cadence IUS v5.8.
10/10/07	5.0	Update core to version 1.5; Cadence IUS to v6.1.
3/24/08	6.0	Update core to version 1.6; Xilinx tools 10.1.
4/16/08	7.0	Update core to version 1.7.
6/27/08	8.0	Update core to version 1.8.
9/19/08	9.0	Update core to version 1.9.

Table of Contents

Preface: About This Guide

Contents	13
Conventions	14
Typographical	14
Online Document	15

Chapter 1: Introduction

About the Core	17
Designs Using RocketIO Transceivers	17
Recommended Design Experience	17
Additional Core Resources	18
Technical Support	18
Feedback	18
Core	18
Document	18

Chapter 2: Core Overview

Overview	19
Protocol Layers	20
PCI Configuration Space	21
Core Interfaces	24
Transaction Interface	28
Configuration Interface	32
Error Reporting Signals	35

Chapter 3: Generating and Customizing the Core

Using the CORE Generator	37
Basic Parameter Settings	38
Component Name	39
Reference Clock Frequency	39
Number of Lanes	39
Interface Frequency	40
ID Initial Values	40
Class Code	41
Cardbus CIS Pointer	41
Base Address Registers	41
Base Address Register Overview	42
Managing Base Address Register Settings	43
Configuration Register Settings	44
Capabilities Register	45
Device Capabilities Register	45
Link Capabilities Register	45
Slot Clock Configuration	46
MSI Control Register	46

Advanced Settings	47
Transaction Layer Module	48
Advanced Flow Control Credit	48
Advanced Physical Layer	48
Power Management Registers	49
Power Consumption	50
Power Dissipated	50

Chapter 4: Designing with the Core

TLP Format on the Transaction Interface	52
Transmitting Outbound Packets	53
Receiving Inbound Packets	63
Performance Considerations on Receive Transaction Interface	74
Accessing Configuration Space Registers	78
Additional Packet Handling Requirements	82
Reporting User Error Conditions	83
Power Management	87
Generating Interrupt Requests	90
Link Training: 4-Lane and 8-Lane Endpoints	93
Upstream Partner Supports Fewer Lanes	93
Lane Becomes Faulty	93
Clocking and Reset of the Block Plus Core	94
Reset	94
Clocking	94

Chapter 5: Core Constraints

Contents of the User Constraints File	99
Part Selection Constraints: Device, Package, and Speedgrade	99
User Timing Constraints	99
User Physical Constraints	99
Core Pinout and I/O Constraints	99
Core Physical Constraints	100
Core Timing Constraints	100
Required Modifications	100
Device Selection	100
Core I/O Assignments	100
Core Physical Constraints	101
Core Timing Constraints	102
Relocating the Endpoint Block Plus Core	102
Supported Core Pinouts	103

Chapter 6: Hardware Verification

PCI Special Interest Group	107
PCI SIG Integrators List	107

Chapter 7: FPGA Configuration

Configuration Terminology	109
Configuration Access Time	110
Configuration Access Specification Requirements	110
Board Power in Real-world Systems	112
Recommendations	114
FPGA Configuration Times for Virtex-5 Devices	114
Sample Problem Analysis	117
Workarounds for Closed Systems	118

Appendix A: Programmed Input Output

Example Design

System Overview	119
PIO Hardware	120
Base Address Register Support	121
TLP Data Flow	122
PIO File Structure	123
PIO Application	125
Receive Path	126
Transmit Path	128
Endpoint Memory	129
PIO Operation	131
PIO Read Transaction	131
PIO Write Transaction	132
Device Utilization	132
Dual Core Example Design	132
Summary	133

Appendix B: Downstream Port Model Test Bench

Architecture	136
Simulating the Design	137
Test Selection	138
VHDL Test Selection	139
Verilog Test Selection	139
VHDL and Verilog Downstream Port Model Differences	139
Waveform Dumping	140
VHDL Flow	140
Verilog Flow	141
Output Logging	141
Parallel Test Programs	141
Test Description	142
Test Program: pio_writeReadBack_test0	144
Expanding the Downstream Port Model	145
Downstream Port Model TPI Task List	145

Appendix C: Migration Considerations

Transaction Interfaces 155

Configuration Interface 155

System and PCI Express Interfaces 156

Configuration Space 156

Schedule of Figures

Chapter 2: Core Overview

<i>Figure 2-1: Top-level Functional Blocks and Interfaces</i>	20
---	----

Chapter 3: Generating and Customizing the Core

<i>Figure 3-1: Block Plus Parameters: Screen 1</i>	38
<i>Figure 3-2: Block Plus Parameters: Screen 2</i>	39
<i>Figure 3-3: BAR Options: Screen 3</i>	41
<i>Figure 3-4: BAR Options: Screen 4</i>	42
<i>Figure 3-5: Configuration Settings: Screen 5</i>	44
<i>Figure 3-6: Configuration Settings: Screen 6</i>	45
<i>Figure 3-7: Advanced Settings: Screen 7</i>	47
<i>Figure 3-8: Advanced Settings: Screen 8</i>	48

Chapter 4: Designing with the Core

<i>Figure 4-1: PCI Express Base Specification Byte Order</i>	52
<i>Figure 4-2: Endpoint Block Plus Byte Order</i>	52
<i>Figure 4-3: TLP 3-DW Header without Payload</i>	54
<i>Figure 4-4: TLP with 4-DW Header without Payload</i>	55
<i>Figure 4-5: TLP with 3-DW Header with Payload</i>	56
<i>Figure 4-6: TLP with 4-DW Header with Payload</i>	57
<i>Figure 4-7: Back-to-back Transaction on Transmit Transaction Interface</i>	58
<i>Figure 4-8: Source Throttling on the Transmit Data Path</i>	59
<i>Figure 4-9: Destination Throttling of the Endpoint Transmit Transaction Interface</i> ...	60
<i>Figure 4-10: Source Driven Transaction Discontinue on Transmit Interface</i>	61
<i>Figure 4-11: TLP 3-DW Header without Payload</i>	64
<i>Figure 4-12: TLP 4-DW Header without Payload</i>	65
<i>Figure 4-13: TLP 3-DW Header with Payload</i>	66
<i>Figure 4-14: TLP 4-DW Header with Payload</i>	67
<i>Figure 4-15: User Application Throttling Receive TLP</i>	68
<i>Figure 4-16: Receive Back-To-Back Transactions</i>	69
<i>Figure 4-17: User Application Throttling Back-to-Back TLPs</i>	69
<i>Figure 4-18: Packet Re-ordering on Receive Transaction Interface</i>	70
<i>Figure 4-19: Receive Transaction Data Poisoning</i>	72
<i>Figure 4-20: BAR Target Determination using trn_rbar_hit</i>	73
<i>Figure 4-21: Options for Allocating Completion Buffer Space</i>	75
<i>Figure 4-22: Example Configuration Space Access</i>	82
<i>Figure 4-23: Signaling Unsupported Request for Non-Posted TLP</i>	85
<i>Figure 4-24: Signaling Unsupported Request for Posted TLP</i>	85

<i>Figure 4-25: Signaling Locked Unsupported Request for Locked Non-Posted TLP. . . .</i>	86
<i>Figure 4-26: Requesting Interrupt Service: MSI and Legacy Mode</i>	92
<i>Figure 4-27: Scaling of 4-lane Endpoint Core from 4-lane to 1-lane Operation</i>	93
<i>Figure 4-28: Embedded System Using 100 MHz Reference Clock</i>	95
<i>Figure 4-29: Embedded System Using 250 MHz Reference Clock</i>	96
<i>Figure 4-30: Open System Add-In Card Using 100 MHz Reference Clock.</i>	96
<i>Figure 4-31: Open System Add-In Card Using 250 MHz Reference Clock.</i>	97

Chapter 7: FPGA Configuration

<i>Figure 7-1: Power Up</i>	111
<i>Figure 7-2: ATX Power Supply</i>	113
<i>Figure 7-3: Default Configuration Time on LX50T Device (2 MHz Clock)</i>	117
<i>Figure 7-4: Fast Configuration Time on LX50T Device (50 MHz Clock).</i>	118

Appendix A: Programmed Input Output Example Design

<i>Figure A-1: System Overview</i>	120
<i>Figure A-2: PIO Design Components.</i>	124
<i>Figure A-3: PIO 64-bit Application</i>	125
<i>Figure A-4: PIO 32-bit Application</i>	125
<i>Figure A-5: Rx Engines</i>	126
<i>Figure A-6: Tx Engines</i>	128
<i>Figure A-7: EP Memory Access</i>	129
<i>Figure A-8: Back-to-Back Read Transactions</i>	131
<i>Figure A-9: Back-to-Back Write Transactions</i>	132

Appendix B: Downstream Port Model Test Bench

<i>Figure B-1: Downstream Port Model and Top-level Endpoint</i>	136
---	-----

Schedule of Tables

Chapter 2: Core Overview

Table 2-1: Product Overview	19
Table 2-2: PCI Configuration Space Header	22
Table 2-3: System Interface Signals	24
Table 2-4: PCI Express Interface Signals for the 1-lane Endpoint Core.....	24
Table 2-5: PCI Express Interface Signals for the 4-lane Endpoint Core.....	25
Table 2-6: PCI Express Interface Signals for the 8-lane Endpoint Core.....	26
Table 2-7: Common Transaction Interface Signals	28
Table 2-8: Transaction Transmit Interface Signals.....	29
Table 2-9: Receive Transaction Interface Signals.....	30
Table 2-10: Configuration Interface Signals	32
Table 2-11: User Application Error-Reporting Signals	35

Chapter 3: Generating and Customizing the Core

Table 3-1: Lane Width	39
Table 3-2: Default and Alternate Lane Width Frequency.....	40
Table 3-3: Dynamic Reconfiguration Ports	49

Chapter 4: Designing with the Core

Table 4-1: <code>trn_tbuf_av[3:0]</code> Bits	62
Table 4-2: <code>trn_rbar_hit_n</code> to Base Address Register Mapping	73
Table 4-3: Completion Buffer Space Management Solutions	74
Table 4-4: Command and Status Registers Mapped to the Configuration Port.....	78
Table 4-5: Bit Mapping on Header Status Register	79
Table 4-6: Bit Mapping on Header Command Register	79
Table 4-7: Bit Mapping on PCI Express Device Status Register	80
Table 4-8: Bit Mapping of PCI Express Device Control Register	80
Table 4-9: Bit Mapping of PCI Express Link Status Register.....	81
Table 4-10: Bit Mapping of PCI Express Link Control Register.....	81
Table 4-11: User-indicated Error Signaling	84
Table 4-12: Possible Error Conditions for TLPs Received by the User Application....	84
Table 4-13: Interrupt Signalling.....	90
Table 4-14: Legacy Interrupt Mapping	92

Chapter 5: Core Constraints

<i>Table 5-1: Corresponding Lane Numbers and GTP Transceiver Locations</i>	101
<i>Table 5-2: Supported Core Pinouts Virtex-5 LXT /SXT FPGAs</i>	103
<i>Table 5-3: Supported Core Pinouts Virtex-5 FXT FPGA</i>	104
<i>Table 5-4: Supported Core Pinouts Virtex-5 TXT FPGA</i>	106

Chapter 6: Hardware Verification

<i>Table 6-1: Platforms Tested</i>	107
--	-----

Chapter 7: FPGA Configuration

<i>Table 7-1: TPVPERL Specification</i>	111
<i>Table 7-2: Configuration Time Matrix (ATX Motherboards): Virtex-5 FPGA Bitstream Transfer Time in Milliseconds</i>	115
<i>Table 7-3: Configuration Time Matrix (Generic Platforms: Non-ATX Motherboards): Virtex-5 FPGA Bitstream Transfer Time in Milliseconds</i>	116

Appendix A: Programmed Input Output Example Design

<i>Table A-1: TLP Traffic Types</i>	121
<i>Table A-2: PIO Design File Structure</i>	123
<i>Table A-3: PIO Configuration</i>	124
<i>Table A-4: Rx Engine: Read Outputs</i>	126
<i>Table A-5: Rx Engine: Write Outputs</i>	127
<i>Table A-6: Tx Engine Inputs</i>	128
<i>Table A-7: EP Memory: Write Inputs</i>	130
<i>Table A-8: EP Memory: Read Inputs</i>	130
<i>Table A-9: PIO Design FPGA Resources</i>	132

Appendix B: Downstream Port Model Test Bench

<i>Table B-1: Downstream Port Model Provided Tests</i>	138
<i>Table B-2: Simulator Dump File Format</i>	140
<i>Table B-3: Test Setup Tasks</i>	145
<i>Table B-4: TLP Tasks</i>	146
<i>Table B-5: BAR Initialization Tasks</i>	150
<i>Table B-6: Example PIO Design Tasks</i>	151
<i>Table B-7: Expectation Tasks</i>	152

About This Guide

The *Endpoint Block Plus for PCI Express® User Guide* describes the function and operation of the Endpoint Block Plus for PCI Express (PCIe®) core, including how to design, customize, and implement the core.

Contents

This guide contains the following chapters:

- [Preface, “About this Guide”](#) introduces the organization and purpose of this user guide and the conventions used in this document.
- [Chapter 1, “Introduction,”](#) describes the core and related information, including recommended design experience, additional resources, technical support, and submitting feedback to Xilinx.
- [Chapter 2, “Core Overview,”](#) describes the main components of the Block Plus core architecture.
- [Chapter 3, “Generating and Customizing the Core,”](#) describes how to use the graphical user interface (GUI) to configure the Block Plus using the Xilinx CORE Generator™ software.
- [Chapter 4, “Designing with the Core,”](#) provides instructions on how to design a device using the Block Plus core.
- [Chapter 5, “Core Constraints,”](#) discusses the required and optional constraints for the Block Plus core.
- [Appendix A, “Programmed Input Output Example Design,”](#) describes the Programmed Input Output (PIO) example design for use with the core.
- [Appendix B, “Downstream Port Model Test Bench,”](#) describes the test bench environment, which provides a test program interface for use with the PIO example design.
- [Appendix C, “Migration Considerations”](#) defines the differences in behaviors and options between the Block Plus and the Endpoint for PCI Express (versions 3.5 and earlier).

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
Courier font	Messages, prompts, and program files that the system displays	speed grade: - 100
Courier bold	Literal commands you enter in a syntactical statement	ngdbuild design_name
<i>Italic font</i>	References to other manuals	See the <i>User Guide</i> for details.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are <i>not</i> connected.
Dark Shading	Items that are either reserved or not supported	This feature is not supported
Square brackets []	An optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.	ngdbuild [option_name] design_name
Braces { }	A list of items from which you must choose one or more	lowpwr = {on off}
Vertical bar	Separates items in a list of choices	lowpwr = {on off}
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Omitted repetitive material	allow block block_name loc1 loc2... locn;
Notations	The prefix '0x' or the suffix 'h' indicate hexadecimal notation	A read of address 0x00112975 returned 45524943h.
	An '_n' means the signal is active low	usr_teof_n is active low.

Online Document

The following linking conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section “ Additional Resources ” for details. See “ Title Formats ” in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the <i>Virtex-II FPGA Handbook</i> .
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

Introduction

This chapter introduces the Endpoint Block Plus for PCI Express and provides related information including system requirements, recommended design experience, additional core resources, technical support, and submitting feedback to Xilinx.

About the Core

The Endpoint Block Plus for PCIe solution from Xilinx is a reliable, high-bandwidth, scalable serial interconnect building block for use with the Virtex®-5 LXT, SXT, FXT, and TXT FPGA architecture. The core instantiates the Virtex-5 Integrated Block for PCI Express found in Virtex-5 LXT, SXT, FXT, and TXT platforms, and supports both Verilog®-HDL and VHDL.

The Endpoint Block Plus core is a Xilinx CORE Generator™ IP core, included in the latest IP Update on the Xilinx IP Center. For detailed information about the core, see the Endpoint for Block Plus [product page](#). For information about licensing options, see “Chapter 2, Licensing the Core,” in the *LogiCORE IP Endpoint Block Plus for PCI Express Getting Started Guide*.

Designs Using RocketIO Transceivers

RocketIO™ transceivers are defined by device family in the following way:

- For Virtex-5 LXT and SXT FPGAs, RocketIO GTP transceivers
- For Virtex-5 FXT and TXT FPGAs, RocketIO GTX transceivers

Recommended Design Experience

Although the Block Plus core is a fully verified solution, the challenge associated with implementing a complete design varies depending on the configuration and functionality of the application. For best results, previous experience building high-performance, pipelined FPGA designs using Xilinx implementation software and User Constraints Files (UCF) is recommended.

Additional Core Resources

For detailed information and updates about the Endpoint Block Plus core, see the following documents:

- *LogiCORE IP Endpoint Block Plus for PCI Express Plus Data Sheet* (available from the product page)
- *LogiCORE IP Endpoint Block Plus for PCI Express Getting Started Guide* (available from the product lounge)
- *LogiCORE IP Endpoint Block Plus for PCI Express Release Notes* (available from the product lounge)
- [Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs User Guide](#) (UG197)

Additional information and resources related to the PCI Express technology are available from the following web sites:

- [PCI Express at PCI-SIG](#)
- [PCI Express Developer's Forum](#)

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team of engineers with expertise using the Endpoint Block Plus for PCIe core.

Xilinx will provide technical support for use of this product as described in the [Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs User Guide](#) and the *LogiCORE IP Endpoint Block Plus for PCI Express Getting Started Guide*. Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the core and the accompanying documentation.

Core

For comments or suggestions about the core, please submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Document

For comments or suggestions about this document, please submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

Core Overview

This chapter describes the main components of the Endpoint Block Plus for PCI Express core architecture.

Overview

Table 2-1 defines the Endpoint Block Plus for PCIe solutions.

Table 2-1: Product Overview

Product Name	FPGA Architecture	User Interface Width	Lane Widths Supported	PCI Express Base Specification Compliance
1-lane Endpoint Block Plus	Virtex-5	64	x1	v1.1
4-lane Endpoint Block Plus	Virtex-5	64	x1, x2 ¹ , x4 ²	v1.1
8-lane Endpoint Block Plus	Virtex-5	64	x1, x2 ¹ , x4, x8 ²	v1.1

1. 2-lane operation is supported only through a 4-lane or 8-lane core training down to 2-lane operation.
2. See “[Link Training: 4-Lane and 8-Lane Endpoints](#)” for additional information.

The Endpoint Block Plus core internally instances the Virtex-5 Integrated Endpoint Block. See the [Virtex-5 Integrated Endpoint Block for PCI Express Designs User Guide](#) (UG 197) for information about the internal architecture of the block. The integrated block follows the *PCI Express Base Specification* layering model, which consists of the Physical, Data Link, and Transaction Layers.

Figure 2-1 illustrates the interfaces to the core, as defined below:

- System (SYS) interface
- PCI Express (PCI_EXP) interface
- Configuration (CFG) interface
- Transaction (TRN) interface

The core uses packets to exchange information between the various modules. Packets are formed in the Transaction and Data Link Layers to carry information from the transmitting component to the receiving component. Necessary information is added to the packet being transmitted, which is required to handle the packet at those layers. At the receiving end, each layer of the receiving element processes the incoming packet, strips the relevant information and forwards the packet to the next layer.

As a result, the received packets are transformed from their Physical Layer representation to their Data Link Layer representation and the Transaction Layer representation.

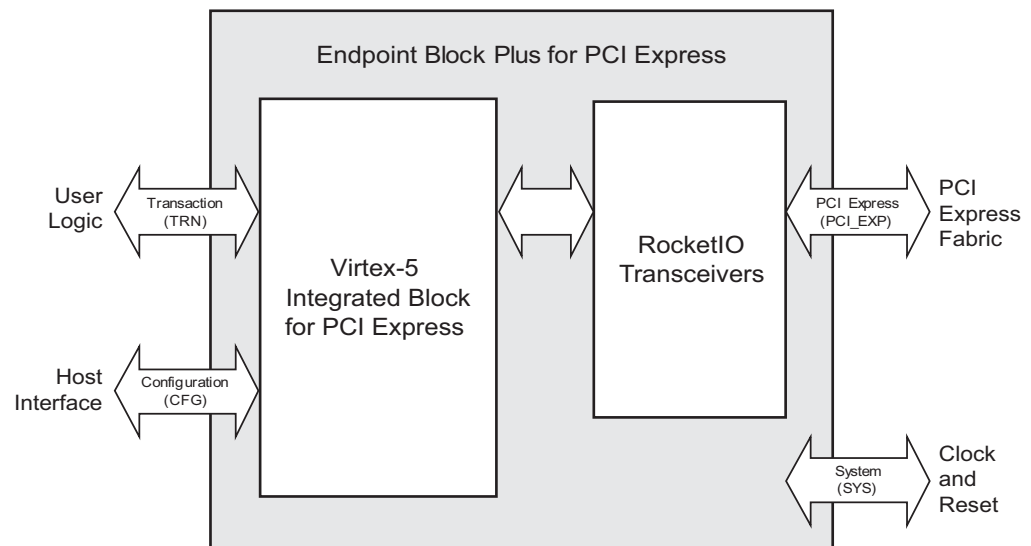


Figure 2-1: Top-level Functional Blocks and Interfaces

Protocol Layers

The functions of the protocol layers, as defined by the *PCI Express Base Specification*, include generation and processing of Transaction Layer Packets (TLPs), flow control management, initialization and power management, data protection, error checking and retry, physical link interface initialization, maintenance and status tracking, serialization, de-serialization and other circuitry for interface operation. Each layer is defined below.

Transaction Layer

The Transaction Layer is the upper layer of the PCI Express architecture, and its primary function is to accept, buffer, and disseminate Transaction Layer packets or TLPs. TLPs communicate information through the use of memory, IO, configuration, and message transactions. To maximize the efficiency of communication between devices, the Transaction Layer enforces PCI-compliant Transaction ordering rules and manages TLP buffer space via credit-based flow control.

Data Link Layer

The Data Link Layer acts as an intermediate stage between the Transaction Layer and the Physical Layer. Its primary responsibility is to provide a reliable mechanism for the exchange of TLPs between two components on a link.

Services provided by the Data Link Layer include data exchange (TLPs), error detection and recovery, initialization services and the generation and consumption of Data Link Layer Packets (DLLPs). DLLPs are used to transfer information between Data Link Layers of two directly connected components on the link. DLLPs convey information such as Power Management, Flow Control, and TLP acknowledgments.

Physical Layer

The Physical Layer interfaces the Data Link Layer with signalling technology for link data interchange, and is subdivided into the Logical sub-block and the Electrical sub-block.

- The Logical sub-block is responsible for framing and de-framing of TLPs and DLLPs. It also implements the Link Training and Status State machine (LTSSM) which handles link initialization, training, and maintenance. Scrambling, de-scrambling and 8b/10b encoding and decoding of data is also performed in this sub-block.
- The Electrical sub-block defines the input and output buffer characteristics that interfaces the device to the PCIe link.

The Physical Layer also supports Lane Reversal (for multi-lane designs) and Lane Polarity Inversion, as indicated in the *PCI Express Base Specification rev 1.1* requirement.

Configuration Management

The Configuration Management layer maintains the PCI Type0 Endpoint configuration space and supports the following features:

- Implements PCI Configuration Space
- Supports Configuration Space accesses
- Power Management functions
- Implements error reporting and status functionality
- Implements packet processing functions
 - Receive
 - Configuration Reads and Writes
 - Transmit
 - Completions with or without data
 - TLM Error Messaging
 - User Error Messaging
 - Power Management Messaging/Handshake
- Implements MSI and INTx interrupt emulation
- Implements the Device Serial Number Capability in the PCIe Extended Capability space

PCI Configuration Space

The configuration space consists of three primary parts, illustrated in [Table 2-4](#). These include the following:

- Legacy PCI v3.0 Type 0 Configuration Header
- Legacy Extended Capability Items
 - PCIe Capability Item
 - Power Management Capability Item
 - Message Signaled Interrupt Capability Item

- PCIe Extended Capabilities
 - Device Serial Number Extended Capability Structure

The core implements three legacy extended capability items. The remaining legacy extended capability space from address 0x6C to 0xFF is reserved. The core returns 0x00000000 when this address range is read.

The core also implements one PCIe Extended Capability. The remaining PCIe Extended Capability space from addresses 0x10C to 0xFFF is reserved. The core returns 0x00000000 for reads to this range; writes are ignored.

Table 2-2: PCI Configuration Space Header

31	16 15			0
Device ID		Vendor ID		000h
Status		Command		004h
Class Code			Rev ID	008h
BIST	Header	Lat Timer	Cache Ln	00Ch
Base Address Register 0				010h
Base Address Register 1				014h
Base Address Register 2				018h
Base Address Register 3				01Ch
Base Address Register 4				020h
Base Address Register 5				024h
Cardbus CIS Pointer				028h
Subsystem ID		Subsystem Vendor ID		02Ch
Expansion ROM Base Address				030h
Reserved			CapPtr	034h
Reserved				038h
Max Lat	Min Gnt	Intr Pin	Intr Line	03Ch
PM Capability		NxtCap	PM Cap	040h
Data	BSE	PMCSR		044h
MSI Control		NxtCap	MSI Cap	048h
Message Address (Lower)				04Ch
Message Address (Upper)				050h
Reserved		Message Data		054h
Reserved Legacy Configuration Space (Returns 0x00000000)				058h-05Ch
PE Capability		NxtCap	PE Cap	060h
PCI Express Device Capabilities				064h
Device Status		Device Control		068h
PCI Express Link Capabilities				06Ch
Link Status		Link Control		070h
Reserved Legacy Configuration Space (Returns 0x00000000)				074h-0FFh

Table 2-2: PCI Configuration Space Header

Next Cap	Cap. Ver.	PCI Exp. Capability	100h
PCI Express Device Serial Number (1st)			104h
PCI Express Device Serial Number (2nd)			108h
Reserved Extended Configuration Space (Returns 0x00000000)			10Ch-FFFh

Core Interfaces

The Endpoint Block Plus core for PCIe includes top-level signal interfaces that have sub-groups for the receive direction, transmit direction, and signals common to both directions.

System Interface

The System (SYS) interface consists of the system reset signal, `sys_reset_n`, the system clock signal, `sys_clk`, and a free running reference clock output signal, `refclkout`, as described in [Table 2-3](#).

Table 2-3: System Interface Signals

Function	Signal Name	Direction	Description
System Reset	<code>sys_reset_n</code>	Input	Asynchronous, active low signal.
System Clock	<code>sys_clk</code>	Input	Reference clock: 100 MHz or 250 MHz.
Reference Clock	<code>refclkout</code>	Output	A free running reference clock output, based on the reference clock.

The system reset signal is an asynchronous active-low input. The assertion of `sys_reset_n` causes a hard reset of the entire core. The system input clock must be either 100 MHz or 250 MHz, as selected in the CORE Generator GUI. For important information about resetting and clocking the Endpoint Block Plus core, see [“Clocking and Reset of the Block Plus Core,”](#) [page 94](#) and the [Virtex-5 Integrated Endpoint Block for PCI Express Designs User Guide](#).

PCI Express Interface

The PCI Express (PCI_EXP) interface consists of differential transmit and receive pairs organized in multiple lanes. A PCI Express lane consists of a pair of transmit differential signals {`pci_exp_txp`, `pci_exp_txn`} and a pair of receive differential signals {`pci_exp_rxp`, `pci_exp_rxn`}. The 1-lane core supports only Lane 0, the 4-lane core supports lanes 0-3, and the 8-lane core supports lanes 0-7. Transmit and receive signals of the PCI_EXP interface are defined in [Tables 2-4](#), [2-5](#), and [2-6](#).

Table 2-4: PCI Express Interface Signals for the 1-lane Endpoint Core

Lane Number	Name	Direction	Description
0	<code>pci_exp_txp0</code>	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
0	<code>pci_exp_txn0</code>	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
0	<code>pci_exp_rxp0</code>	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
0	<code>pci_exp_rxn0</code>	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)

Table 2-5: PCI Express Interface Signals for the 4-lane Endpoint Core

Lane Number	Name	Direction	Description
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
0	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
0	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
0	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
1	pci_exp_txn1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (-)
1	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
1	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (-)
2	pci_exp_txp2	Output	PCI Express Transmit Positive: Serial Differential Output 2 (+)
2	pci_exp_txn2	Output	PCI Express Transmit Negative: Serial Differential Output 2 (-)
2	pci_exp_rxp2	Input	PCI Express Receive Positive: Serial Differential Input 2 (+)
2	pci_exp_rxn2	Input	PCI Express Receive Negative: Serial Differential Input 2 (-)
3	pci_exp_txp3	Output	PCI Express Transmit Positive: Serial Differential Output 3 (+)
3	pci_exp_txn3	Output	PCI Express Transmit Negative: Serial Differential Output 3 (-)
3	pci_exp_rxp3	Input	PCI Express Receive Positive: Serial Differential Input 3 (+)
3	pci_exp_rxn3	Input	PCI Express Receive Negative: Serial Differential Input 3 (-)

Table 2-6: PCI Express Interface Signals for the 8-lane Endpoint Core

Lane Number	Name	Direction	Description
0	pci_exp_txp0	Output	PCI Express Transmit Positive: Serial Differential Output 0 (+)
0	pci_exp_txn0	Output	PCI Express Transmit Negative: Serial Differential Output 0 (-)
0	pci_exp_rxp0	Input	PCI Express Receive Positive: Serial Differential Input 0 (+)
0	pci_exp_rxn0	Input	PCI Express Receive Negative: Serial Differential Input 0 (-)
1	pci_exp_txp1	Output	PCI Express Transmit Positive: Serial Differential Output 1 (+)
1	pci_exp_txn1	Output	PCI Express Transmit Negative: Serial Differential Output 1 (-)
1	pci_exp_rxp1	Input	PCI Express Receive Positive: Serial Differential Input 1 (+)
1	pci_exp_rxn1	Input	PCI Express Receive Negative: Serial Differential Input 1 (-)
2	pci_exp_txp2	Output	PCI Express Transmit Positive: Serial Differential Output 2 (+)
2	pci_exp_txn2	Output	PCI Express Transmit Negative: Serial Differential Output 2 (-)
2	pci_exp_rxp2	Input	PCI Express Receive Positive: Serial Differential Input 2 (+)
2	pci_exp_rxn2	Input	PCI Express Receive Negative: Serial Differential Input 2 (-)
3	pci_exp_txp3	Output	PCI Express Transmit Positive: Serial Differential Output 3 (+)
3	pci_exp_txn3	Output	PCI Express Transmit Negative: Serial Differential Output 3 (-)
3	pci_exp_rxp3	Input	PCI Express Receive Positive: Serial Differential Input 3 (+)
3	pci_exp_rxn3	Input	PCI Express Receive Negative: Serial Differential Input 3 (-)
4	pci_exp_txp4	Output	PCI Express Transmit Positive: Serial Differential Output 4 (+)
4	pci_exp_txn4	Output	PCI Express Transmit Negative: Serial Differential Output 4 (-)
4	pci_exp_rxp4	Input	PCI Express Receive Positive: Serial Differential Input 4 (+)
4	pci_exp_rxn4	Input	PCI Express Receive Negative: Serial Differential Input 4 (-)

Table 2-6: PCI Express Interface Signals for the 8-lane Endpoint Core

Lane Number	Name	Direction	Description
5	pci_exp_txp5	Output	PCI Express Transmit Positive: Serial Differential Output 5 (+)
5	pci_exp_txn5	Output	PCI Express Transmit Negative: Serial Differential Output 5 (-)
5	pci_exp_rxp5	Input	PCI Express Receive Positive: Serial Differential Input 5 (+)
5	pci_exp_rxn5	Input	PCI Express Receive Negative: Serial Differential Input 5 (-)
6	pci_exp_txp6	Output	PCI Express Transmit Positive: Serial Differential Output 6 (+)
6	pci_exp_txn6	Output	PCI Express Transmit Negative: Serial Differential Output 6 (-)
6	pci_exp_rxp6	Input	PCI Express Receive Positive: Serial Differential Input 6 (+)
6	pci_exp_rxn6	Input	PCI Express Receive Negative: Serial Differential Input 6 (-)
7	pci_exp_txp7	Output	PCI Express Transmit Positive: Serial Differential Output 7 (+)
7	pci_exp_txn7	Output	PCI Express Transmit Negative: Serial Differential Output 7 (-)
7	pci_exp_rxp7	Input	PCI Express Receive Positive: Serial Differential Input 7 (+)
7	pci_exp_rxn7	Input	PCI Express Receive Negative: Serial Differential Input 7 (-)

Transaction Interface

The Transaction (TRN) interface provides a mechanism for the user design to generate and consume TLPs. The signal names and signal descriptions for this interface are shown in [Tables 2-7, 2-8, and 2-9](#).

Common TRN Interface

[Table 2-7](#) defines and describes the common TRN interface signals.

Table 2-7: Common Transaction Interface Signals

Name	Direction	Description												
trn_clk	Output	<p>Transaction Clock: Transaction and Configuration interface operations are referenced-to and synchronous-with the rising edge of this clock. trn_clk is unavailable when the core sys_reset_n is held asserted. trn_clk is guaranteed to be stable at the nominal operating frequency once the core deasserts trn_reset_n. The trn_clk clock output is a fixed frequency configured in the CORE Generator. trn_clk does not shift frequencies in case of link recovery or training down.</p> <table> <tr> <th>Product</th><th>Recommended Frequency (MHz)</th><th>Optional Frequency (MHz)</th></tr> <tr> <td>1-lane Endpoint Block Plus</td><td>62.5</td><td>125.0</td></tr> <tr> <td>4-lane Endpoint Block Plus</td><td>125.0</td><td>250.0</td></tr> <tr> <td>8-lane Endpoint Block Plus</td><td>250.0</td><td>125.0</td></tr> </table>	Product	Recommended Frequency (MHz)	Optional Frequency (MHz)	1-lane Endpoint Block Plus	62.5	125.0	4-lane Endpoint Block Plus	125.0	250.0	8-lane Endpoint Block Plus	250.0	125.0
Product	Recommended Frequency (MHz)	Optional Frequency (MHz)												
1-lane Endpoint Block Plus	62.5	125.0												
4-lane Endpoint Block Plus	125.0	250.0												
8-lane Endpoint Block Plus	250.0	125.0												
trn_reset_n	Output	<p>Transaction Reset: Active low. User logic interacting with the Transaction and Configuration interfaces must use trn_reset_n to return to their quiescent states. trn_reset_n is deasserted synchronously with respect to trn_clk, trn_reset_n is deasserted and is asserted asynchronously with sys_reset_n assertion. Note that trn_reset_n is not asserted for core in-band reset events like Hot Reset or Link Disable.</p>												
trn_lnk_up_n	Output	<p>Transaction Link Up: Active low. Transaction link-up is asserted when the core and the connected upstream link partner port are ready and able to exchange data packets. Transaction link-up is deasserted when the core and link partner are attempting to establish communication, and when communication with the link partner is lost due to errors on the transmission channel. When the core is driven to Hot Reset and Link Disable states by the link partner, trn_lnk_up_n is deasserted and all TLPs stored in the endpoint core are lost.</p>												

Transmit TRN Interface

Table 2-8 defines the transmit (Tx) TRN interface signals.

Table 2-8: Transaction Transmit Interface Signals

Name	Direction	Description
trn_tsof_n	Input	Transmit Start-of-Frame (SOF): Active low. Signals the start of a packet. Valid only along with assertion of trn_tsrc_rdy_n.
trn_teof_n	Input	Transmit End-of-Frame (EOF): Active low. Signals the end of a packet. Valid only along with assertion of trn_tsrc_rdy_n.
trn_td[63:0]	Input	Transmit Data: Packet data to be transmitted.
trn_trem_n[7:0]	Input	Transmit Data Remainder: Valid only if both trn_teof_n, trn_tsrc_rdy_n, and trn_tdst_rdy_n are asserted. Legal values are: <ul style="list-style-type: none"> 0000_0000b = packet data on all of trn_td[63:0] 0000_1111b = packet data only on trn_td[63:32]
trn_tsrc_rdy_n	Input	Transmit Source Ready: Active low. Indicates that the User Application is presenting valid data on trn_td[63:0]. Active low. Indicates that the User Application is presenting valid data on trn_td[63:0].
trn_tdst_rdy_n	Output	Transmit Destination Ready: Active low. Indicates that the core is ready to accept data on trn_td[63:0]. The simultaneous assertion of trn_tsrc_rdy_n and trn_tdst_rdy_n marks the successful transfer of one data beat on trn_td[63:0].
trn_tsrc_dsc_n	Input	Transmit Source Discontinue: Can be asserted any time starting on the first cycle after SOF to EOF, inclusive.
trn_tdst_dsc_n	Output	Transmit Destination Discontinue: Active low. Indicates that the core is aborting the current packet. Asserted when the physical link is going into reset. Not supported; signal is tied high.
trn_tbuf_av [3:0]	Output	Transmit Buffers Available: Indicates transmit buffer availability in the core. Each bit of trn_tbuf_av corresponds to one of the following credit queues: <ul style="list-style-type: none"> trn_tbuf_av[0] ≥ Non Posted Queue trn_tbuf_av[1] ≥ Posted Queue trn_tbuf_av[2] ≥ Completion Queue trn_tbuf_av[3] ≥ Look-Ahead Completion Queue A value of 1 indicates that the core can accept at least 1 TLP of that particular credit class. A value of 0 indicates no buffer availability in the particular queue. trn_tbuf_av[3] indicates that the core may be about to run out of Completion Queue buffers. If this is deasserted, performance can be optimized by sending a Posted or Non-Posted TLP instead of a Completion TLP. If a Completion TLP is sent when this signal is deasserted, the core may be forced to stall the TRN interface for all TLP types, until new Completion Queue buffers become available.

Receive TRN Interface

Table 2-9 defines the receive (Rx) TRN interface signals.

Table 2-9: Receive Transaction Interface Signals

Name	Direction	Description
trn_rsof_n	Output	Receive Start-of-Frame (SOF): Active low. Signals the start of a packet. Valid only if trn_rsrc_rdy_n is also asserted.
trn_reof_n	Output	Receive End-of-Frame (EOF): Active low. Signals the end of a packet. Valid only if trn_rsrc_rdy_n is also asserted.
trn_rd[63:0]	Output	Receive Data: Packet data being received. Valid only if trn_rsrc_rdy_n is also asserted.
trn_rrem_n[7:0]	Output	Receive Data Remainder: Valid only if both trn_reof_n, trn_rsrc_rdy_n, and trn_rdst_rdy_n are asserted. Legal values are: <ul style="list-style-type: none"> 0000_0000b = packet data on all of trn_rd[63:0] 0000_1111b = packet data only on trn_rd[63:32]
trn_rerrfwd_n	Output	Receive Error Forward: Active low. Marks the packet in progress as error poisoned. Asserted by the core for the entire length of the packet.
trn_rsrc_rdy_n	Output	Receive Source Ready: Active low. Indicates the core is presenting valid data on trn_rd[63:0]
trn_rdst_rdy_n	Input	Receive Destination Ready: Active low. Indicates the User Application is ready to accept data on trn_rd[63:0]. The simultaneous assertion of trn_rsrc_rdy_n and trn_rdst_rdy_n marks the successful transfer of one data beat on trn_td[63:0].
trn_rsrc_dsc_n	Output	Receive Source Discontinue: Active low. Indicates the core is aborting the current packet. Asserted when the physical link is going into reset. Not supported; signal is tied high.
trn_rnp_ok_n	Input	Receive Non-Posted OK: Active low. The User Application asserts trn_rnp_ok_n when it is ready to accept a Non-Posted Request packet. When asserted, packets are presented to the user application in the order they are received unless trn_rcpl_streaming_n is asserted. When the User Application approaches a state where it is unable to service Non-Posted Requests, it must deassert trn_rnp_ok_n one clock cycle before the core presents EOF of the next-to-last Non-Posted TLP the User Application can accept allowing Posted and Completion packets to bypass Non-Posted packets in the inbound queue.

Table 2-9: Receive Transaction Interface Signals (Continued)

Name	Direction	Description
trn_rcpl_streaming_n	Input	Receive Completion Streaming: Active low. Asserted to enable Upstream Memory Read transmission without the need for throttling. See “Performance Considerations on Receive Transaction Interface,” page 74 for details.
trn_rbar_hit_n[6:0]	Output	Receive BAR Hit: Active low. Indicates BAR(s) targeted by the current receive transaction. $\text{trn_rbar_hit_n}[0] \geq \text{BAR0}$ $\text{trn_rbar_hit_n}[1] \geq \text{BAR1}$ $\text{trn_rbar_hit_n}[2] \geq \text{BAR2}$ $\text{trn_rbar_hit_n}[3] \geq \text{BAR3}$ $\text{trn_rbar_hit_n}[4] \geq \text{BAR4}$ $\text{trn_rbar_hit_n}[5] \geq \text{BAR5}$ $\text{trn_rbar_hit_n}[6] \geq \text{Expansion ROM Address}$. Note that if two BARs are configured into a single 64-bit address, both corresponding trn_rbar_hit_n bits are asserted.
trn_rfc_ph_av[7:0] ¹	Output	Receive Posted Header Flow Control Credits Available: The number of Posted Header FC credits available to the remote link partner.
trn_rfc_pd_av[11:0] ¹	Output	Receive Posted Data Flow Control Credits Available: The number of Posted Data FC credits available to the remote link partner.
trn_rfc_nph_av[7:0] ¹	Output	Receive Non-Posted Header Flow Control Credits Available: Number of Non-Posted Header FC credits available to the remote link partner.
trn_rfc_npd_av[11:0] ¹	Output	Receive Non-Posted Data Flow Control Credits Available: Number of Non-Posted Data FC credits available to the remote link partner. Always 0 as a result of advertising infinite initial data credits.

1. Credit values given to the user are instantaneous quantities, not the cumulative (from time zero) values seen by the remote link partner.

Configuration Interface

The Configuration (CFG) interface enables the user design to inspect the state of the Endpoint for PCIe configuration space. The user provides a 10-bit configuration address, which selects one of the 1024 configuration space double word (DWORD) registers. The endpoint returns the state of the selected register over the 32-bit data output port.

Table 2-10 defines the Configuration interface signals. See “Accessing Configuration Space Registers,” page 78 for usage.

Table 2-10: Configuration Interface Signals

Name	Direction	Description
cfg_do[31:0]	Output	Configuration Data Out: A 32-bit data output port used to obtain read data from the configuration space inside the core.
cfg_rd_wr_done_n	Output	Configuration Read Write Done: Active low. The read-write done signal indicates a successful completion of the user configuration register access operation. For a user configuration register read operation, the signal validates the cfg_do[31:0] data-bus value. Currently, writes to the configuration space through the configuration port are not supported. ¹
cfg_di[31:0]	Input	Configuration Data In: 32-bit data input port used to provide write data to the configuration space inside the core. Not supported. ¹
cfg_dwaddr[9:0]	Input	Configuration DWORD Address: A 10-bit address input port used to provide a configuration register DWORD address during configuration register accesses.
cfg_wr_en_n	Input	Configuration Write Enable: Active-low write-enable for configuration register access. Not supported. ¹
cfg_rd_en_n	Input	Configuration Read Enable: Active low read-enable for configuration register access. Note: cfg_rd_en_n must be asserted for no more than 1 trn_clk cycle at a time.
cfg_interrupt_n	Input	Configuration Interrupt: Active-low interrupt-request signal. The User Application may assert this to cause appropriate interrupt messages to be transmitted by the core.
cfg_interrupt_rdy_n	Output	Configuration Interrupt Ready: Active-low interrupt grant signal. Assertion on this signal indicates that the core has successfully transmitted the appropriate interrupt message.
cfg_interrupt_mmenable[2:0]	Output	Configuration Interrupt Multiple Message Enable: This is the value of the Multiple Message Enable field. Values range from 000b to 101b. A value of 000b indicates that single vector MSI is enabled, while other values indicate the number of bits that may be used for multi-vector MSI.

Table 2-10: Configuration Interface Signals (Continued)

Name	Direction	Description										
cfg_interrupt_msienable	Output	Configuration Interrupt MSI Enabled: Indicates that the Message Signaling Interrupt (MSI) messaging is enabled. If 0, then only Legacy (INTx) interrupts may be sent.										
cfg_interrupt_di[7:0]	Input	Configuration Interrupt Data In: For Message Signaling Interrupts (MSI), the portion of the Message Data that the endpoint must drive to indicate MSI vector number, if Multi-Vector Interrupts are enabled. The value indicated by cfg_interrupt_mmenable[2:0] determines the number of lower-order bits of Message Data that the endpoint provides; the remaining upper bits of cfg_interrupt_di[7:0] are not used. For Single-Vector Interrupts, cfg_interrupt_di[7:0] is not used. For Legacy interrupt messages (Assert_INTx, Deassert_INTx), the following list defines the type of message to be sent: <table><tr><th>Value</th><th>Legacy Interrupt</th></tr><tr><td>00h</td><td>INTA</td></tr><tr><td>01h</td><td>INTB</td></tr><tr><td>02h</td><td>INTC</td></tr><tr><td>03h</td><td>INTD</td></tr></table>	Value	Legacy Interrupt	00h	INTA	01h	INTB	02h	INTC	03h	INTD
Value	Legacy Interrupt											
00h	INTA											
01h	INTB											
02h	INTC											
03h	INTD											
cfg_interrupt_do[7:0]	Output	Configuration Interrupt Data Out: The value of the lowest 8 bits of the Message Data field in the endpoint's MSI capability structure. This value is used in conjunction with cfg_interrupt_mmenable[2:0] to drive cfg_interrupt_di[7:0].										
cfg_interrupt_assert_n	Input	Configuration Legacy Interrupt Assert/Deassert Select: Selects between Assert and Deassert messages for Legacy interrupts when cfg_interrupt_n is asserted. Not used for MSI interrupts. <table><tr><th>Value</th><th>Message Type</th></tr><tr><td>0</td><td>Assert</td></tr><tr><td>1</td><td>Deassert</td></tr></table>	Value	Message Type	0	Assert	1	Deassert				
Value	Message Type											
0	Assert											
1	Deassert											
cfg_to_turnoff_n	Output	Configuration To Turnoff: Notifies the user that a PME_TURN_Off message has been received and the main power will soon be removed.										
cfg_byte_en_n[3:0]	Input	Configuration Byte Enable: Active-low byte enables for configuration register access signal. Not supported. ¹										
cfg_bus_number[7:0]	Output	Configuration Bus Number: Provides the assigned bus number for the device. The User Application must use this information in the Bus Number field of outgoing TLP requests. Default value after reset is 00h. Refreshed whenever a Type 0 Configuration packet is received.										

Table 2-10: Configuration Interface Signals (Continued)

Name	Direction	Description
cfg_device_number[4:0]	Output	Configuration Device Number: Provides the assigned device number for the device. The User Application must use this information in the Device Number field of outgoing TLP requests. Default value after reset is 0000b. Refreshed whenever a Type 0 Configuration packet is received.
cfg_function_number[2:0]	Output	Configuration Function Number: Provides the function number for the device. The User Application must use this information in the Function Number field of outgoing TLP request. Function number is hard-wired to 000b.
cfg_status[15:0]	Output	Configuration Status: Status register from the Configuration Space Header.
cfg_command[15:0]	Output	Configuration Command: Command register from the Configuration Space Header.
cfg_dstatus[15:0]	Output	Configuration Device Status: Device status register from the PCI Express Extended Capability Structure.
cfg_dcommand[15:0]	Output	Configuration Device Command: Device control register from the PCI Express Extended Capability Structure.
cfg_lstatus[15:0]	Output	Configuration Link Status: Link status register from the PCI Express Extended Capability Structure.
cfg_lcommand[15:0]	Output	Configuration Link Command: Link control register from the PCI Express Extended Capability Structure.
cfg_pm_wake_n	Input	Configuration Power Management Wake: A one-clock cycle active low assertion signals the core to generate and send a Power Management Wake Event (PM_PME) Message TLP to the upstream link partner. Note: The user is required to assert this input only under stable link conditions as reported on the cfg_pcie_link_state[2:0]. Assertion of this signal when the PCIe link is in transition results in incorrect behavior on the PCIe link.
cfg_pcie_link_state_n[2:0]	Output	PCI Express Link State: One-hot encoded bus that reports the PCIe Link State Information to the user. 110b - PCI Express Link State is "L0" 101b - PCI Express Link State is "L0s" 011b - PCI Express Link State is "L1" 111b - PCI Express Link State is "in transition"

Table 2-10: Configuration Interface Signals (Continued)

Name	Direction	Description
cfg_trn_pending_n	Input	User Transaction Pending: If asserted, sets the Transactions Pending bit in the Device Status Register. Note: The user is required to assert this input if the User Application has not received a completion to an upstream request.
cfg_dsn[63:0]	Input	Configuration Device Serial Number: Serial Number Register fields of the Device Serial Number extended capability.

1. Currently, writing to the configuration space through the user configuration port is not supported; these ports are on the Endpoint core to allow for future feature enhancement.

Error Reporting Signals

Table 2-11 defines the User Application error-reporting signals.

Table 2-11: User Application Error-Reporting Signals

Port Name	Direction	Description
cfg_err_ecrc_n	Input	ECRC Error Report: The user can assert this signal to report an ECRC error (end-to-end CRC).
cfg_err_ur_n	Input	Configuration Error Unsupported Request: The user can assert this signal to report that an unsupported request was received.
cfg_err_cpl_timeout_n	Input	Configuration Error Completion Timeout: The user can assert this signal to report a completion timed out. Note: The user should assert this signal only if the device power state is D0. Asserting this signal in non-D0 device power states might result in an incorrect operation on the PCIe link. For additional information, see the <i>PCI Express Base Specification</i> , Rev.1.1, Section 5.3.1.2.
cfg_err_cpl_unexpect_n	Input	Configuration Error Completion Unexpected: The user can assert this signal to report that an unexpected completion was received.
cfg_err_cpl_abort_n	Input	Configuration Error Completion Aborted: The user can assert this signal to report that a completion was aborted.
cfg_err_posted_n	Input	Configuration Error Posted: This signal is used to further qualify any of the cfg_err_* input signals. When this input is asserted concurrently with one of the other signals, it indicates that the transaction which caused the error was a posted transaction.
cfg_err_cor_n	Input	Configuration Error Correctable Error: The user can assert this signal to report that a correctable error was detected.

Table 2-11: User Application Error-Reporting Signals (Continued)

Port Name	Direction	Description												
cfg_err_tlp_cpl_header[47:0]	Input	<p>Configuration Error TLP Completion Header: Accepts the header information from the user when an error is signaled. This information is required so that the core can issue a correct completion, if required.</p> <p>The following information should be extracted from the received error TLP and presented in the format below:</p> <table><tr><td>[47:41]</td><td>Lower Address</td></tr><tr><td>[40:29]</td><td>Byte Count</td></tr><tr><td>[28:26]</td><td>TC</td></tr><tr><td>[25:24]</td><td>Attr</td></tr><tr><td>[23:8]</td><td>Requester ID</td></tr><tr><td>[7:0]</td><td>Tag</td></tr></table>	[47:41]	Lower Address	[40:29]	Byte Count	[28:26]	TC	[25:24]	Attr	[23:8]	Requester ID	[7:0]	Tag
[47:41]	Lower Address													
[40:29]	Byte Count													
[28:26]	TC													
[25:24]	Attr													
[23:8]	Requester ID													
[7:0]	Tag													
cfg_err_cpl_rdy_n	Output	<p>Configuration Error Completion Ready: When asserted, this signal indicates that the core can accept assertions on cfg_err_ur_n and cfg_err_cpl_abort_n for Non-Posted Transactions. Assertions on cfg_err_ur_n and cfg_err_cpl_abort_n are ignored when cfg_err_cpl_rdy_n is deasserted.</p>												
cfg_err_locked_n	Input	<p>Configuration Error Locked: This signal is used to further qualify any of the cfg_err_* input signals. When this input is asserted concurrently with one of the other signals, it indicates that the transaction that caused the error was a locked transaction.</p> <p>This signal is intended to be used in Legacy mode. If the user needs to signal an unsupported request or an aborted completion for a locked transaction, this signal can be used to return a Completion Locked with UR or CA status.</p> <p>Note: When not in Legacy mode, the core will automatically return a Completion Locked, if appropriate.</p>												

Generating and Customizing the Core

The Endpoint Block Plus for PCI Express core is fully configurable and highly customizable, yielding a resource-efficient implementation tailored to your design requirements. The Endpoint Block Plus core is customized using the CORE Generator GUI.

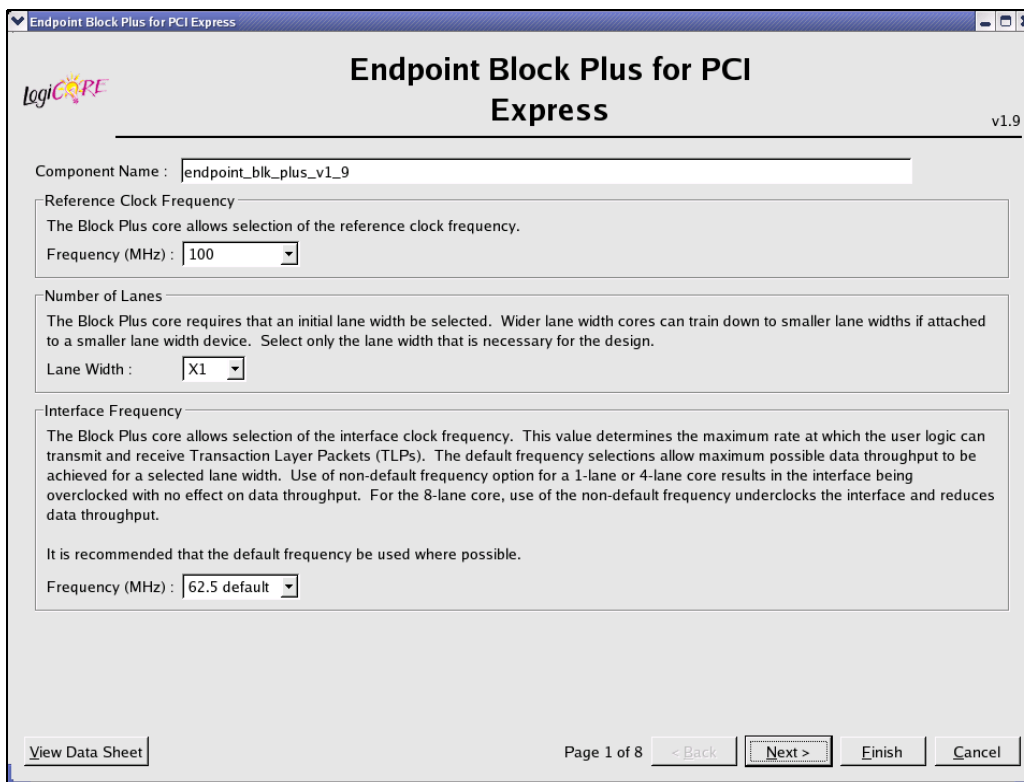
Using the CORE Generator

The Block Plus for PCIe CORE Generator GUI consists of eight screens:

- Screens 1 and 2: “Basic Parameter Settings”
- Screens 3 and 4: “Base Address Registers”
- Screens 5 and 6: “Configuration Register Settings”
- Screens 7 and 8: “Advanced Settings”

Basic Parameter Settings

The initial customization screens (Figures 3-1 and 3-2) are used to define basic parameters for the core, including component name, lane width, interface frequency, ID initial values, class code, and Cardbus CIS pointer information.



Endpoint Block Plus for PCI Express v1.9

Component Name : endpoint_blk_plus_v1_9

Reference Clock Frequency
The Block Plus core allows selection of the reference clock frequency.
Frequency (MHz) : 100

Number of Lanes
The Block Plus core requires that an initial lane width be selected. Wider lane width cores can train down to smaller lane widths if attached to a smaller lane width device. Select only the lane width that is necessary for the design.
Lane Width : X1

Interface Frequency
The Block Plus core allows selection of the interface clock frequency. This value determines the maximum rate at which the user logic can transmit and receive Transaction Layer Packets (TLPs). The default frequency selections allow maximum possible data throughput to be achieved for a selected lane width. Use of non-default frequency option for a 1-lane or 4-lane core results in the interface being overclocked with no effect on data throughput. For the 8-lane core, use of the non-default frequency underclocks the interface and reduces data throughput.
It is recommended that the default frequency be used where possible.
Frequency (MHz) : 62.5 default

View Data Sheet Page 1 of 8 < Back Next > Finish Cancel

Figure 3-1: Block Plus Parameters: Screen 1

Endpoint Block Plus for PCI Express

LogiCORE

Endpoint Block Plus for PCI Express v1.9

ID Initial Values

Vendor ID : 10EE Range: 0000..FFFF (Hex)

Device ID : 0007 Range: 0000..FFFF (Hex)

Revision ID : 00 Range: 00..FF (Hex)

Subsystem Vendor ID : 10EE Range: 0000..FFFF (Hex)

Subsystem ID : 0007 Range: 0000..FFFF (Hex)

Class Code

Base Class : 05 Range: 00..FF (Hex)

Sub-Class : 00 Range: 00..FF (Hex)

Interface : 00 Range: 00..FF (Hex)

Class Code : 050000 (Hex)

Cardbus CIS Pointer

Cardbus CIS Pointer : 00000000 (Hex)

View Data Sheet Page 2 of 8 < Back Next > Finish Cancel

Figure 3-2: Block Plus Parameters: Screen 2

Component Name

Base name of the output files generated for the core. The name must begin with a letter and can be composed of the following characters: a to z, 0 to 9, and "_."

Reference Clock Frequency

Selects the frequency of the reference clock provided on `sys_clk`. For important information about clocking the Block Plus core, see [“Clocking and Reset of the Block Plus Core,”](#) page 94 and the [Virtex-5 Integrated Endpoint Block for PCI Express Designs User Guide](#) (UG197).

Number of Lanes

The Block Plus for PCIe requires the selection of the initial lane width. Table 3-1 defines the available widths and associated generated core. Wider lane width cores are capable of training down to smaller lane widths if attached to a smaller lane-width device. See [“Link Training: 4-Lane and 8-Lane Endpoints,”](#) page 93 for more information.

Table 3-1: Lane Width

Lane Width	Product Generated
x1	1-lane Endpoint Block Plus

Table 3-1: Lane Width

Lane Width	Product Generated
x4	4-lane Endpoint Block Plus
x8	8-lane Endpoint Block Plus

Interface Frequency

The clock frequency of the core's user interface is selectable. Each lane width provides two frequency choices: a default frequency and an alternate frequency, as defined in Table 3-2. Where possible, Xilinx recommends using the default frequency. Note that using the alternate frequency in a x8 configuration results in reduced throughput. Selecting the alternate frequency for x1 and x4 does not result in higher throughput in the core, but does allow the User Application to run at a higher speed.

Table 3-2: Default and Alternate Lane Width Frequency

Lane Width	Default Frequency	Alternate Frequency
x1	62.5 MHz	125 MHz
x4	125 MHz	250 MHz
x8	250 MHz	125 MHz

ID Initial Values

- **Vendor ID.** Identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The default value, 10EEh, is the Vendor ID for Xilinx. Enter a vendor identification number here. FFFFh is reserved.
- **Device ID.** A unique identifier for the application; the default value is 0007h. This field can be any value; change this value for the application.
- **Revision ID.** Indicates the revision of the device or application; an extension of the Device ID. The default value is 00h; enter values appropriate for the application.
- **Subsystem Vendor ID.** Further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; the default value is 10EE. Typically, this value is the same as Vendor ID. Note that setting the value to 0000h can cause compliance testing issues.
- **Subsystem ID.** Further qualifies the manufacturer of the device or application. This value is typically the same as the Device ID; default value is 0007h. Note that setting the value 0000h can cause compliance testing issues.

Class Code

The Class Code identifies the general function of a device, and is divided into three byte-size fields:

- **Base Class.** Broadly identifies the type of function performed by the device.
- **Sub-Class.** More specifically identifies the device function.
- **Interface.** Defines a specific register-level programming interface, if any, allowing device-independent software to interface with the device.

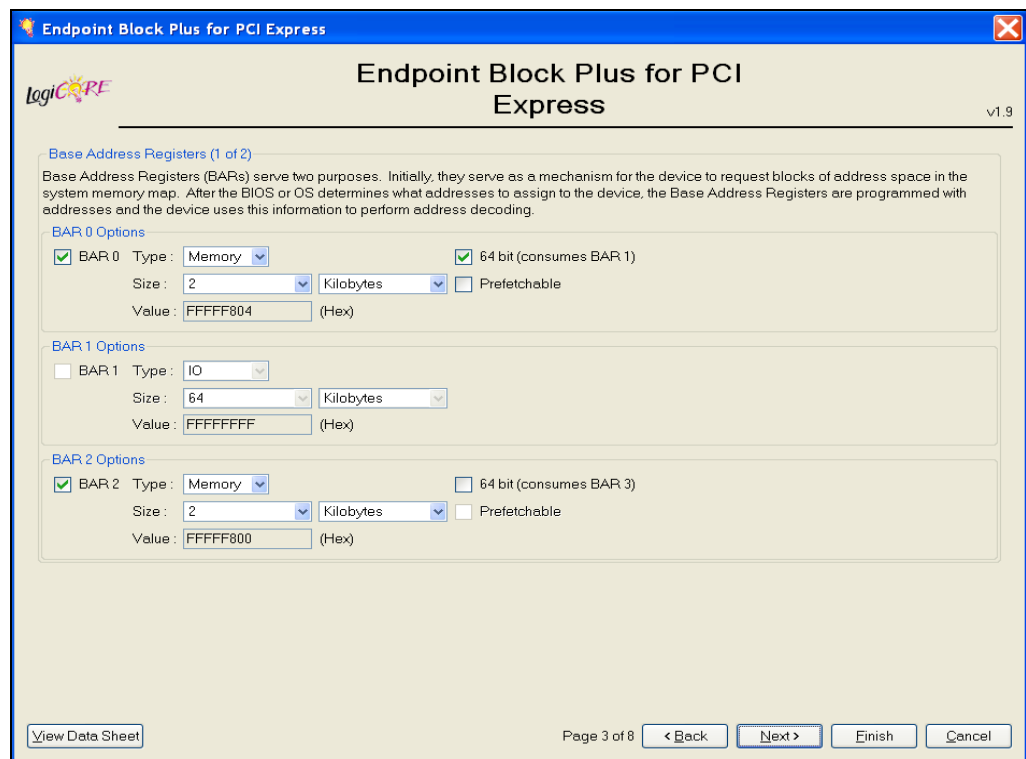
Class code encoding can be found at www.pcisig.com.

Cardbus CIS Pointer

Used in cardbus systems and points to the Card Information Structure for the cardbus card. If this field is non-zero, an appropriate Card Information Structure must exist in the correct location. The default value is 0000_0000h; value range is 0000_0000h-FFFF_FFFFh.

Base Address Registers

The Base Address Register (BAR) screens (Figures 3-3 and 3-4) let you set the base address register space. Each Bar (0 through 5) represents a 32-bit parameter.



Endpoint Block Plus for PCI Express v1.9

Base Address Registers (1 of 2)

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

BAR 0 Options

☒ BAR 0 Type: Memory ☒ 64 bit (consumes BAR 1)
 Size: 2 Kilobytes ☐ Prefetchable
 Value: FFFFF804 (Hex)

BAR 1 Options

☐ BAR 1 Type: IO
 Size: 64 Kilobytes
 Value: FFFFFFFF (Hex)

BAR 2 Options

☒ BAR 2 Type: Memory ☐ 64 bit (consumes BAR 3)
 Size: 2 Kilobytes ☐ Prefetchable
 Value: FFFFF800 (Hex)

[View Data Sheet](#) Page 3 of 8 < Back Next > Finish Cancel

Figure 3-3: BAR Options: Screen 3

Figure 3-4: BAR Options: Screen 4

Base Address Register Overview

The Block Plus core for PCIe supports up to six 32-bit Base Address Registers (BARs) or three 64-bit BARs, and the Expansion ROM BAR. BARs can be one of two sizes:

- **32-bit BARs.** The address space can be as small as 128 bytes or as large as 2 gigabytes. Used for Memory or I/O.
- **64-bit BARs.** The address space can be as small as 128 bytes or as large as 8 exabytes. Used for Memory only.

All BAR registers share the following options:

- **Checkbox.** Click the checkbox to enable the BAR; deselect the checkbox to disable the BAR.
- **Type.** BARs can either be I/O or Memory.
 - **I/O.** I/O BARs can only be 32-bit; the Prefetchable option does not apply to I/O BARs.
 - **Memory.** Memory BARs can be either 64-bit or 32-bit and can be prefetchable. When a BAR is set as 64 bits, it uses the next BAR for the extended address space and makes the next BAR inaccessible to the user.
- **Size**
 - **Memory:** When Memory and 64-bit are *not selected*, the size can range from 128 bytes to 2 gigabytes. When Memory and 64-bit are *selected*, the size can range between 128 bytes and 8 exabytes.
 - **I/O.** When selected, the size can range from 128 bytes to 2 gigabytes.

- **Prefetchable.** Identifies the ability of the memory space to be prefetched.
- **Value.** The value assigned to the BAR based on the current selections.

For more information about managing the Base Address Register settings, see [“Managing Base Address Register Settings”](#) below.

Expansion ROM Base Address Register

The 1-megabyte Expansion ROM BAR is always enabled in the Block Plus core, and cannot be disabled. User applications that do not intend to use the Expansion ROM BAR will need to implement logic to respond to accesses to the Expansion ROM BAR.

Accesses to the Expansion ROM BAR are indicated by `trn_rbar_hit[6]` assertion. Xilinx recommends that users return a Completion with Data of all zeroes if the Expansion ROM is not in use. Xilinx has found that some system BIOSs will probe the Expansion ROM BAR during boot. Not returning a completion results in a completion timeout on the requesting device, which may lead to a system hang.

Managing Base Address Register Settings

Memory, I/O, Type, and Prefetchable settings are handled by setting the appropriate GUI settings for the desired base address register.

Memory or I/O settings indicate whether the address space is defined as memory or I/O. The base address register only responds to commands that access the specified address space. Generally, memory spaces less than 4Kbytes in size should be avoided. The maximum I/O space allowed is 256 bytes. The use of I/O space should be avoided in all new designs.

Prefetchability is the ability of memory space to be prefetched. A memory space is prefetchable if there are no side-effects on reads (that is, data will not be destroyed by reading, as from a RAM). Byte write operations can be merged into a single double-word write, when applicable.

When configuring the core as a PCI Express Endpoint (non-Legacy), 64-bit addressing must be supported for all BARs (except BAR5) that have the prefetchable bit set. 32-bit addressing is permitted for all BARs that do not have the prefetchable bit set. The prefetchable bit related requirement does not apply to a Legacy Endpoint. In either of the above cases (PCI Express Endpoint or Legacy Endpoint), the minimum memory address range supported by a BAR is 128 bytes.

Disabling Unused Resources

For best results, disable unused base address registers to trim associated logic. A base address register is disabled by deselecting unused BARs in the GUI.

Configuration Register Settings

The Configuration Registers screens (Figures 3-5 and 3-6) let you set options for the Capabilities register, Device Capabilities register, and Link Capabilities register.

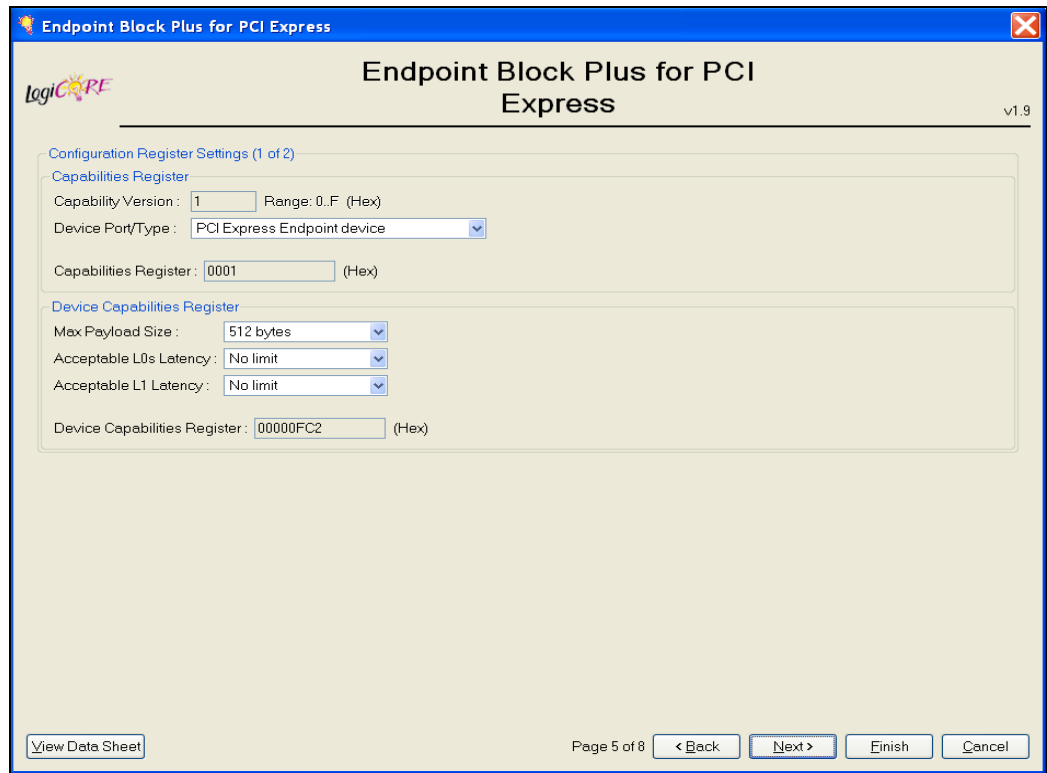


Figure 3-5: Configuration Settings: Screen 5

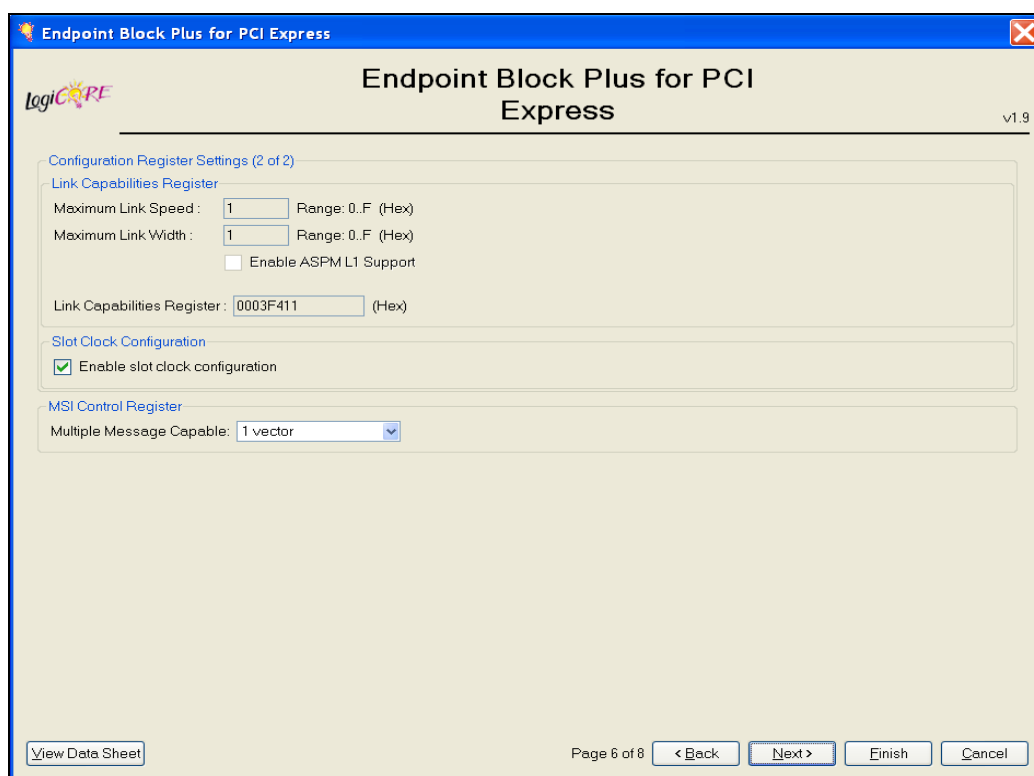


Figure 3-6: Configuration Settings: Screen 6

Capabilities Register

- **Capability Version.** Indicates PCI-SIG defined PCI Express capability structure version number; this value cannot be changed.
- **Device Port Type.** Indicates the PCI Express logical device type.
- **Capabilities Register.** Displays the value of the Capabilities register sent to the core, and is not editable.

Device Capabilities Register

- **Max Payload Size:** Indicates the maximum payload size that the device/function can support for TLPs.
- **Acceptable L0s Latency:** Indicates the acceptable total latency that an Endpoint can withstand due to the transition from L0s state to the L0 state.
- **Acceptable L1 Latency:** Indicates the acceptable latency that an Endpoint can withstand due to the transition from L1 state to the L0 state.
- **Device Capabilities Register:** Displays the value of the Device Capabilities register sent to the core, and is not editable.

Link Capabilities Register

This section is used to set the Link Capabilities register.

- **Maximum Link Speed:** Indicates the maximum link speed of the given PCI Express Link. This value is set to 2.5 Gbps and is not editable.
- **Maximum Link Width:** This value is set to the initial lane width specified in the first GUI screen and is not editable.
- **Enable ASPM L1 Support:** Indicates the level of ASPM supported on the given PCI Express Link. Only L0s entry is supported by the Block Plus core; this field is not editable.
- **Link Capabilities Register:** Displays the value of the Link Capabilities register sent to the core and is not editable.

Slot Clock Configuration

Enable Slot Clock Configuration: Indicates that the Endpoint uses the platform-provided physical reference clock available on the connector. Must be cleared if the Endpoint uses an independent reference clock.

MSI Control Register

Multiple Message Capable. Selects the number of MSI vectors to request from the Root Complex.

Advanced Settings

The Advanced Settings screens (Figure 3-7 and 3-8) include settings for the Transaction layer, Physical layer, power consumption, power management registers, power consumption, and power dissipation options.

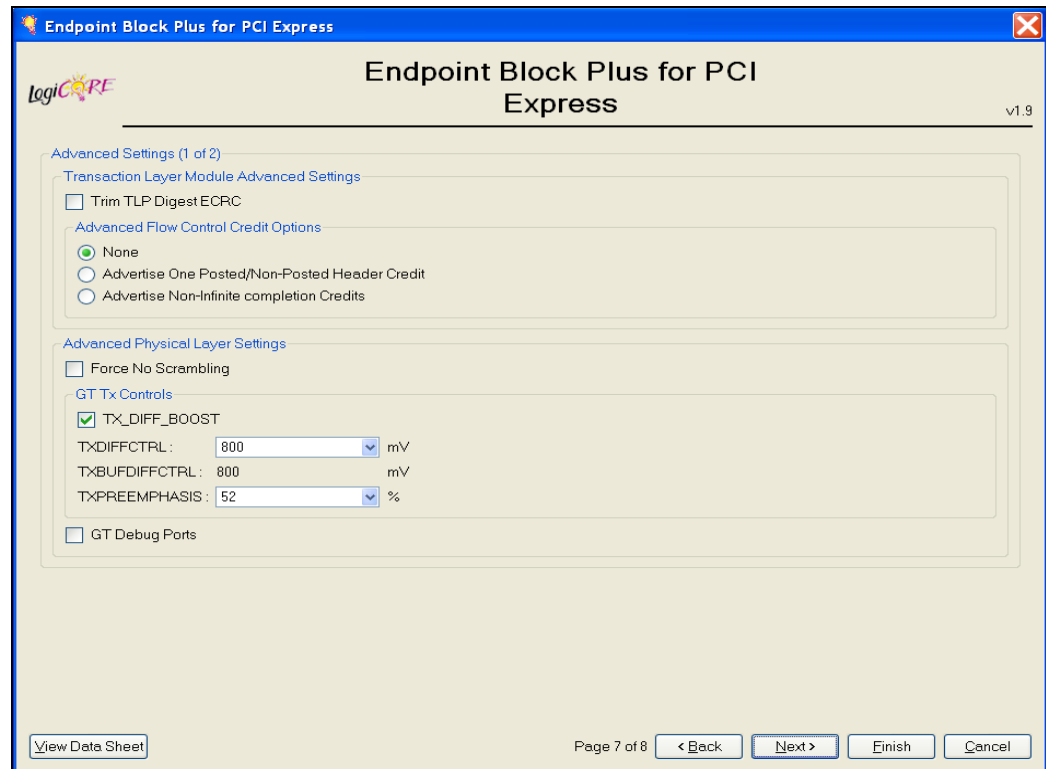


Figure 3-7: Advanced Settings: Screen 7

Endpoint Block Plus for PCI Express

LogiCORE

Endpoint Block Plus for PCI Express v1.9

Advanced Settings (2 of 2)

Power Management Registers

☐ Device Specific Initialization

☐ D1 Support ☐ D2 Support

PME Support from: ☒ D0 ☐ D1 ☐ D2 ☐ D3hot

AUX Max Current: 0mA

Power Consumption

	Power Consumed	Scale Factor	Total Power
D0:	0	x 0	= unknown (watts)
D1:	0	x 0	= unknown (watts)
D2:	0	x 0	= unknown (watts)
D3:	0	x 0	= unknown (watts)

Power Dissipation

	Power Dissipated	Scale Factor	Total Power
D0:	0	x 0	= unknown (watts)
D1:	0	x 0	= unknown (watts)
D2:	0	x 0	= unknown (watts)
D3:	0	x 0	= unknown (watts)

View Data Sheet

Page 8 of 8

< Back Next > Finish Cancel

Figure 3-8: Advanced Settings: Screen 8

Transaction Layer Module

Trim TLP Digest ECRC. Causes the core to trim any TLP digest from an inbound packet before presenting it to the user.

Advanced Flow Control Credit

- **None.** No special flow control options are selected—this is the default selection for x4 cores using a 250 MHz interface frequency or x1 cores.
- **Advertise 1 Posted / Non-Posted Header Credit.** Causes the Endpoint to advertise one posted and one non-posted header flow control credit to the upstream device. This is the default selection for x4 cores using a 125 MHz interface frequency or x8 cores using a 125 MHz or 250 MHz interface frequency. See [“Using One Posted/Non-Posted Header Credit,”](#) page 77.
- **Advertise Non-Infinite Completion Credits.** The *PCI Express Base Specification* requires that Endpoints advertise infinite completion credits. This option allows the user to disregard the specification requirement and advertise non-infinite completion credits. See [“Non-Infinite Completion Credits,”](#) page 77 for details.

Advanced Physical Layer

- **Force No Scrambling.** Used for diagnostic purposes only and should never be enabled in a working design. Setting this bit results in the data scramblers being turned off so that the serial data stream can be analyzed.

GT Tx Controls

- **TX_DIFF_BOOST.** This option changes the strength of the Tx driver and the pre-emphasis buffers. Setting this to TRUE causes the pre-emphasis percentage to be boosted or increased and the overall differential swing to be reduced.
- **TXDIFFCTRL.** This setting controls the differential output swing of the transmitter.
- **TXBUFDIFFCTRL.** This setting controls the strength of the pre-drivers and is tied to the setting of TXDIFFCTRL.
- **TXPREEMPHASIS.** This setting controls the relative strength of the main drive and the pre-emphasis
- **GT Debug Ports.** Selecting this option enables the generation of DRP ports for the GTP/X transceivers, allowing dynamic control over the GTP/X transceiver attributes. This setting may be used to perform advanced debugging, and any modifications to the transceiver default attributes must be made only if directed by Xilinx Technical Support.

After selecting this option, the core is generated with the GT DRP ports (which are generated per tile), with the exception of the DRP interface clock port. For descriptions of these ports, see the description of the corresponding DRP Port in the “Dynamic Reconfiguration Port (DRP)” section of Chapter 5: Tile Features, in the *RocketIO GTP User Guide* ([UG196](#)) and *RocketIO GTX User Guide* ([UG198](#)). [Table 3-3](#) identifies the additional DRP ports generated, and the corresponding GT DRP ports.

Note: Any modification to the transceiver attributes, by use of these ports, should be made only under the direction of Xilinx technical support.

Table 3-3: Dynamic Reconfiguration Ports

Port Name ¹	Direction	Corresponding GT DRP Port	X1		X4		X8	
gt_dclk	In	DCLK						
gt<n>_daddr[6:0]	In	DADDR[6:0]	Lane	<n>	Lanes	<n>	Lanes	<n>
gt<n>_dwen	In	DWE	0	0	0,1	0	0,1	0
gt<n>_den	In	DEN			2, 3	2	2, 3	2
gt<n>_di[15:0]	In	DI[15:0]					4, 5	4
gt<n>_drdy	Out	DRDY					6, 7	6
gt<n>_do[15:0]	Out	DO[15:0]						

1. Value of <n> in port name changes based on lane numbers; see specific lane number for value of <n>, where applicable.

Power Management Registers

- **Device Specific Initialization.** This bit indicates whether special initialization of this function is required (beyond the standard PCI configuration header) before the generic class device driver is able to use it. When selected, this option indicates that the function requires a device specific initialization sequence following transition to the D0 uninitialized state. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **D1 Support.** Option disabled; not supported by the core. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.

- **D2 Support.** Option disabled; not supported by the core. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **PME Support From:** Option disabled; not supported by the core. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **AUX Max Current.** The core always reports an auxiliary current requirement of 0 mA. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.

Power Consumption

The Block Plus core for PCIe always reports a power budget of 0W. For information about power consumption, see section 3.2.6 of the *PCI Bus Power Management Interface Specification Revision 1.2*.

Power Dissipated

The Block Plus core for PCIe always reports a power dissipation of 0W. For information about power dissipation, see section 3.2.6 of the *PCI Bus Power Management Interface Specification Revision 1.2*.

Designing with the Core

This chapter provides design instructions for the Endpoint Block Plus for PCI Express User Interface and assumes knowledge of the PCI Express Transaction Layer Packet (TLP) header fields. Header fields are defined in *PCI Express Base Specification v1.1*, Chapter 2, Transaction Layer Specification.

This chapter includes the following design guidelines:

- ◆ “Transmitting Outbound Packets”
- ◆ “Receiving Inbound Packets”
- ◆ “Accessing Configuration Space Registers”
- ◆ “Additional Packet Handling Requirements”
- ◆ “Power Management”
- ◆ “Generating Interrupt Requests”
- ◆ “Link Training: 4-Lane and 8-Lane Endpoints”
- ◆ “Clocking and Reset of the Block Plus Core”

TLP Format on the Transaction Interface

Data is transmitted and received in Big-Endian order as required by the *PCI Express Base Specification*. See Chapter 2 of the *PCI Express Base Specification* for detailed information about TLP packet ordering. Figure 4-1 represents a typical 32-bit addressable Memory Write Request TLP (as illustrated in Chapter 2 of the specification).

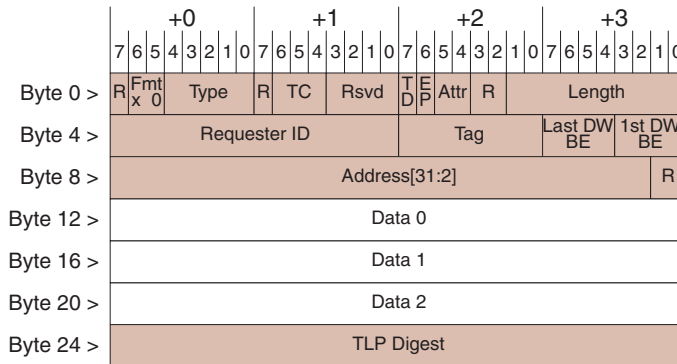


Figure 4-1: PCI Express Base Specification Byte Order

When using the Transaction interface, packets are arranged on the entire 64-bit data path. Figure 4-2 shows the same example packet on the Transaction interface. Byte 0 of the packet appears on `trn_td[63:56]` (outbound) or `trn_rd[63:56]` (inbound) of the first QWORD, byte 1 on `trn_td[55:48]` or `trn_rd[55:48]`, and so forth. Byte 8 of the packet then appears on `trn_td[63:56]` or `trn_rd[63:56]` of the second QWORD. The Header section of the packet consists of either three or four DWORDs, determined by the TLP format and type as described in section 2.2 of the *PCI Express Base Specification*.

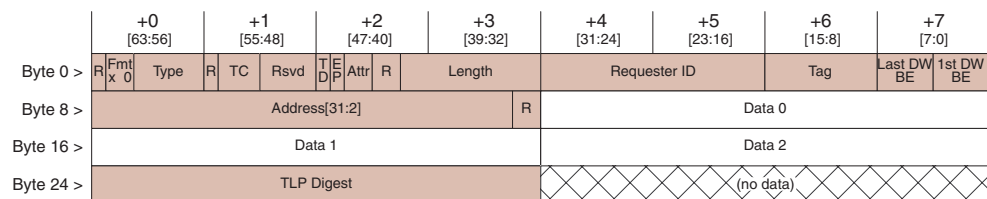


Figure 4-2: Endpoint Block Plus Byte Order

Packets sent to the core for transmission must follow the formatting rules for Transaction Layer Packets (TLPs) as specified in Chapter 2 of the *PCI Express Base Specification*. The User Application is responsible for ensuring its packets' validity, as the core does not check packet validity or validate packets. The exact fields of a given TLP vary depending on the type of packet being transmitted.

The presence of a TLP Digest or ECRC is indicated by the value of TD field in the TLP Header section. When TD=1, a correctly computed CRC32 remainder is expected to be presented as the last DWORD of the packet. The CRC32 remainder DWORD is not included in the length field of the TLP header. The User Application must calculate and present the TLP Digest as part of the packet when transmitting packets. Upon receiving packets with a TLP Digest present, the User Application must check the validity of the

CRC32 based on the contents of the packet. The core does not check the TLP Digest for incoming packets.

Transmitting Outbound Packets

Basic TLP Transmit Operation

The Block Plus core for PCIe automatically transmits the following types of packets:

- Completions to a remote device in response to Configuration Space requests.
- Error-message responses to inbound requests malformed or unrecognized by the core.

Note: Certain unrecognized requests, for example, unexpected completions, can only be detected by the User Application, which is responsible for generating the appropriate response.

The User Application is responsible for constructing the following types of outbound packets:

- Memory and I/O Requests to remote devices.
- Completions in response to requests to the User Application, for example, a Memory Read Request.

The Block Plus core for PCIe performs weighted round-robin arbitration between packets presented on the transmit interface and packets formed on the Configuration Interface (user-reported error conditions and interrupts), with a priority for Configuration Interface packets.

Table 2-8, page 29 defines the transmit User Application signals. To transmit a TLP, the User Application must perform the following sequence of events on the transmit Transaction interface:

1. The User Application logic asserts `trn_tsrc_rdy_n`, `trn_tsof_n` and presents the first TLP QWORD on `trn_td[63:0]` when it is ready to transmit data. If the core is asserting `trn_tdst_rdy_n`, the QWORD is accepted immediately; otherwise, the User Application must keep the QWORD presented until the core asserts `trn_tdst_rdy_n`.
2. The User Application asserts `trn_tsrc_rdy_n` and presents the remainder of the TLP QWORDS on `trn_td[63:0]` for subsequent clock cycles (for which the core asserts `trn_tdst_rdy_n`).
3. The User Application asserts `trn_tsrc_rdy_n` and `trn_teof_n` together with the last QWORD data. If all 8 data bytes of the last transfer are valid, they are presented on `trn_td[63:0]` and `trn_trem_n[7:0]` is driven to 00h; otherwise, the 4 remaining data bytes are presented on `trn_td[63:32]` and `trn_trem_n[7:0]` is driven to 0Fh.
4. At the next clock cycle, the User Application deasserts `trn_tsrc_rdy_n` to signal the end of valid transfers on `trn_td[63:0]`.

Figure 4-3 illustrates a 3-DW TLP header without a data payload; an example is a 32-bit addressable Memory Read request. When the User Application asserts `trn_teof_n`, it also places a value of 0Fh on `trn_trem_n[7:0]` notifying the core that only `trn_td[63:32]` contains valid data.

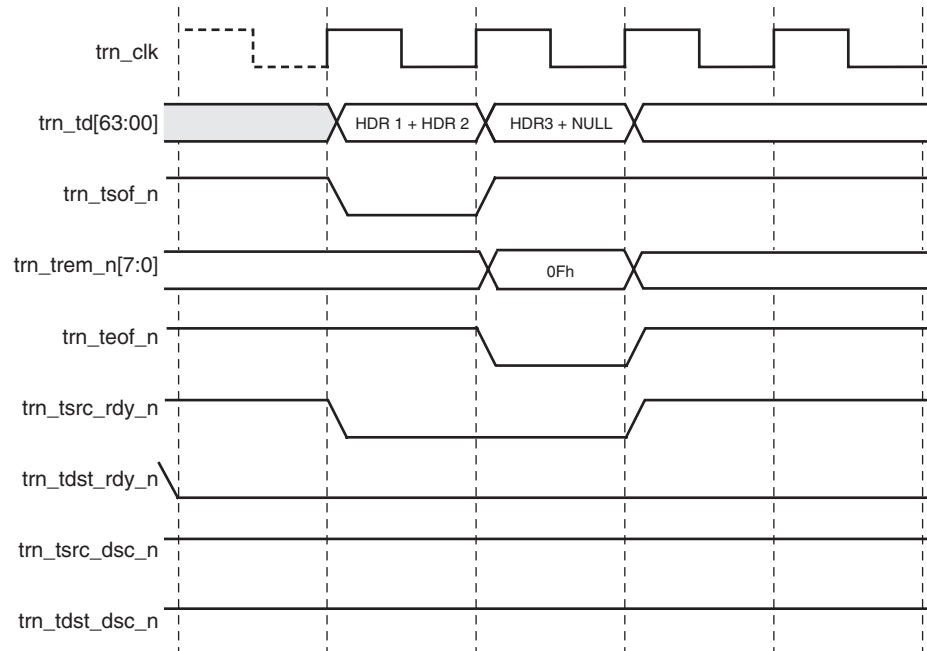


Figure 4-3: TLP 3-DW Header without Payload

Figure 4-4 illustrates a 4-DW TLP header without a data payload; an example is a 64-bit addressable Memory Read request. When the User Application asserts `trn_teof_n`, it also places a value of 00h on `trn_trem_n[7:0]` notifying the core that `trn_td[63:0]` contains valid data.

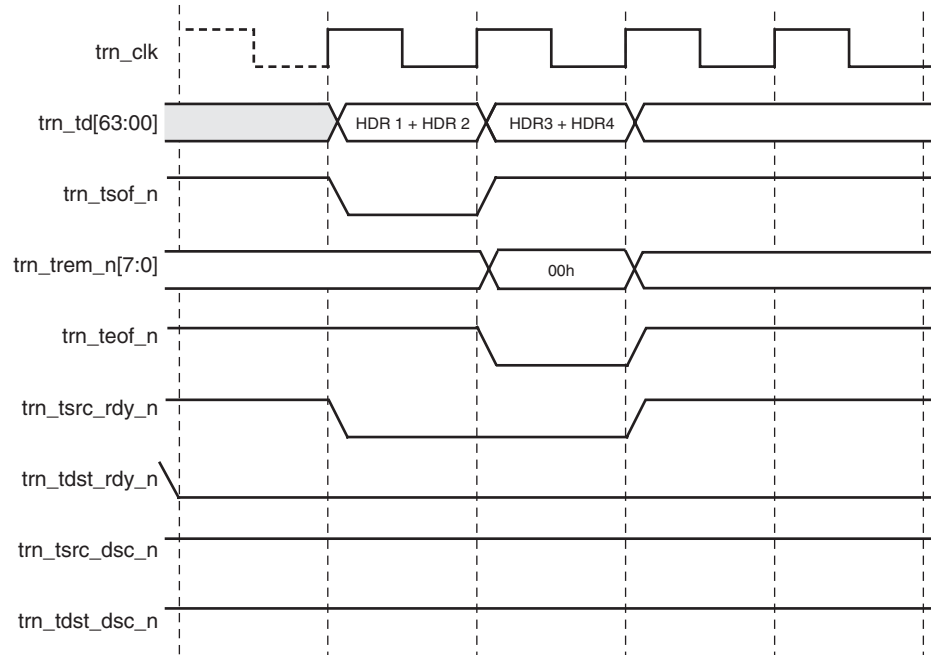


Figure 4-4: TLP with 4-DW Header without Payload

Figure 4-5 illustrates a 3-DW TLP header with a data payload; an example is a 32-bit addressable Memory Write request. When the User Application asserts `trn_teof_n`, it also puts a value of 00h on `trn_trem_n[7:0]` notifying the core that `trn_td[63:00]` contains valid data. The user must ensure the remainder field selected for the final data cycle creates a packet of length equivalent to the length field in the packet header.

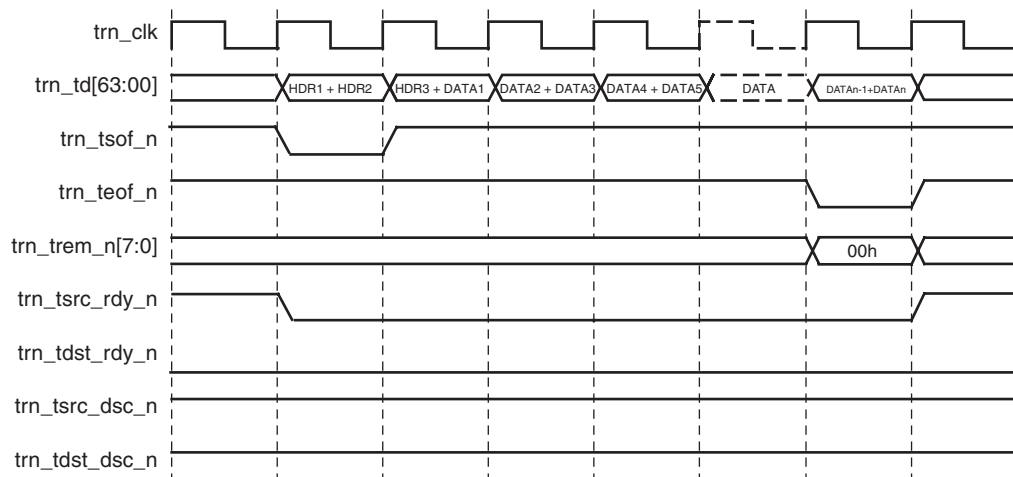


Figure 4-5: TLP with 3-DW Header with Payload

Figure 4-6 illustrates a 4-DW TLP header with a data payload; an example is a 64-bit addressable Memory Write request. When the User Application asserts `trn_teof_n`, it also places a value of 0Fh on `trn_trem_n[7:0]` notifying the core that only `trn_td[63:32]` contains valid data. The user must ensure the remainder field is selected for the final data cycle creates a packet of length equivalent to the length field in the packet header.

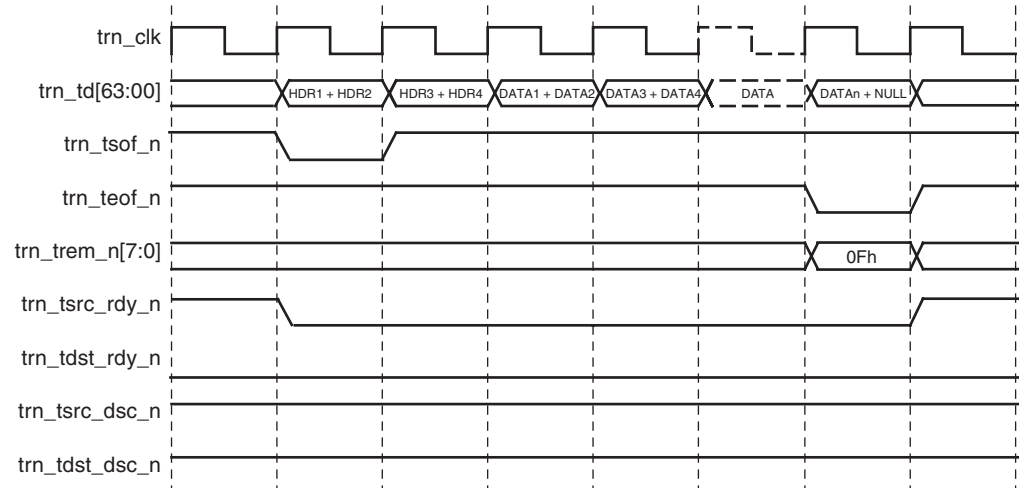


Figure 4-6: TLP with 4-DW Header with Payload

Presenting Back-to-Back Transactions on the Transmit Interface

The User Application may present back-to-back TLPs on the transmit Transaction interface to maximize bandwidth utilization. [Figure 4-7](#) illustrates back-to-back TLPs presented on the transmit interface. The User Application asserts `trn_tsof_n` and presents a new TLP on the next clock cycle after asserting `trn_teof_n` for the previous TLP.

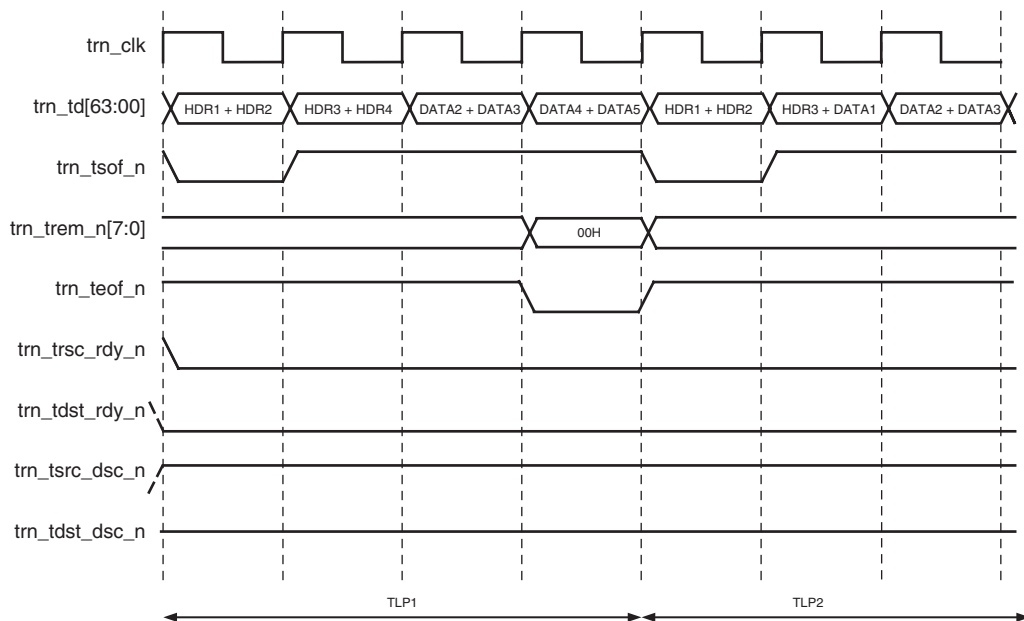


Figure 4-7: Back-to-back Transaction on Transmit Transaction Interface

Source Throttling on the Transmit Data Path

The Block Plus core for PCIe Transaction interface lets the User Application throttle back if it has no data to present on `trn_td[63:0]`. When this condition occurs, the User Application deasserts `trn_tsrc_rdy_n`, which instructs the core Transaction interface to disregard data presented on `trn_td[63:0]`. [Figure 4-8](#) illustrates the source throttling mechanism, where the User Application does not have data to present every clock cycle, and for this reason must deassert `trn_tsrc_rdy_n` during these cycles.

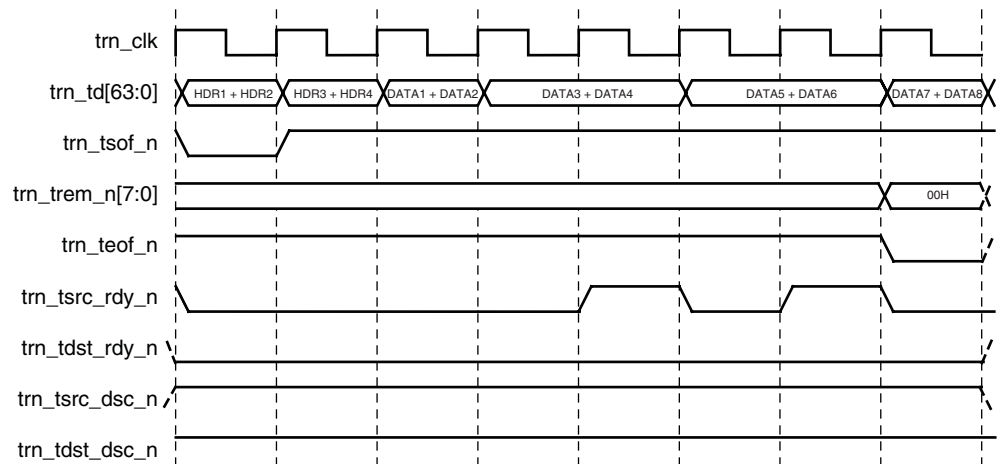


Figure 4-8: Source Throttling on the Transmit Data Path

Destination Throttling of the Transmit Data Path

The core Transaction interface throttles the transmit User Application if there is no space left for a new TLP in its transmit buffer pool. This can occur if the link partner is not processing incoming packets at a rate equal to or greater than the rate at which the User Application is presenting TLPs. Figure 4-9 illustrates the deassertion of `trn_tdst_rdy_n` to throttle the User Application when the core's internal transmit buffers are full.

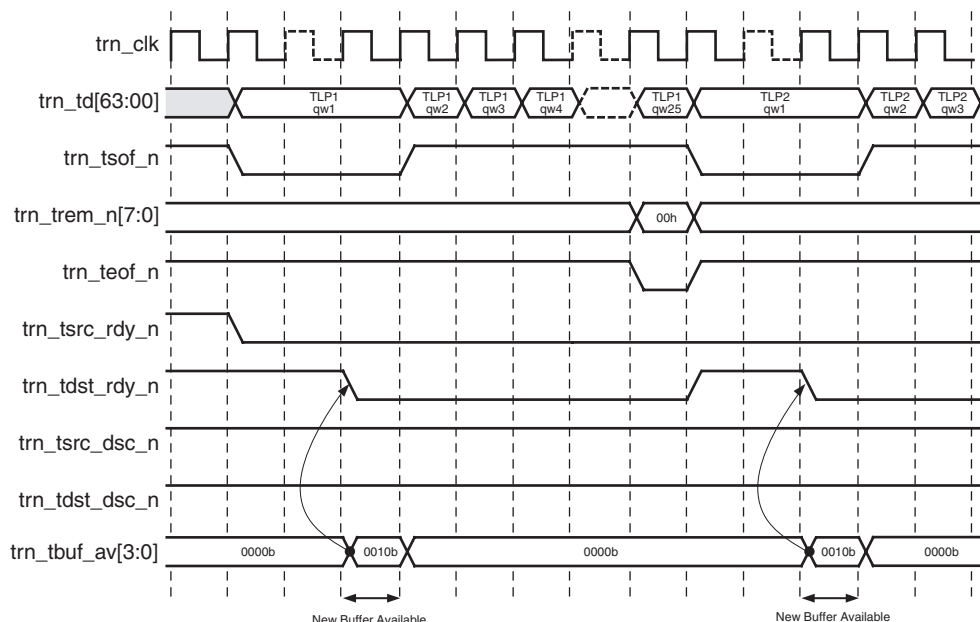


Figure 4-9: Destination Throttling of the Endpoint Transmit Transaction Interface

The core transmit Transaction interface throttles the User Application when the Power State field in Power Management Control/Status Register (Offset 0x4) of the PCI Power Management Capability Structure is changed to a non-D0 state. When this occurs, any on-going TLP is accepted completely and `trn_tdst_rdy_n` is subsequently deasserted, disallowing the User Application from initiating any new transactions—for the duration that the core is in the non-D0 power state.

Discontinuing Transmission of Transaction by Source

The core Transaction interface lets the User Application terminate transmission of a TLP by asserting `trn_tsrc_dsc_n`. Both `trn_tsrc_rdy_n` and `trn_tdst_rdy_n` must be asserted together with `trn_tsrc_dsc_n` for the TLP to be discontinued. The signal `trn_tsrc_dsc_n` must not be asserted together with `trn_tsof_n`. It can be asserted on any cycle after `trn_sof_n` deasserts up to and including the assertion of `trn_teof_n`. Asserting `trn_tsrc_dsc_n` has no effect if no TLP transaction is in progress on the transmit interface. Figure 4-10 illustrates the User Application discontinuing a packet using `trn_tsrc_dsc_n`. Asserting `trn_teof_n` together with `trn_tsrc_dsc_n` is optional.

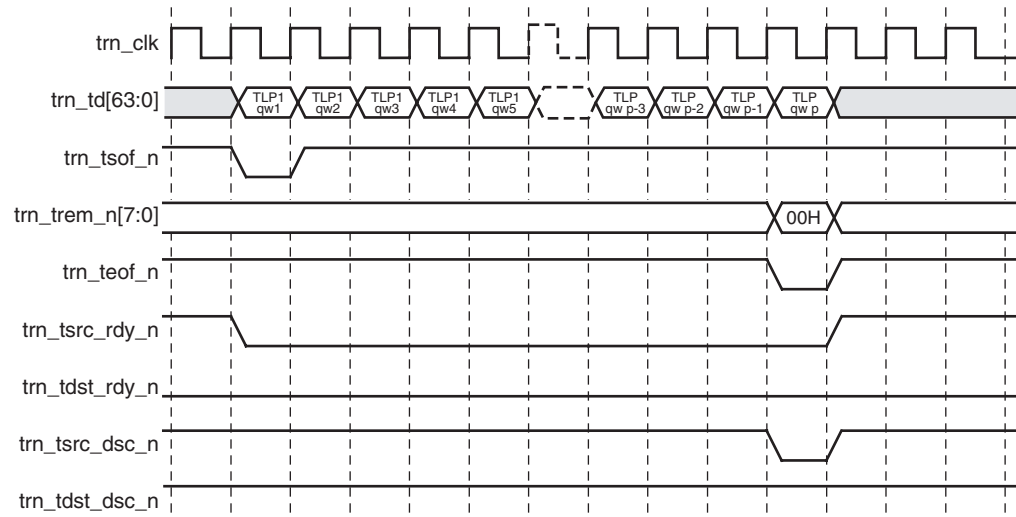


Figure 4-10: Source Driven Transaction Discontinue on Transmit Interface

Packet Data Poisoning on the Transmit Transaction Interface

The User Application uses the following mechanism to mark the data payload of a transmitted TLP as poisoned:

- Set EP=1 in the TLP header. This mechanism can be used if the payload is known to be poisoned when the first DWORD of the header is presented to the core on the TRN interface.

Appending ECRC to Protect TLPs

If the User Application needs to send a TLP Digest associated with a TLP, it must construct the TLP header such that the TD bit is set and the User Application must properly compute and append the 1-DWORD TLP Digest after the last valid TLP payload section (if applicable).

Maximum Payload Size

TLP size is restricted by the capabilities of both link partners. A maximum-sized TLP is a TLP with a 4- DWORD header, a data payload equal to the MAX_PAYLOAD_SIZE of the core (as defined in the Device Capability register) and a TLP Digest. After the link is trained, the root complex sets the MAX_PAYLOAD_SIZE value in the Device Control register. This value is equal to or less than the value advertised by the core's Device Capability register. The advertised value in the Device Capability register of the Block Plus

core is either 128, 256, or 512 bytes, depending on the setting in the CORE Generator GUI. For more information about these registers, see section 7.8 of the *PCI Express Base Specification*. The value of the core's Device Control register is provided to the user application on the `cfg_dcommand[15:0]` output. See [“Accessing Configuration Space Registers”](#) for information about this output.

Transmit Buffers

The Endpoint Block Plus core for PCIe provides buffering for up to 8 TLPs per transaction type: posted, non-posted, and completion. However, the actual number that can be buffered depends on the size of each TLP. There is space for a maximum of three 512 byte (or seven 256 byte) posted TLPs, three 512 byte (or seven 256 byte) completion TLPs, and 8 non-posted TLPs. Transmission of smaller TLPs allows for more TLPs to be buffered, but never more than 8 per type.

Outgoing TLPs are held in the core's transmit buffer until transmitted on the PCI Express interface. After a TLP is transmitted, it is stored in an internal retry buffer until the link partner acknowledges receipt of the packet.

The Endpoint Block Plus for PCIe core supports the PCI Express transaction ordering rules and promotes Posted and Completion packets ahead of blocked Non-Posted TLPs. Non-Posted TLPs can become blocked if the link partner is in a state where it momentarily has no Non-Posted receive buffers available, which it advertises through Flow Control updates. In this case, the core promotes Completion and Posted TLPs ahead of these blocked Non-Posted TLPs. However, this can only occur if the Completion or Posted TLP has been loaded into the core by the User Transmit Application. Promotion of Completion and Posted TLPs only occurs when Non-Posted TLPs are blocked; otherwise, packets are sent on the link in the order they are received from the user transmit application. See section 2.4 of the *PCI Express Base Specification* for more information about ordering rules.

The Transmit Buffers Available (`trn_tbuf_av[3:0]`) signal enables the User Application to completely utilize the PCI transaction ordering feature of the core transmitter. By monitoring the `trn_tbuf_av` bus, the User Application can ensure at least one free buffer available for any Completion or Posted TLP. [Table 4-1](#) defines the `trn_tbuf_av[3:0]` bits. A value of 1 indicates that the core can accept at least 1 TLP of that particular credit class. A value of 0 indicates that no buffers are available in that particular queue.

The `trn_tbuf_av[3]` bit indicates that the core may be about to run out of Completion Queue buffers. If this is deasserted, performance can be optimized by sending a Posted or Non-Posted TLP instead of a Completion TLP. If a Completion TLP is sent when this signal is deasserted, the core may be forced to stall the TRN interface for all TLP types until new Completion Queue buffers become available.

Table 4-1: trn_tbuf_av[3:0] Bits

Bit	Definition
<code>trn_tbuf_av[0]</code>	Non-posted buffer available
<code>trn_tbuf_av[1]</code>	Posted buffer available
<code>trn_tbuf_av[2]</code>	Completion buffer available
<code>trn_tbuf_av[3]</code>	Look-Ahead Completion buffer

Receiving Inbound Packets

Basic TLP Receive Operation

Table 2-9, page 30 defines the receive Transaction interface signals. The following sequence of events must occur on the receive Transaction interface for the Endpoint core to present a TLP to the User Application logic:

1. When the User Application is ready to receive data, it asserts `trn_rdst_rdy_n`.
2. When the core is ready to transfer data, the core asserts `trn_rsrc_rdy_n` with `trn_rsof_n` and presents the first complete TLP QWORD on `trn_rd[63:0]`.
3. The core then deasserts `trn_rsof_n`, asserts `trn_rsrc_rdy_n`, and presents TLP QWORDS on `trn_rd[63:0]` for subsequent clock cycles, for which the User Application logic asserts `trn_rdst_rdy_n`.
4. The core then asserts `trn_rsrc_rdy_n` with `trn_reof_n` and presents either the last QWORD on `trn_td[63:0]` and a value of 00h on `trn_rrem_n[7:0]` or the last DWORD on `trn_td[63:32]` and a value of 0Fh on `trn_rrem_n[7:0]`.
5. If no further TLPs are available at the next clock cycle, the core deasserts `trn_rsrc_rdy_n` to signal the end of valid transfers on `trn_rd[63:0]`.

Note: The user application should ignore any assertions of `trn_rsof_n`, `trn_reof_n`, `trn_rrem_n`, and `trn_rd` unless `trn_rsrc_rdy_n` is concurrently asserted.

Figure 4-11 shows a 3-DW TLP header without a data payload; an example is a 32-bit addressable Memory Read request. Notice that when the core asserts `trn_reof_n`, it also places a value of 0Fh on `trn_rrem_n[7:0]` notifying the user that only `trn_rd[63:32]` contains valid data.

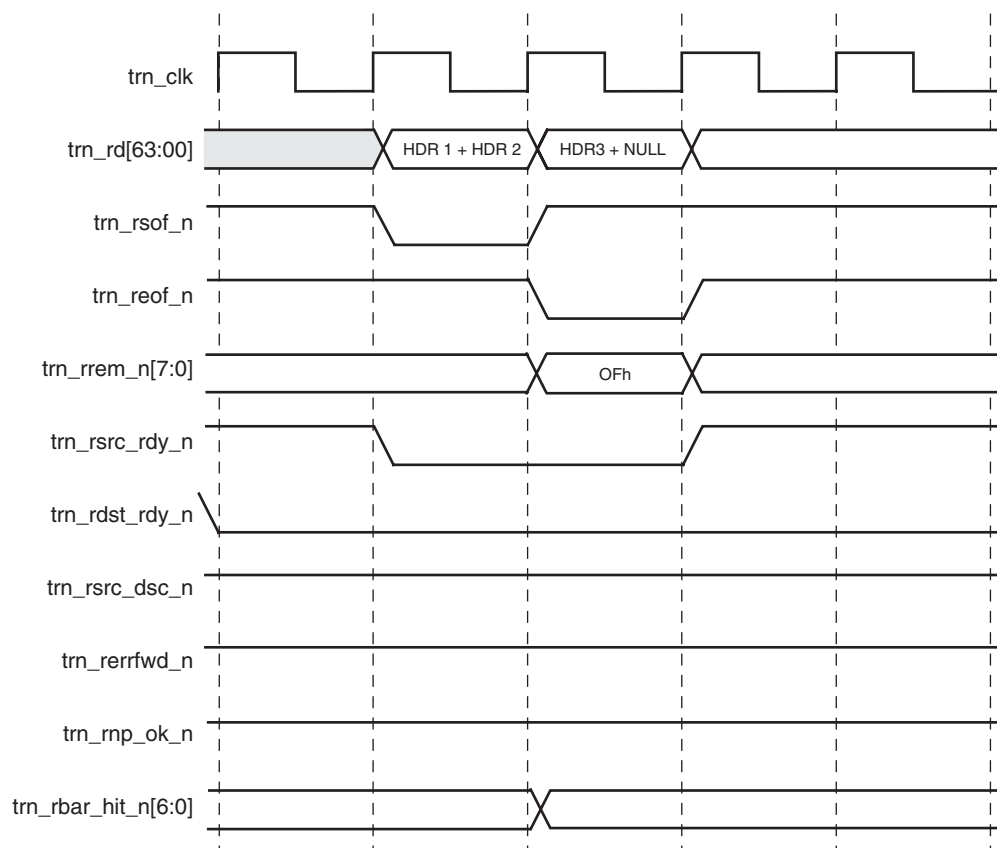


Figure 4-11: TLP 3-DW Header without Payload

Figure 4-12 shows a 4-DW TLP header without a data payload; an example is a 64-bit addressable Memory Read request. Notice that when the core asserts `trn_reof_n`, it also places a value of 00h on `trn_rrem_n[7:0]` notifying the user that `trn_rd[63:0]` contains valid data.

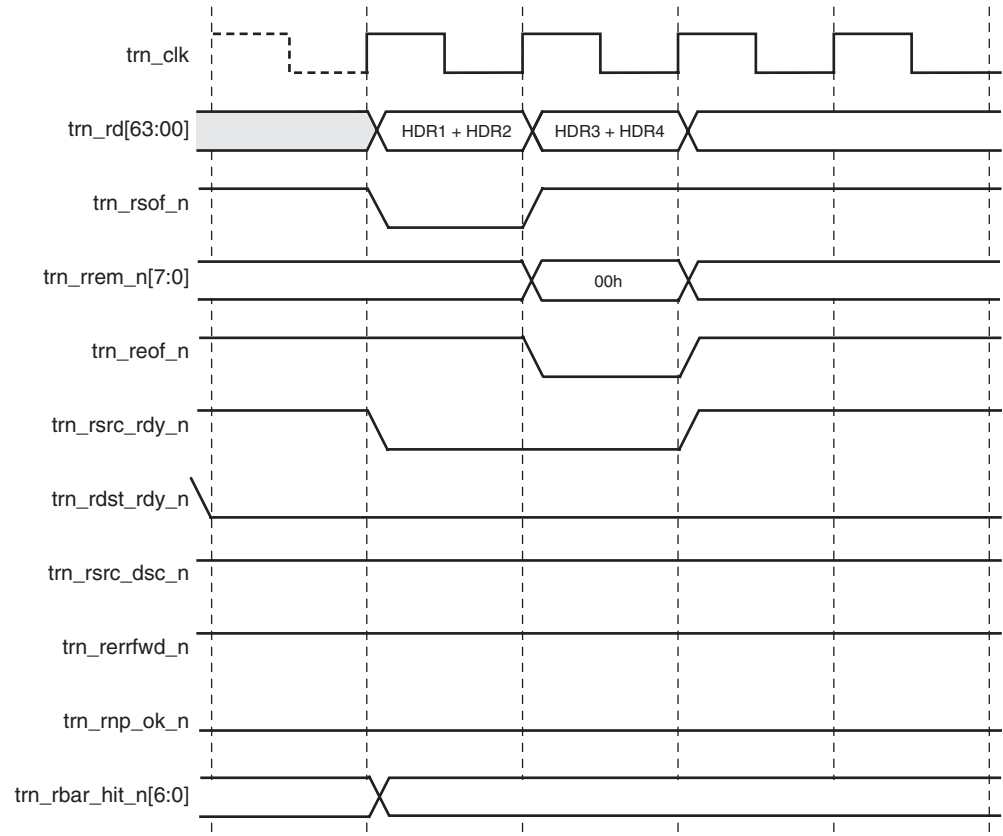


Figure 4-12: TLP 4-DW Header without Payload

Figure 4-13 shows a 3-DW TLP header with a data payload; an example is a 32-bit addressable Memory Write request. Notice that when the core asserts `trn_reof_n`, it also places a value of 00h on `trn_rrem_n[7:0]` notifying the user that `trn_rd[63:00]` contains valid data.

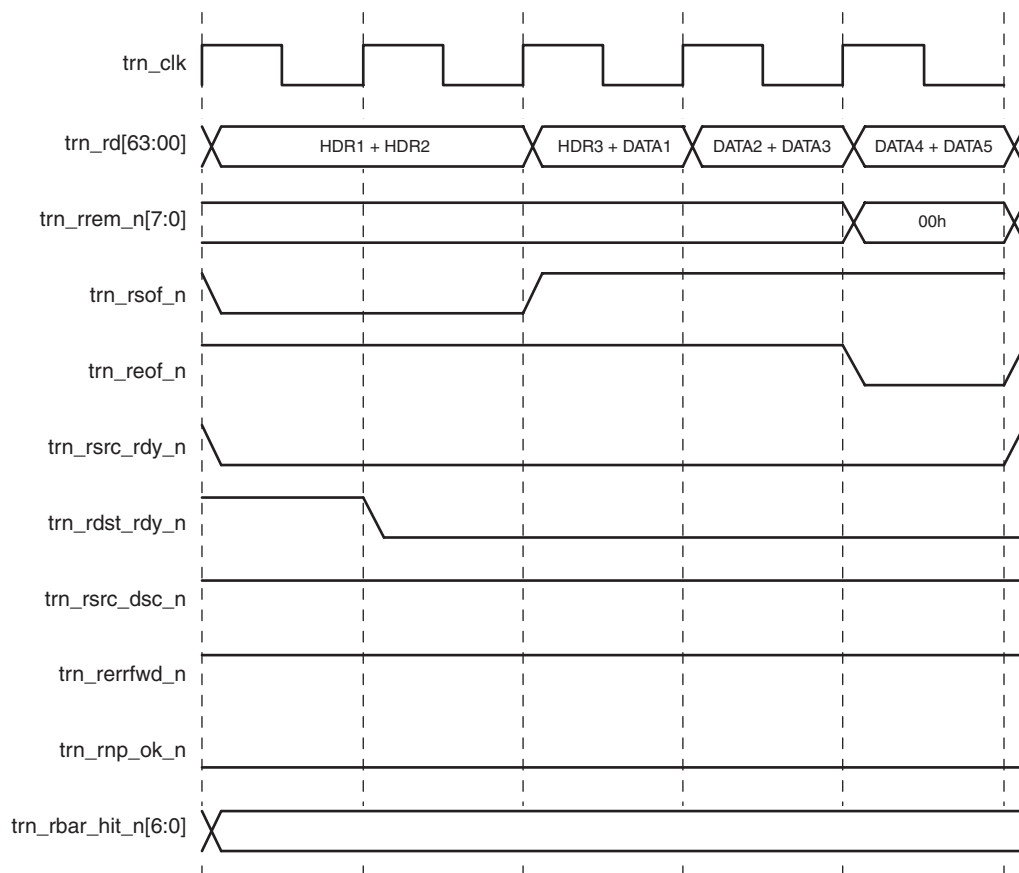


Figure 4-13: TLP 3-DW Header with Payload

Figure 4-14 shows a 4-DW TLP header with a data payload; an example is a 64-bit addressable Memory Write request. Notice that when the core asserts `trn_reof_n`, it also places a value of 0Fh on `trn_rrem_n[7:0]` notifying the user that only `trn_rd[63:32]` contains valid data.

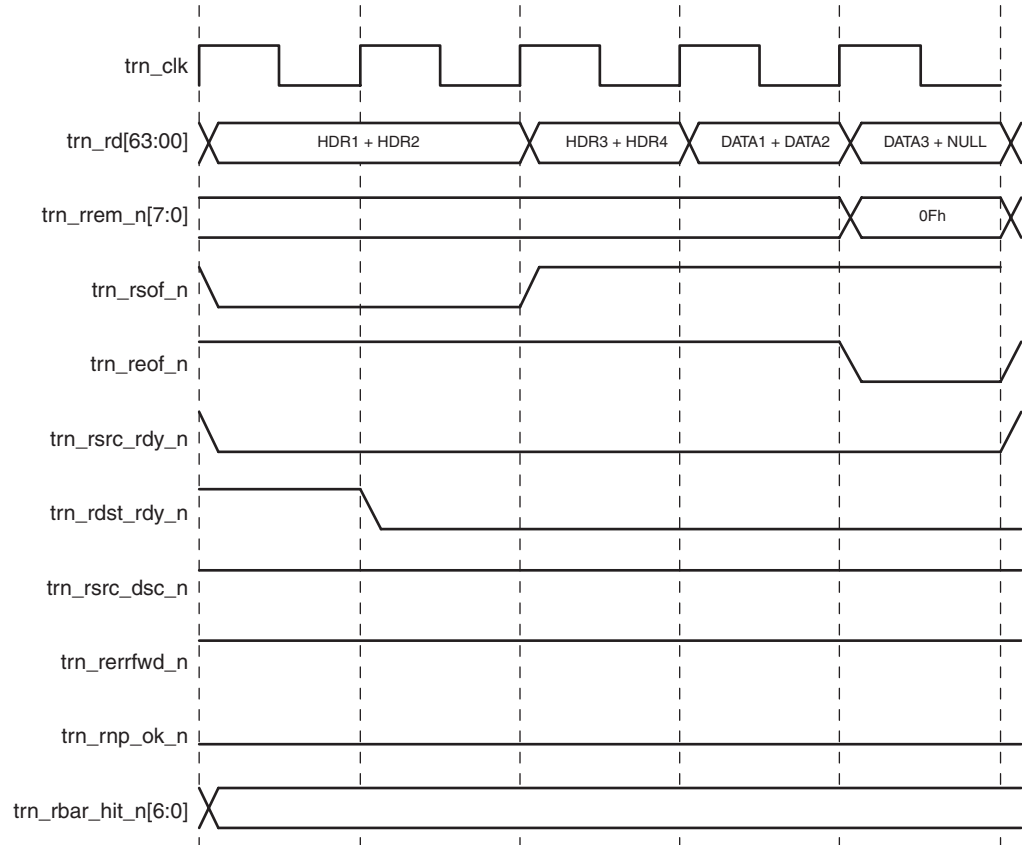


Figure 4-14: TLP 4-DW Header with Payload

Throttling the Data Path on the Receive Transaction Interface

The User Application may stall the transfer of data from the core at any time by deasserting `trn_rdst_rdy_n`. If the user deasserts `trn_rdst_rdy_n` while no transfer is in progress and if a TLP becomes available, the core asserts `trn_rsrc_rdy_n` and `trn_rsof_n` and presents the first TLP QWORD on `trn_rd[63:0]`. The core remains in this state until the user asserts `trn_rdst_rdy_n` to signal the acceptance of the data presented on `trn_rd[63:0]`. At that point, the core presents subsequent TLP QWORDS as long as `trn_rdst_rdy_n` remains asserted. If the user deasserts `trn_rdst_rdy_n` during the middle of a transfer, the core stalls the transfer of data until the user asserts `trn_rdst_rdy_n` again. There is no limit to the number of cycles the user can keep `trn_rdst_rdy_n` deasserted. The core will pause until the user is again ready to receive TLPs.

Figure 4-15 illustrates the core asserting `trn_rsrc_rdy_n` and `trn_rsof_n` along with presenting data on `trn_rd[63:0]`. The User Application logic inserts wait states by deasserting `trn_rdst_rdy_n`. The core will not present the next TLP QWORD until it detects `trn_rdst_rdy_n` assertion. The User Application logic can assert or deassert `trn_rdst_rdy_n` as required to balance receipt of new TLP transfers with the rate of TLP data processing inside the application logic.

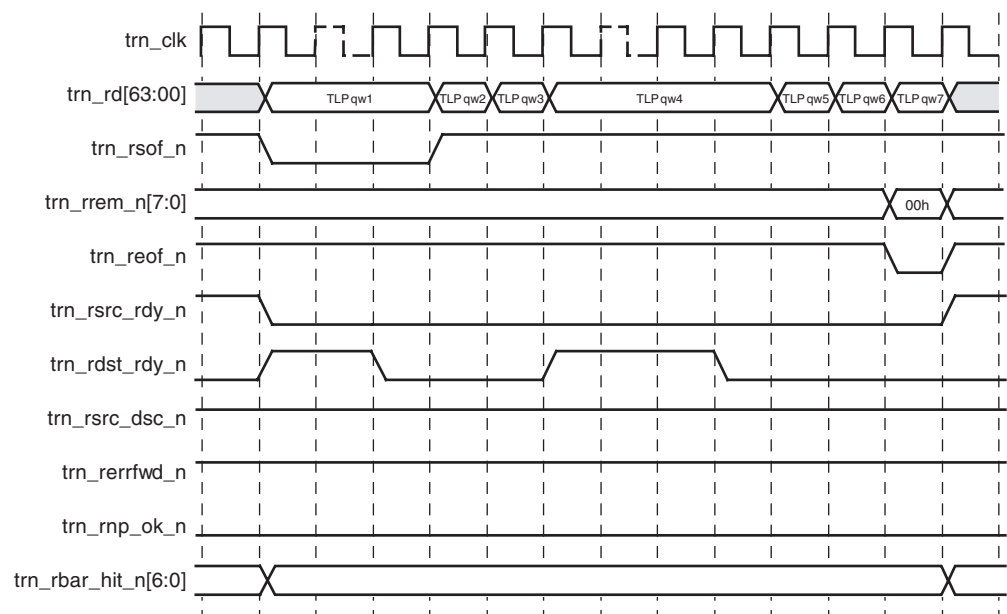


Figure 4-15: User Application Throttling Receive TLP

Receiving Back-To-Back Transactions on the Receive Transaction Interface

The User Application logic must be designed to handle presentation of back-to-back TLPs on the receive Transaction interface by the core. The core can assert `trn_rsof_n` for a new TLP at the clock cycle after `trn_reof_n` assertion for the previous TLP. [Figure 4-16](#) illustrates back-to-back TLPs presented on the receive interface.

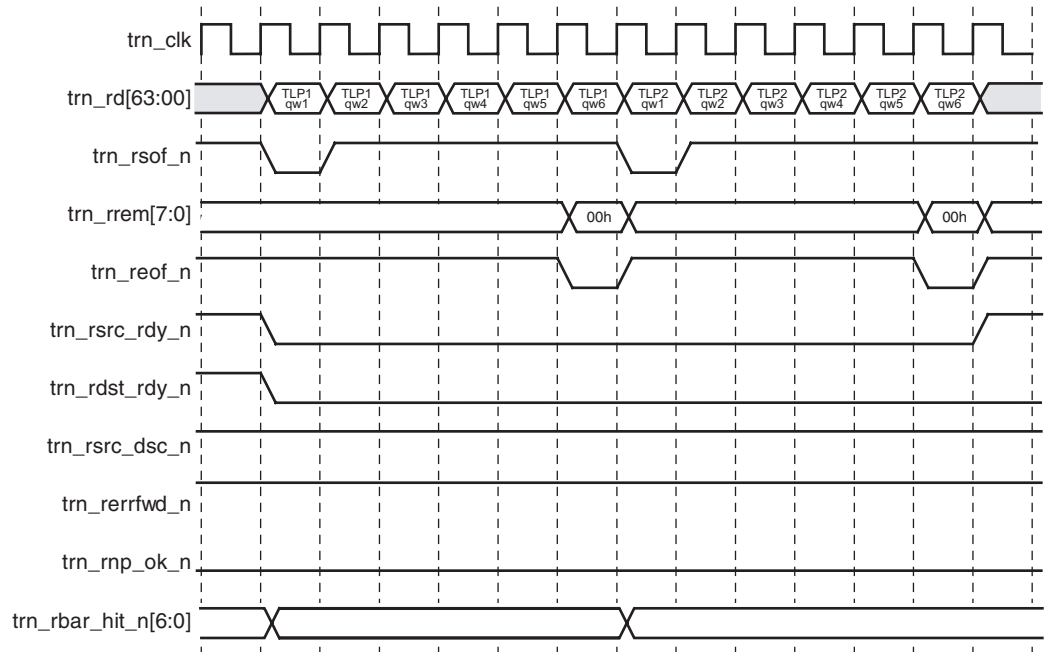


Figure 4-16: Receive Back-To-Back Transactions

If the User Application cannot accept back-to-back packets, it can stall the transfer of the TLP by deasserting `trn_rdst_rdy_n` as discussed in the previous section. [Figure 4-17](#) shows an example of using `trn_rdst_rdy_n` to pause the acceptance of the second TLP.

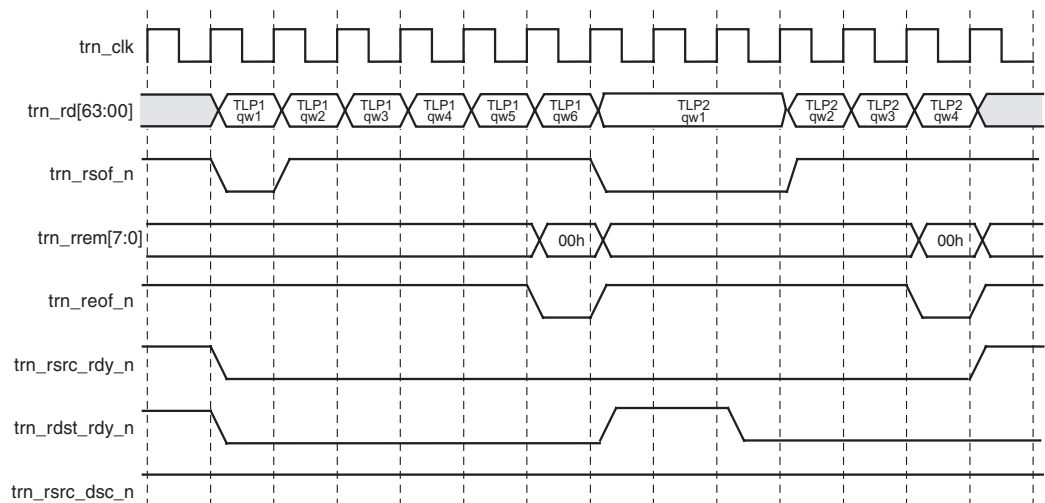


Figure 4-17: User Application Throttling Back-to-Back TLPs

Packet Re-ordering on Receive Transaction Interface

Transaction processing in the core receiver is fully compliant with the PCI transaction ordering rules when `trn_rcpl_streaming_n` is deasserted. The transaction ordering rules allow for Posted and Completion TLPs to bypass Non-Posted TLPs. See section 2.4 of the *PCI Express Base Specification* for more information about the ordering rules. See “[Performance Considerations on Receive Transaction Interface](#)” for more information on `trn_rcpl_streaming_n`.

The User Application may deassert the signal `trn_rnp_ok_n` if it is not ready to accept Non-Posted Transactions from the core, as shown in Figure 4-18 but can receive Posted and Completion Transactions. The User Application must deassert `trn_rnp_ok_n` at least one clock cycle before `trn_eof_n` of the next-to-last non-posted packet the user can accept. While `trn_rnp_ok_n` is deasserted, received Posted and Completion Transactions pass Non-Posted Transactions. After the User Application is ready to accept Non-Posted Transactions, it must reassert `trn_rnp_ok_n`. Previously bypassed Non-Posted Transactions are presented to the User Application before other received TLPs.

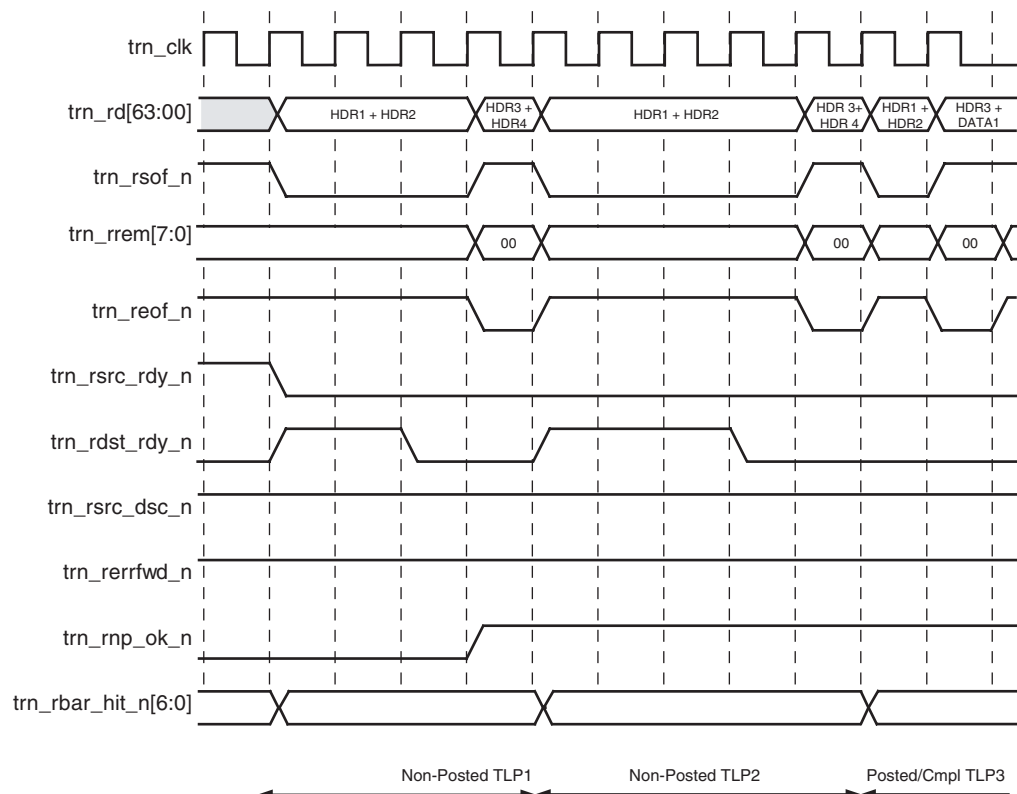


Figure 4-18: Packet Re-ordering on Receive Transaction Interface

To ensure that there is no overflow of Non-Posted storage space available in the user's logic, the following algorithm must be used:

```
For every clock cycle, do {  
  if (Valid transaction Start-Of-Frame accepted by user application) {  
    Extract TLP_type from the 1st TLP DW  
    if (TLP_type == Non-Posted) {  
      if (Non-Posted_Buffers_Available <= 2)  
        Deassert trn_rnp_ok_n on the following clock cycle.  
      else if (Other user policies stall Non-Posted transactions)  
        Deassert trn_rnp_ok_n on the following clock cycle.  
      else  
        Assert trn_rnp_ok_n on the following clock cycle.  
      Decrement Non-Posted_Buffers_Available in User Application  
    }  
  }  
}
```

Packet Data Poisoning and TLP Digest on Receive Transaction Interface

To simplify logic within the User Application, the core performs automatic pre-processing based on values of TLP Digest (TD) and Data Poisoning (EP) header bit fields on the received TLP.

All received TLPs with the Data Poisoning bit in the header set (EP=1) are presented to the user. The core asserts the `trn_rerrfwd_n` signal for the duration of each poisoned TLP, as illustrated in Figure 4-19.

If the TLP Digest bit field in the TLP header is set (TD=1), the TLP contains an End-to-End CRC (ECRC). The core performs the following operations based on how the user configured the core during core generation:

- If the Trim TLP Digest option is on, the core removes and discards the ECRC field from the received TLP and clears the TLP Digest bit in the TLP header.
- If the Trim TLP Digest option is off, the core does not remove the ECRC field from the received TLP and presents the entire TLP including TLP Digest to the User Application receiver interface.

See Chapter 3, “Generating and Customizing the Core,” for more information about how to enable the Trim TLP Digest option during core generation.

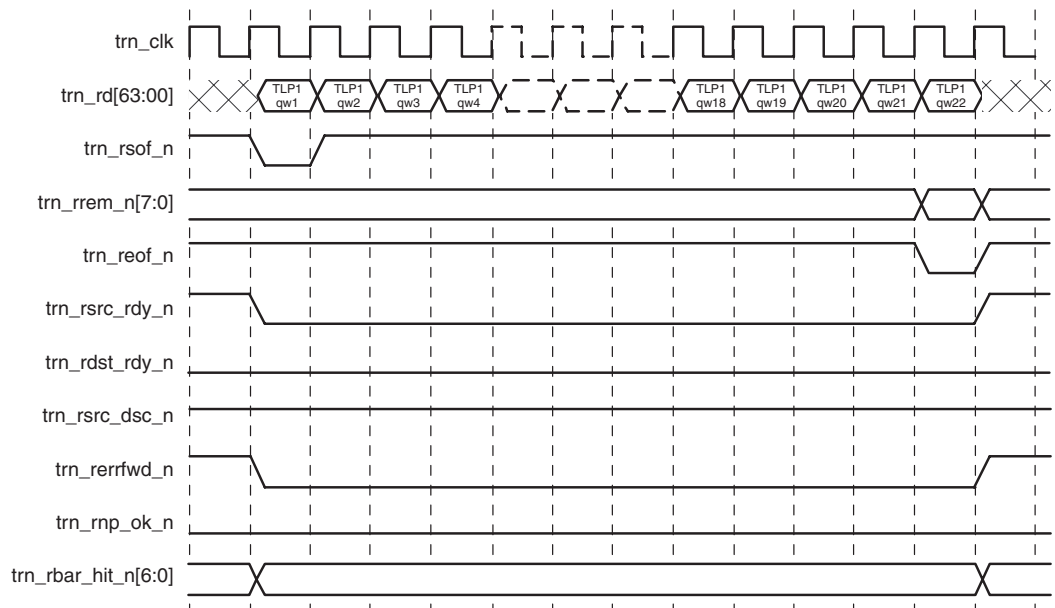


Figure 4-19: Receive Transaction Data Poisoning

Packet Base Address Register Hit on Receive Transaction Interface

The Endpoint core decodes incoming Memory and IO TLP request addresses to determine which Base Address Register (BAR) in the core's Type0 configuration space is being targeted, and indicates the decoded base address on `trn_rbar_hit_n[6:0]`. For each received Memory or IO TLP, a minimum of one and a maximum of two (adjacent) bit(s) are set to 0. If the received TLP targets a 32-bit Memory or IO BAR, only one bit is asserted. If the received TLP targets a 64-bit Memory BAR, two adjacent bits are asserted. If the core receives a TLP that is not decoded by one of the BARs (that is, a misdirected TLP), then the core will drop it without presenting it to the user and it will automatically generate an

Unsupported Request message. Note that even if the core is configured for a 64-bit BAR, the system may not always allocate a 64-bit address, in which case only one `trn_rbar_hit_n[6:0]` signal will be asserted.

Table 4-2 illustrates mapping between `trn_rbar_hit_n[6:0]` and the BARs, and the corresponding byte offsets in the core Type0 configuration header.

Table 4-2: `trn_rbar_hit_n` to Base Address Register Mapping

<code>trn_rbar_hit_n[x]</code>	BAR	Byte Offset
0	0	10h
1	1	14h
2	2	18h
3	3	1Ch
4	4	20h
5	5	24h
6	Expansion ROM BAR ¹	30h

1. Expansion ROM BAR is disabled for PCI Express Block Plus and is tied high.

For a Memory or IO TLP Transaction on the receive interface, `trn_rbar_hit_n[6:0]` is valid for the entire TLP, starting with the assertion of `trn_rsof_n`, as shown in Figure 4-20. When receiving non-Memory and non-IO transactions, `trn_rbar_hit_n[6:0]` is undefined.

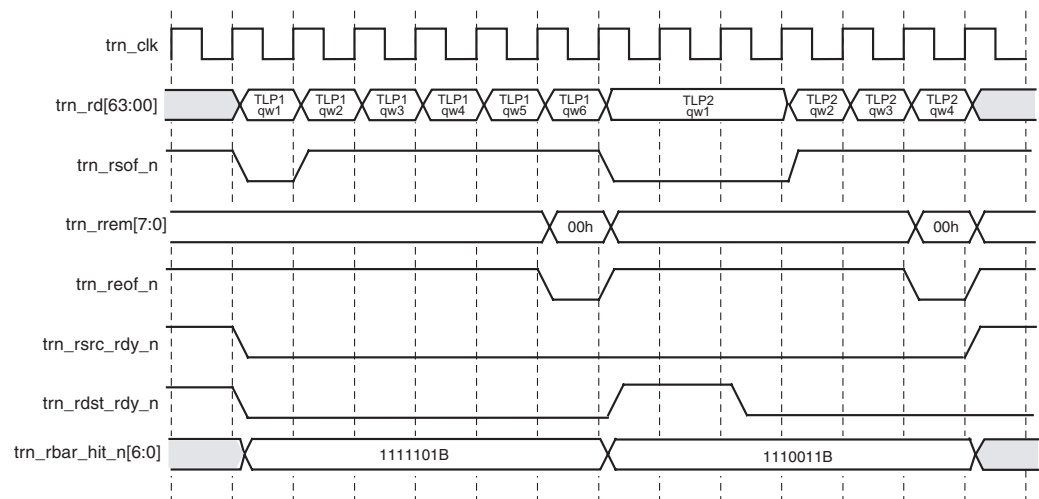


Figure 4-20: BAR Target Determination using `trn_rbar_hit`

The signal `trn_rbar_hit_n[6:0]` enables received Memory/IO Transactions to be directed to the appropriate destination Memory/IO apertures in the User Application. By utilizing `trn_rbar_hit_n[6:0]`, application logic can inspect only the lower order Memory/IO address bits within the address aperture to simplify decoding logic.

Performance Considerations on Receive Transaction Interface

The *PCI Express Base Specification v1.1* requires that all Endpoints advertise infinite Completion credits. To comply with this requirement, receive buffers must be allocated for Completions before Read Requests are initiated by the Endpoint. The Block Plus core receiver can store up to 8 Completion or Completion with Data TLPs to satisfy this requirement. If the user application generates requests that result in more than 8 outstanding Completion TLPs at any given time, caution must be taken to not overflow the core's completion buffer.

A variety of methods are available to avoid cases where the completion buffer may overflow. Applications targeting a x4 core using a 125 MHz interface frequency or x8 core using a 250 MHz interface frequency need to apply one or more of these methods.

- Completion Streaming
- Metering Reads
- Non-Infinite Completion Credits.

A x4 core using a 250 MHz interface frequency or a x1 core need not worry about applying these methods; under normal circumstances the user application should not cause the completion buffer to overflow. However, for information about designs targeting these cores, see [“x4 Core Using 250 MHz Interface and x1 Core Designs,” on page 78](#).

Special consideration must be given to designs targeting a x8 core using a 125 MHz interface frequency. The Completion Streaming method does not help in this configuration because the interface frequency is running too slowly to keep up with the incoming packets. When operating in this mode, the user must either meter the reads and keep less than 8 completion TLPs outstanding at any time, or use Non-Infinite Completion Credits. [Table 4-3](#) provides a summary of operation modes and methods available to manage completion buffer space.

Table 4-3: Completion Buffer Space Management Solutions

Core	Interface Frequency	Potential for Overflow	Available Solutions
x8	250 MHz	Yes	Completion Streaming Metering Reads Advertising Non-Infinite Completion Credits
	125 MHz	Yes	Metering Reads Advertising Non-Infinite Completion Credits
x4	250 MHz	No ¹	NA
	125 MHz	Yes	Completion Streaming Metering Reads Advertising Non-Infinite Completion Credits
x1	125 MHz	No ¹	NA
	62.5 MHz	No ¹	NA

1. See [“x4 Core Using 250 MHz Interface and x1 Core Designs,” on page 78](#) for more information.

Figure 4-21 illustrates the available options for managing completion space depending on the core being implemented.

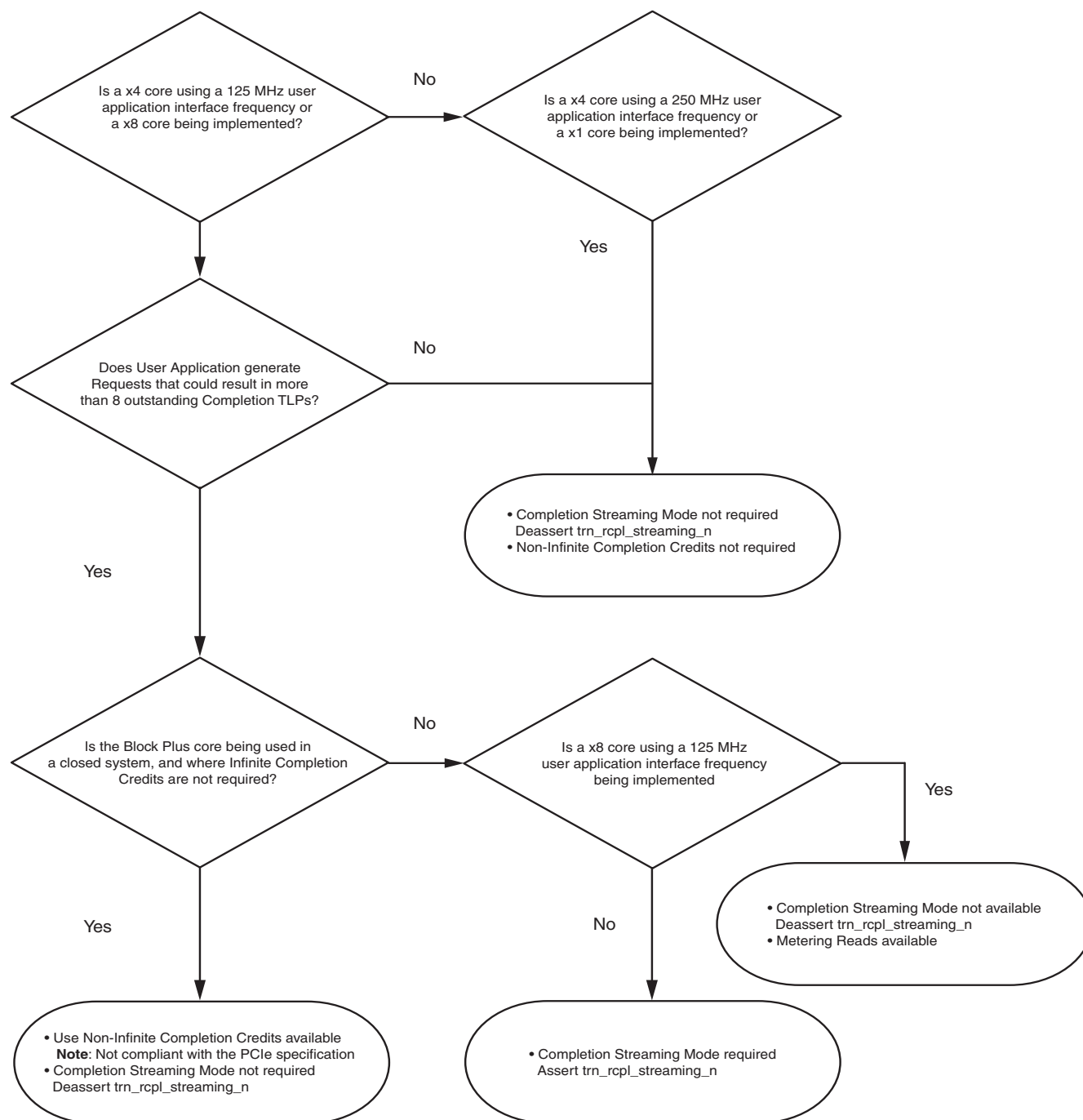


Figure 4-21: Options for Allocating Completion Buffer Space

Completion Streaming on Receive Transaction Interface

The PCI Express system responds to each Memory Read request initiated by the Endpoint with one or more Completion or Completion with Data TLPs. The exact number of Completions varies based on the RCB (Read Completion Boundary) and the starting address and size of the request. See Chapter 2 of the *PCI Express Base Specification v1.1* to determine the worst-case number of Completions that could be generated for a given request.

Pre-allocation of Completion buffers cannot provide sufficient upstream read bandwidth for all applications. For these cases the Block Plus core provides a second option, called Completion Streaming. Completion Streaming is enabled by asserting `trn_rcpl_streaming_n`, and allows a higher rate of Read Requests when certain system conditions can be met:

- Ability of User Application to receive data at line rate
- Control over relative payload sizes of downstream Posted and Completion TLPs

The Block Plus core allows the User Application to receive data in different ways depending on the requirements of the application. In most cases the different options provide a trade-off between raw performance and system restrictions. The designer must determine which option best suits the particular system requirements while ensuring the core receiver does not overflow. Options are:

- **Metered Reads:** The User Application can limit the number of outstanding reads such that there can never be more than 8 Completions at once in the receive buffer. This method precludes using the entire receive bandwidth for Completions, but does not place further restrictions on system design. In this case, the `trn_rcpl_streaming_n` signal should be deasserted.
- **Completion Streaming:** Completion Streaming increases the receive throughput with the trade-offs mentioned previously. The User Application can choose to continuously drain all available Completions from the receive buffer, before processing other TLP types, by asserting the `trn_rcpl_streaming_n` input to the core. When using Completion Streaming, the User Application must set the Relaxed Ordering bit for all Memory Read TLPs, and must continuously assert the `trn_rdst_rdy_n` signal when there are outstanding Completions. If precautions are not taken, it is possible to overflow the receive buffer if several small Completions arrive while a large received TLP (Posted or Completion) is being internally processed by the core and delaying the processing of these small Completions. There are two mechanisms to avoid this possibility; both assume the designer has control over the whole system:
 - ♦ **Control Payload Size:** The size of the payloads of incoming Posted TLPs must not be more than 4 times the size of the payloads of incoming Completion TLPs.
 - ♦ **Traffic Separation:** Completion traffic with small payloads (such as register reads) must be separated over time from Posted and Completion traffic with large payloads (more than 4 times the size of the small-payload Completions), such that one traffic type does not proceed until the other has finished.
- **Metered/Streaming Hybrid:** The `trn_rcpl_streaming_n` signal can be asserted only when bursts of Completions are expected. When bursts are not expected, `trn_rcpl_streaming_n` is deasserted so that Relaxed Ordering is not required and `trn_rdst_rdy_n` can be deasserted. Switching between Completion Streaming and regular operation can only occur when there are no outstanding Completions to be received.

Example Transactions In Violation of System Requirements

Below are examples of specific situations which violate the Block Plus core system requirements. These examples can lead to receive buffer overflows and are given as illustrations of system designs to be avoided.

- **Back-to-back completions returned without streaming:** Consider an example system wherein the Endpoint User Application transmits a Memory Read Request TLP with a Length of 512 DW (2048 Bytes). The system has a RCB of 64 Bytes, and the requested address is aligned on a 64-Byte boundary. The request can be fulfilled with up to 32 Completion with Data TLPs. If the User Application does not assert `trn_rcpl_streaming_n` and the Completions are transmitted back-to-back, this can overflow the core's receive buffers.
- **Payload size not controlled:** Consider the same system. This time, the User Application asserts `trn_rcpl_streaming_n`, but the ratio of Posted to Completion payload size is not controlled. If 9 or more 64 Byte Completions are received immediately after a 512 DW (Posted) Memory Write TLP, then the core's receive buffers can be overflowed because the Completions are less than 1/4th the size of the Memory Write TLP.

Using One Posted/Non-Posted Header Credit

The Block Plus core implements all necessary precautions for dealing with completion streaming and relaxed ordering as provided [AR25473](#). The One Header Credit configuration option allows the core to leverage PCI Express flow control to limit transmission of posted and non-posted packets by the link partner, guaranteeing this issue will not be exposed. When this option is selected, the user should not deassert `trn_rnp_ok_n`, because Non-Posted packets must be drained immediately on receipt to guarantee compliant operation. This option must be used in conjunction with Completion Streaming mode, thus `trn_rcpl_streaming_n` should be asserted.

For x4 cores running at 125MHz and x8 cores, this option is selected by default to guarantee compliant operation in Completion Streaming mode. If the user requires a higher bandwidth for Posted and Non-Posted traffic than allowed by 1-header credit for Posted and Non-Posted TLPs, this option may be turned off, provided that the user can guarantee that the core will not receive a Posted or Non-Posted TLP immediately followed by a continuous burst of more than 64 Completions. If this situation *could* occur, the user may still turn off this option, but hardware testing is required to ensure proper operation.

Non-Infinite Completion Credits

If the Block Plus core is to be used in a closed system in which infinite Completion credits are not a requirement, the core may be configured to advertise finite completion credits. When configured this way, the core advertises 8 CplH and 128 (decimal) CplD credits. Enabling non-infinite Completion credits allows the core to leverage PCI Express flow control to prevent receive buffer overflows. If the core is used with this configuration, Completion Streaming is not used.

NOTE: Enabling non-infinite Completion credits violates the requirements of the *PCI Express Base Specification* and should only be used when necessary and in a closed system.

x4 Core Using 250 MHz Interface and x1 Core Designs

Under normal circumstances, the completion buffer should not overflow as long as the user does not throttle back when receiving data by deasserting `trn_tdst_rdy_n`. However, if the user application needs to throttle the datapath, then the application must meter the read requests to ensure that no more than 8 Completion or Completion with Data TLPs are outstanding at any given time.

Accessing Configuration Space Registers

Registers Mapped Directly onto the Configuration Interface

The Block Plus core provides direct access to select command and status registers in its Configuration Space. Values in these registers are modified by Configuration Writes received from the Root Complex and cannot be modified by the User Application. [Table 4-4](#) defines the command and status registers mapped to the configuration port.

Table 4-4: Command and Status Registers Mapped to the Configuration Port

Port Name	Direction	Description
<code>cfg_bus_number[7:0]</code>	Output	Bus Number: Default value after reset is 00h. Refreshed whenever a Type 0 Configuration packet is received.
<code>cfg_device_number[4:0]</code>	Output	Device Number: Default value after reset is 00000b. Refreshed whenever a Type 0 Configuration packet is received.
<code>cfg_function_number[2:0]</code>	Output	Function Number: Function number of the core, hard wired to 000b.
<code>cfg_status[15:0]</code>	Output	Status Register: Status register from the Configuration Space Header.
<code>cfg_command[15:0]</code>	Output	Command Register: Command register from the Configuration Space Header.
<code>cfg_dstatus[15:0]</code>	Output	Device Status Register: Device status register from the PCI Express Extended Capability Structure.
<code>cfg_dcommand[15:0]</code>	Output	Device Command Register: Device control register from the PCI Express Extended Capability Structure.
<code>cfg_lstatus[15:0]</code>	Output	Link Status Register: Link status register from the PCI Express Extended Capability Structure.
<code>cfg_lcommand[15:0]</code>	Output	Link Command Register: Link control register from the PCI Express Extended Capability Structure.

Device Control and Status Register Definitions

cfg_bus_number[7:0], cfg_device_number[4:0], cfg_function_number[2:0]

Together, these three values comprise the core ID, which the core captures from the corresponding fields of inbound Type 0 Configuration accesses. The User Application is responsible for using this core ID as the Requestor ID on any requests it originates, and using it as the Completer ID on any Completion response it sends. Note that this core supports only one function; for this reason, the function number is hard wired to 000b.

cfg_status[15:0]

This bus allows the User Application to read the Status register in the PCI Configuration Space Header. Table 4-5 defines these bits. See the *PCI Express Base Specification* for detailed information.

Table 4-5: Bit Mapping on Header Status Register

Bit	Name
cfg_status[15]	Detected Parity Error
cfg_status[14]	Signaled System Error
cfg_status[13]	Received Master Abort
cfg_status[12]	Received Target Abort
cfg_status[11]	Signaled Target Abort
cfg_status[10:9]	DEVSEL Timing (hard-wired to 00b)
cfg_status[8]	Master Data Parity Error
cfg_status[7]	Fast Back-to-Back Transactions Capable (hard-wired to 0)
cfg_status[6]	Reserved
cfg_status[5]	66 MHz Capable (hard-wired to 0)
cfg_status[4]	Capabilities List Present (hard-wired to 1)
cfg_status[3]	Interrupt Status
cfg_status[2:0]	Reserved

cfg_command[15:0]

This bus reflects the value stored in the Command register in the PCI Configuration Space Header. Table 4-6 provides the definitions for each bit in this bus. See the *PCI Express Base Specification* for detailed information.

Table 4-6: Bit Mapping on Header Command Register

Bit	Name
cfg_command[15:11]	Reserved
cfg_command[10]	Interrupt Disable
cfg_command[9]	Fast Back-to-Back Transactions Enable (hardwired to 0)
cfg_command[8]	SERR Enable

Table 4-6: Bit Mapping on Header Command Register (Continued)

Bit	Name
cfg_command[7]	IDSEL Stepping/Wait Cycle Control (hardwired to 0)
cfg_command[6]	Parity Error Enable
cfg_command[5]	VGA Palette Snoop (hardwired to 0)
cfg_command[4]	Memory Write and Invalidate (hardwired to 0)
cfg_command[3]	Special Cycle Enable (hardwired to 0)
cfg_command[2]	Bus Master Enable
cfg_command[1]	Memory Address Space Decoder Enable
cfg_command[0]	IO Address Space Decoder Enable

The User Application must monitor the Bus Master Enable bit (`cfg_command[2]`) and refrain from transmitting requests while this bit is not set. This requirement applies only to requests; completions can be transmitted regardless of this bit.

cfg_dstatus[15:0]

This bus reflects the value stored in the Device Status register of the PCI Express Extended Capabilities Structure. Table 4-7 defines each bit in the `cfg_dstatus` bus. See the *PCI Express Base Specification* for detailed information.

Table 4-7: Bit Mapping on PCI Express Device Status Register

Bit	Name
cfg_dstatus[15:6]	Reserved
cfg_dstatus[5]	Transaction Pending
cfg_dstatus[4]	AUX Power Detected
cfg_dstatus[3]	Unsupported Request Detected
cfg_dstatus[2]	Fatal Error Detected
cfg_dstatus[1]	Non-Fatal Error Detected
cfg_dstatus[0]	Correctable Error Detected

cfg_dcommand[15:0]

This bus reflects the value stored in the Device Control register of the PCI Express Extended Capabilities Structure. Table 4-8 defines each bit in the `cfg_dcommand` bus. See the *PCI Express Base Specification* for detailed information.

Table 4-8: Bit Mapping of PCI Express Device Control Register

Bit	Name
cfg_dcommand[15]	Reserved
cfg_dcommand[14:12]	Max_Read_Request_Size
cfg_dcommand[11]	Enable No Snoop
cfg_dcommand[10]	Auxiliary Power PM Enable

Table 4-8: Bit Mapping of PCI Express Device Control Register (Continued)

Bit	Name
cfg_dcommand[9]	Phantom Functions Enable
cfg_dcommand[8]	Extended Tag Field Enable
cfg_dcommand[7:5]	Max_Payload_Size
cfg_dcommand[4]	Enable Relaxed Ordering
cfg_dcommand[3]	Unsupported Request Reporting Enable
cfg_dcommand[2]	Fatal Error Reporting Enable
cfg_dcommand[1]	Non-Fatal Error Reporting Enable
cfg_dcommand[0]	Correctable Error Reporting Enable

cfg_lstatus[15:0]

This bus reflects the value stored in the Link Status register in the PCI Express Extended Capabilities Structure. Table 4-9 defines each bit in the `cfg_lstatus` bus. See the *PCI Express Base Specification* for details.

Table 4-9: Bit Mapping of PCI Express Link Status Register

Bit	Name
cfg_lstatus[15:13]	Reserved
cfg_lstatus[12]	Slot Clock Configuration
cfg_lstatus[11]	Reserved
cfg_lstatus[10]	Reserved
cfg_lstatus[9:4]	Negotiated Link Width
cfg_lstatus[3:0]	Link Speed

cfg_lcommand[15:0]

This bus reflects the value stored in the Link Control register of the PCI Express Extended Capabilities Structure. Table 4-10 provides the definition of each bit in `cfg_lcommand` bus. See the *PCI Express Base Specification* for more details.

Table 4-10: Bit Mapping of PCI Express Link Control Register

Bit	Name
cfg_lcommand[15:8]	Reserved
cfg_lcommand [7]	Extended Synch
cfg_lcommand [6]	Common Clock Configuration
cfg_lcommand [5]	Retrain Link (Reserved for an endpoint device)
cfg_lcommand [4]	Link Disable
cfg_lcommand [3]	Read Completion Boundary
cfg_lcommand[2]	Reserved
cfg_lcommand [1:0]	Active State Link PM Control

Accessing Additional Registers through the Configuration Port

Configuration registers that are not directly mapped to the user interface can be accessed by configuration-space address using the ports shown in [Table 2-10, page 32](#).

The User Application must supply the read address as a DWORD address, not a byte address. To calculate the DWORD address for a register, divide the byte address by four. For example:

- The DWORD address of the Command/Status Register in the PCI Configuration Space Header is 01h. (The byte address is 04h.)
- The DWORD address for BAR0 is 04h. (The byte address is 10h.)

To read any register in this address space, the User Application drives the register DWORD address onto `cfg_dwaddr[9:0]`. The core drives the content of the addressed register onto `cfg_do[31:0]`. The value on `cfg_do[31:0]` is qualified by signal assertion on `cfg_rd_wr_done_n`. [Figure 4-22](#) illustrates an example with two consecutive reads from the Configuration Space.

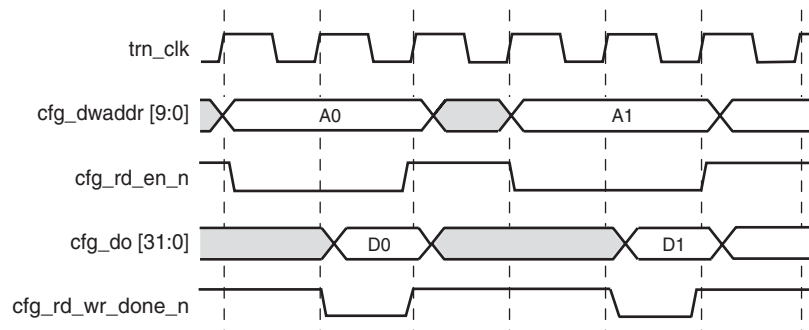


Figure 4-22: Example Configuration Space Access

Note that writing to the Configuration Space using `cfg_di[31:0]` and `cfg_wr_en_n` is not supported in this release. The core ignores any writes to the Configuration Space by the User Application.

Additional Packet Handling Requirements

The User Application must manage the following mechanisms to ensure protocol compliance, because the core does not manage them automatically.

Generation of Completions

The Endpoint Block Plus for PCIe does not generate Completions for Memory Reads or I/O requests made by a remote device. The user is expected to service these completions according to the rules specified in the *PCI Express Base Specification*.

Tracking Non-Posted Requests and Inbound Completions

The Endpoint Block Plus for PCIe does not track transmitted I/O requests or Memory Reads that have yet to be serviced with inbound Completions. The User Application is required to keep track of such requests using the Tag ID or other information.

Keep in mind that one Memory Read request can be answered by several Completion packets. The User Application must accept all inbound Completions associated with the original Memory Read until all requested data has been received.

The *PCI Express Base Specification* requires that an endpoint advertise infinite Completion Flow Control credits as a receiver; the endpoint can only transmit Memory Reads and I/O requests if it has enough space to receive subsequent Completions.

The Block Plus core does not keep track of receive-buffer space for Completion. Rather, it sets aside a fixed amount of buffer space for inbound Completions. The User Application must keep track of this buffer space to know if it can transmit requests requiring a Completion response.

See “[Performance Considerations on Receive Transaction Interface](#),” on page 74 for information about implementing this mechanism.

Reporting User Error Conditions

The User Application must report errors that occur during Completion handling using dedicated error signals on the core interface, and must observe the Device Power State before signaling an error to the core. If the User Application detects an error (for example, a Completion Timeout) while the device has been programmed to a non-D0 state, the User Application is responsible to signal the error after the device is programmed back to the D0 state.

After the User Application signals an error, the core reports the error on the PCI Express Link and also sets the appropriate status bit(s) in the Configuration Space. Because status bits must be set in the appropriate Configuration Space register, the User Application cannot generate error reporting packets on the transmit interface. The type of error-reporting packets transmitted depends on whether or not the error resulted from a Posted or Non-Posted Request. All user-reported errors cause Message packets to be sent to the Root Complex, unless the error is regarded as an Advisory Non-Fatal Error. For more information about Advisory Non-Fatal Errors, see Chapter 6 of the *PCI Express Base Specification*. Errors on Non-Posted Requests can result in either Messages to the Root Complex or Completion packets with non-Successful status sent to the original Requester.

Error Types

The User Application triggers six types of errors using the signals defined in [Table 2-11, page 35](#).

- End-to-end CRC ECRC Error
- Unsupported Request Error
- Completion Timeout Error
- Unexpected Completion Error
- Completer Abort Error
- Correctable Error

Multiple errors can be detected in the same received packet; for example, the same packet can be an Unsupported Request and have an ECRC error. If this happens, only one error should be reported. Because all user-reported errors have the same severity, the User Application design can determine which error to report. The `cfg_err_posted_n` signal, combined with the appropriate error reporting signal, indicates what type of error-reporting packets are transmitted. The user can signal only one error per clock cycle. See [Figures 4-23, 4-24, and 4-25](#), and [Tables 4-11 and 4-12](#).

Table 4-11: User-indicated Error Signaling

Reported Error	<code>cfg_err_posted_n</code>	Action
None	Don't care	No Action Taken
<code>cfg_err_ur_n</code>	0 or 1	0: If enabled, a Non-Fatal Error Message is sent. 1: A Completion with a non-successful status is sent.
<code>cfg_err_cpl_abort_n</code>	0 or 1	0: If enabled, a Non-Fatal Error message is sent. 1: A Completion with a non-successful status is sent.
<code>cfg_err_cpl_timeout_n</code>	Don't care	If enabled, a Non-Fatal Error Message is sent.
<code>cfg_err_ecrc_n</code>	Don't care	If enabled, a Non-Fatal Error Message is sent.
<code>cfg_err_cor_n</code>	Don't care	If enabled, a Correctable Error Message is sent.
<code>cfg_err_cpl_unexpected_n</code>	Don't care	Regarded as an Advisory Non-Fatal Error (ANFE); no action taken.

Table 4-12: Possible Error Conditions for TLPs Received by the User Application

Received TLP Type	Possible Error Condition					Error Qualifying Signal Status	
	Unsupported Request (<code>cfg_err_ur_n</code>)	Completion Abort (<code>cfg_err_cpl_abort_n</code>)	Correctable Error (<code>cfg_err_cor_n</code>)	ECRC Error (<code>cfg_err_ecrc_n</code>)	Unexpected Completion (<code>cfg_err_cpl_unexpect_n</code>)	Value to Drive on (<code>cfg_err_posted_n</code>)	Drive Data on (<code>cfg_err_tlp_cpl_header[47:0]</code>)
Memory Write	✓	×	N/A	✓	×	0	No
Memory Read	✓	✓	N/A	✓	×	1	Yes
I/O	✓	✓	N/A	✓	×	1	Yes
Completion	×	×	N/A	✓	✓	0	No

Whenever an error is detected in a Non-Posted Request, the User Application deasserts `cfg_err_posted_n` and provides header information on `cfg_err_tlp_cpl_header[47:0]` during the same clock cycle the error is reported, as illustrated in [Figure 4-23](#). The additional header information is necessary to construct the required Completion with non-Successful status. Additional information about when to assert or deassert `cfg_err_posted_n` is provided in the following sections.

If an error is detected on a Posted Request, the User Application instead asserts `cfg_err_posted_n`, but otherwise follows the same signaling protocol. This will result in a Non-Fatal Message to be sent, if enabled.

The core's ability to generate error messages can be disabled by the Root Complex issuing a configuration write to the Endpoint core's Device Control register and the PCI Command register setting the appropriate bits to 0. For more information about these registers, see Chapter 7 of the *PCI Express Base Specification*. However, error-reporting status bits are always set in the Configuration Space whether or not their Messages are disabled.

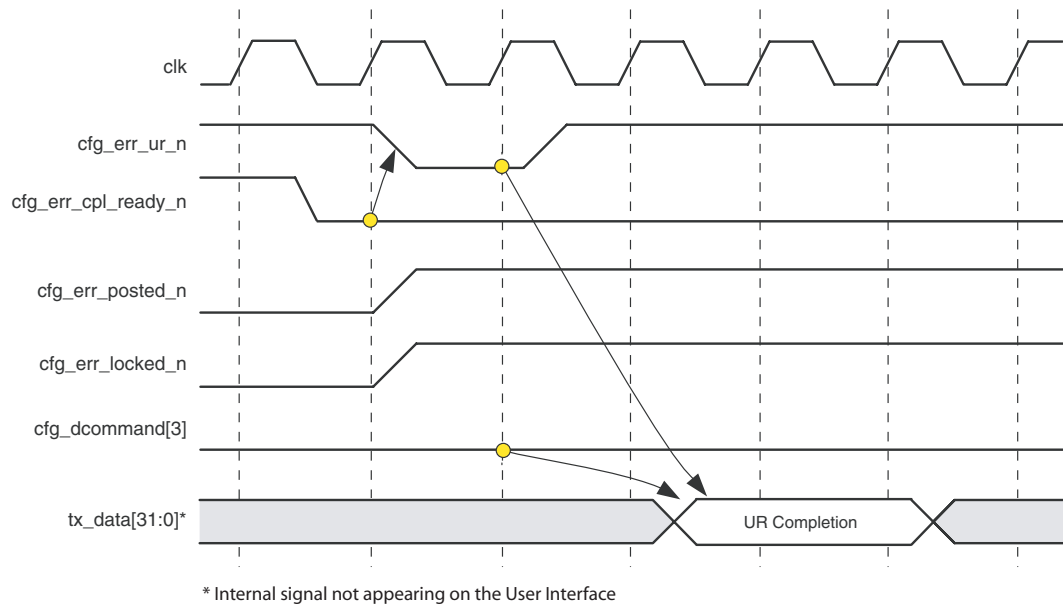


Figure 4-23: Signaling Unsupported Request for Non-Posted TLP

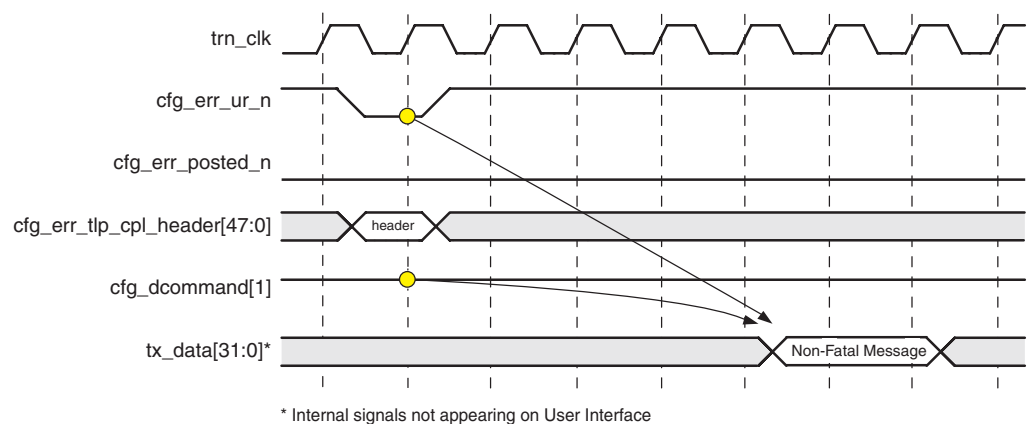


Figure 4-24: Signaling Unsupported Request for Posted TLP

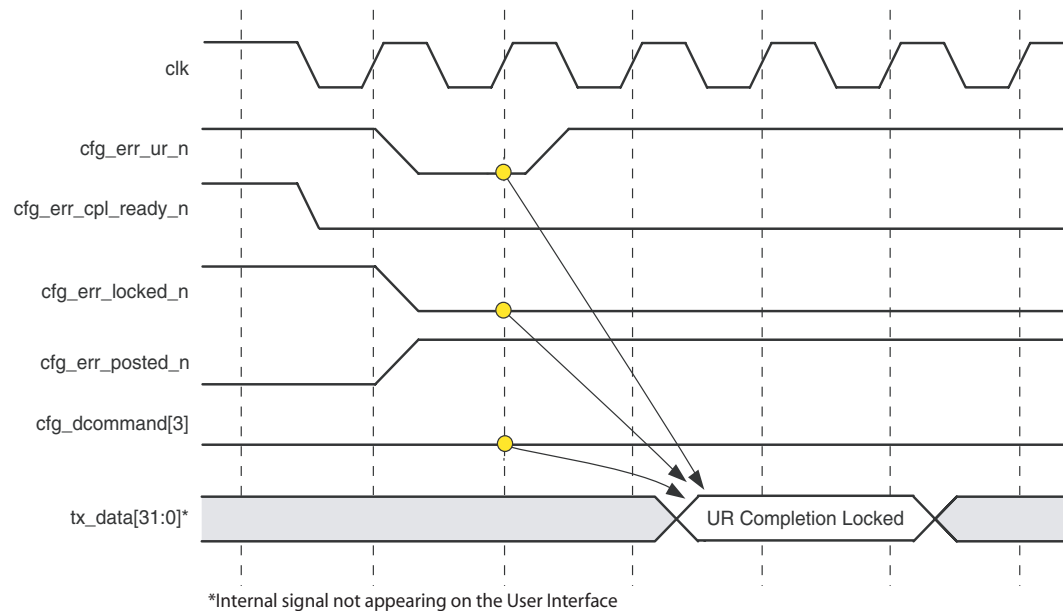


Figure 4-25: Signaling Locked Unsupported Request for Locked Non-Posted TLP

Completion Timeouts

The Block Plus core does not implement Completion timers; for this reason, the User Application must track how long its pending Non-Posted Requests have each been waiting for a Completion and trigger timeouts on them accordingly. The core has no method of knowing when such a timeout has occurred, and for this reason does not filter out inbound Completions for expired requests.

If a request times out, the User Application must assert `cfg_err_cpl_timeout_n`, which causes an error message to be sent to the Root Complex. If a Completion is later received after a request times out, the User Application must treat it as an Unexpected Completion.

Unexpected Completions

The Endpoint Block Plus for PCIe automatically reports Unexpected Completions in response to inbound Completions whose Requestor ID is different than the Endpoint ID programmed in the Configuration Space. These completions are not passed to the User Application. The current version of the core regards an Unexpected Completion to be an Advisory Non-Fatal Error (ANFE), and no message is sent.

Completer Abort

If the User Application is unable to transmit a normal Completion in response to a Non-Posted Request it receives, it must signal `cfg_err_cpl_abort_n`. The `cfg_err_posted_n` signal can also be set to 1 simultaneously to indicate Non-Posted and the appropriate request information placed on `cfg_err_tlp_cpl_header[47:0]`. This sends a Completion with non-Successful status to the original Requester, but does not send an Error Message. When in Legacy mode if the `cfg_err_locked_n` signal is set to 0 (to indicate the transaction causing the error was a locked transaction), a Completion Locked with Non-Successful status is sent. If the `cfg_err_posted_n` signal is set to 0 (to indicate a Posted transaction), no Completion is sent, but a Non-Fatal Error Message will be sent (if enabled).

Unsupported Request

If the User Application receives an inbound Request it does not support or recognize, it must assert `cfg_err_ur_n` to signal an Unsupported Request. The `cfg_err_posted_n` signal must also be asserted or deasserted depending on whether the packet in question is a Posted or Non-Posted Request. If the packet is Posted, a Non-Fatal Error Message will be sent out (if enabled); if the packet is Non-Posted, a Completion with a non-Successful status is sent to the original Requester. When in Legacy mode if the `cfg_err_locked_n` signal is set to 0 (to indicate the transaction causing the error was a locked transaction), a Completion Locked with Unsupported Request status is sent.

The Unsupported Request condition can occur for several reasons, including the following:

- An inbound Memory Write packet violates the User Application's programming model, for example, if the User Application has been allotted a 4 kB address space but only uses 3 kB, and the inbound packet addresses the unused portion. (Note: If this occurs on a Non-Posted Request, the User Application should use `cfg_err_cpl_abort_n` to flag the error.)
- An inbound packet uses a packet Type not supported by the User Application, for example, an I/O request to a memory-only device.

ECRC Error

The Block Plus core does not check the ECRC field for validity. If the User Application chooses to check this field, and finds the CRC is in error, it can assert `cfg_err_ecrc_n`, causing a Non-Fatal Error Message to be sent.

Power Management

The Endpoint Block Plus for PCIe supports the following power management modes:

- Active State Power Management (ASPM)
- Programmed Power Management (PPM)

Implementing these power management functions as part of the PCI Express design enables the PCI Express hierarchy to seamlessly exchange power-management messages to save system power. All power management message identification functions are implemented. The sections below describe the user logic definition to support the above modes of power management.

For additional information on ASPM and PPM implementation, see the *PCI Express Base Specification*.

Active State Power Management

The Active State Power Management (ASPM) functionality is autonomous and transparent from a user-logic function perspective. The core supports the conditions required for ASPM.

Programmed Power Management

To achieve considerable power savings on the PCI Express hierarchy tree, the core supports the following link states of Programmed Power Management (PPM):

- L0: Active State (data exchange state)
- L1: Higher Latency, lower power standby state
- L3: Link Off State

All PPM messages are always initiated by an upstream link partner. Programming the core to a non-D0 state, results in PPM messages being exchanged with the upstream link-partner. The PCI Express Link transitions to a lower power state after completing a successful exchange.

PPM L0 State

The L0 state represents *normal* operation and is transparent to the user logic. The core reaches the L0 (active state) after a successful initialization and training of the PCI Express Link(s) as per the protocol.

PPM L1 State

The following steps outline the transition of the core to the PPM L1 state.

1. The transition to a lower power PPM L1 state is always initiated by an upstream device, by programming the PCI Express device power state to D3-hot.
2. The core then throttles/stalls the user logic from initiating any new transactions on the user interface by deasserting `trn_tdst_rdy_n`. Any pending transactions on the user interface are however accepted fully and can be completed later.
3. The core exchanges appropriate power management messages with its link partner to successfully transition the link to a lower power PPM L1 state. This action is transparent to the user logic.
4. All user transactions are stalled for the duration of time when the device power state is non-D0.
5. The device power state is communicated to the user logic through the user configuration port interface. The user logic is responsible for performing a successful read operation to identify the device power state.

6. The user logic, after identifying the device power state as D1, can initiate a request through the `cfg_pm_wake_n` to the upstream link partner to transition the link to a higher power state L0.

PPM L3 State

The following steps outline the transition of the Endpoint Block Plus for PCIe to the PPM L3 state.

1. The core moves the link to the PPM L3 power state upon the upstream device programming the PCI Express device power state to D3.
2. During this duration, the core may receive a Power-Management Turn-Off (PME-turnoff) Message from its upstream partner.
3. The core notifies the user logic that power is about to be removed by asserting `cfg_to_turnoff_n`.
4. The core transmits a transmission of the Power Management Turn-off Acknowledge (PME-turnoff_ack) Message to its upstream link partner after a delay of no less than 250 nanoseconds.
5. The core closes all its interfaces, disables the Physical/Data-Link/Transaction layers and is ready for *removal* of power to the core.

Power-down negotiation follows these steps:

1. Before power and clock are turned off, the Root Complex or the Hot-Plug controller in a downstream switch issues a PME_TO_Ack broadcast message.
2. When the core receives this TLP, it asserts `cfg_to_turnoff_n` to the User Application.
3. When the User Application detects the assertion of `cfg_to_turnoff_n`, it must complete any packet in progress and stop generating any new packets.
4. The core sends a PME_TO_Ack after approximately 250 nanoseconds.

Generating Interrupt Requests

The Endpoint Block Plus for PCIe supports sending interrupt requests as either legacy interrupts or Message Signaled Interrupts (MSI). The mode is programmed using the MSI Enable bit in the Message Control Register of the MSI Capability Structure. For more information on the MSI capability structure see section 6.8 of the *PCI Local Base Specification v3.0*. If the MSI Enable bit is set to a 1, then the core will generate MSI request by sending Memory Write TLPs. If the MSI Enable bit is set to 0, the core generates legacy interrupt messages as long as the Interrupt Disable bit in the PCI Command Register is set to 0:

- `cfg_command[10] = 0`: interrupts enabled
- `cfg_command[10] = 1`: interrupts disabled (request are blocked by the core)

MSI Enable bit in MSI Control Register:

- `cfg_interrupt_msienable = 0`: Legacy Interrupt
- `cfg_interrupt_msienable = 1`: MSI

Note that the MSI Enable bit in the MSI control register and the Interrupt Disable bit in the PCI Command register are programmed by the Root Complex. The User Application has no direct control over these bits. Regardless of the interrupt type used, the user initiates interrupt requests through the use of `cfg_interrupt_n` and `cfg_interrupt_rdy_n` as shown in [Table 4-13](#)

Table 4-13: Interrupt Signalling

Port Name	Direction	Description
<code>cfg_interrupt_n</code>	Input	Assert to request an interrupt. Leave asserted until the interrupt is serviced.
<code>cfg_interrupt_rdy_n</code>	Output	Asserted when the core accepts the signaled interrupt request.

The User Application requests interrupt service in one of two ways, each of which are described below.

MSI Mode

- As shown in [Figure 4-26](#), the User Application first asserts `cfg_interrupt_n`. Additionally the User Application supplies a value on `cfg_interrupt_di[7:0]` if Multi-Vector MSI is enabled (see below).
- The core asserts `cfg_interrupt_rdy_n` to signal that the interrupt has been accepted and the core sends a MSI Memory Write TLP. On the following clock cycle, the User Application deasserts `cfg_interrupt_n` if no further interrupts are to be sent.

The MSI request is either a 32-bit addressable Memory Write TLP or a 64-bit addressable Memory Write TLP. The address is taken from the Message Address and Message Upper Address fields of the MSI Capability Structure, while the payload is taken from the Message Data field. These values are programmed by system software through configuration writes to the MSI Capability structure. When the core is configured for Multi-Vector MSI, system software may permit Multi-Vector MSI messages by programming a non-zero value to the Multiple Message Enable field.

The type of MSI TLP sent (32-bit addressable or 64-bit addressable) depends on the value of the Upper Address field in the MSI capability structure. By default, MSI messages are sent as 32-bit addressable Memory Write TLPs. MSI messages use 64-bit addressable

Memory Write TLPs only if the system software programs a non-zero value into the Upper Address register.

When Multi-Vector MSI messages are enabled, the User Application may override one or more of the lower-order bits in the Message Data field of each transmitted MSI TLP to differentiate between the various MSI messages sent upstream. The number of lower-order bits in the Message Data field available to the User Application is determined by the lesser of the value of the Multiple Message Capable field, as set in the CORE Generator, and the Multiple Message Enable field, as set by system software and available as the `cfg_interrupt_mmenable[2:0]` core output. The core masks any bits in `cfg_interrupt_di[7:0]` which are not configured by system software via Multiple Message Enable.

The following pseudo-code shows the processing required:

```
// Value MSI_Vector_Num must be in range: 0 ≤ MSI_Vector_Num ≤
(2^cfg_interrupt_mmenable)-1

if (cfg_interrupt_msienable) {           // MSI Enabled
    if (cfg_interrupt_mmenable > 0) {    // Multi-Vector MSI Enabled
        cfg_interrupt_di[7:0] = {Padding_0s, MSI_Vector_Num};
    } else {                             // Single-Vector MSI Enabled
        cfg_interrupt_di[7:0] = Padding_0s;
    }
} else {
    // Legacy Interrupts Enabled
}
```

For example:

1. If `cfg_interrupt_mmenable[2:0] == 000b` i.e 1 MSI Vector Enabled, then `cfg_interrupt_di[7:0] = cfg_interrupt_do[7:0]`;
2. if `cfg_interrupt_mmenable[2:0] == 101b` i.e 32 MSI Vectors Enabled, then `cfg_interrupt_di[7:0] = {{cfg_interrupt_do[7:5]}, {MSI_Vector#}}`;

where `MSI_Vector#` is a 5 bit value and is allowed to be $00000b \leq \text{MSI_Vector\#} \leq 11111b$

Legacy Interrupt Mode

- As shown in [Figure 4-26](#), the User Application first asserts `cfg_interrupt_n` and `cfg_interrupt_assert_n` to assert the interrupt. The User application should select a specific interrupt (INTA, INTB, INTC, or INTD) using `cfg_interrupt_di[7:0]` as shown in [Table 4-14](#).
- The core then asserts `cfg_interrupt_rdy_n` to indicate the interrupt has been accepted. On the following clock cycle, the User Application deasserts `cfg_interrupt_n` and, if the Interrupt Disable bit in the PCI Command register is set to 0, the core sends an assert interrupt message (Assert_INTA, Assert_INTB, and so forth).
- Once the User Application has determined that the interrupt has been serviced, it asserts `cfg_interrupt_n` while deasserting `cfg_interrupt_assert_n` to deassert the interrupt. The appropriate interrupt must be indicated via `cfg_interrupt_di[7:0]`.
- The core then asserts `cfg_interrupt_rdy_n` to indicate the interrupt deassertion has been accepted. On the following clock cycle, the User Application deasserts

cfg_interrupt_n and the core sends a deassert interrupt message (Deassert_INTA, Deassert_INTB, and so forth).

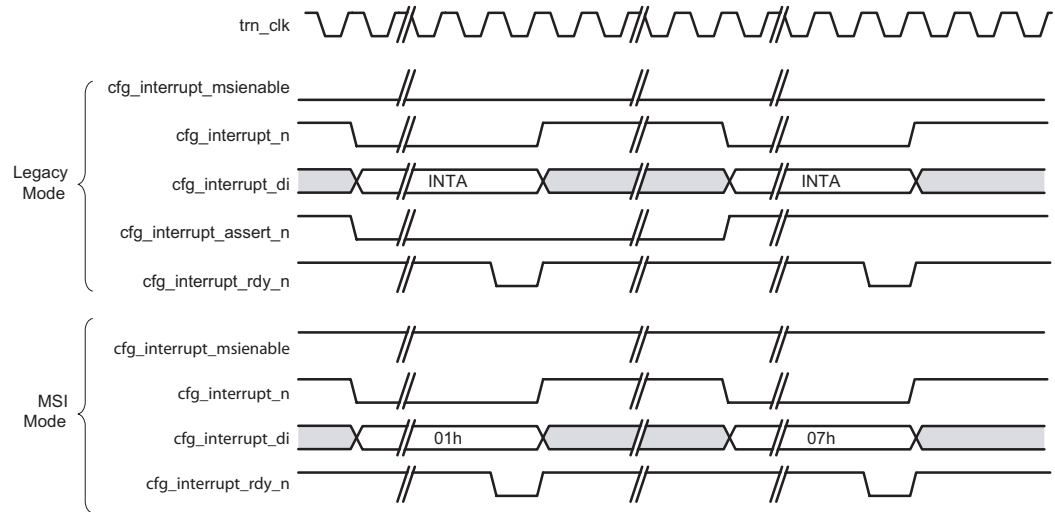


Figure 4-26: Requesting Interrupt Service: MSI and Legacy Mode

Table 4-14: Legacy Interrupt Mapping

cfg_interrupt_di[7:0] value	Legacy Interrupt
00h	INTA
01h	INTB
02h	INTC
03h	INTD
04h - FFh	Not Supported

Link Training: 4-Lane and 8-Lane Endpoints

The 4-lane and 8-lane Endpoint Block Plus cores for PCI Express are capable of operating at less than the maximum lane width as required by the *PCI Express Base Specification*. Two cases cause core to operate at less than its specified maximum lane width, as defined in the following sections.

Upstream Partner Supports Fewer Lanes

When the 4-lane core is connected to an upstream device that implements 1 lane, the 4-lane core trains and operates as a 1-lane device using lane 0, as shown in Figure 4-27. Similarly, if the 4-lane core is connected to a 2-lane upstream device, the core trains and operates as a 2-lane device using lanes 0 and 1.

When the 8-lane core is connected to an upstream device that only implements 4 lanes, it trains and operates as a 4-lane device using lanes 0-3. Additionally, if the upstream device only implements 1 or 2 lanes, the 8-lane core trains and operates as a 1- or 2-lane device.

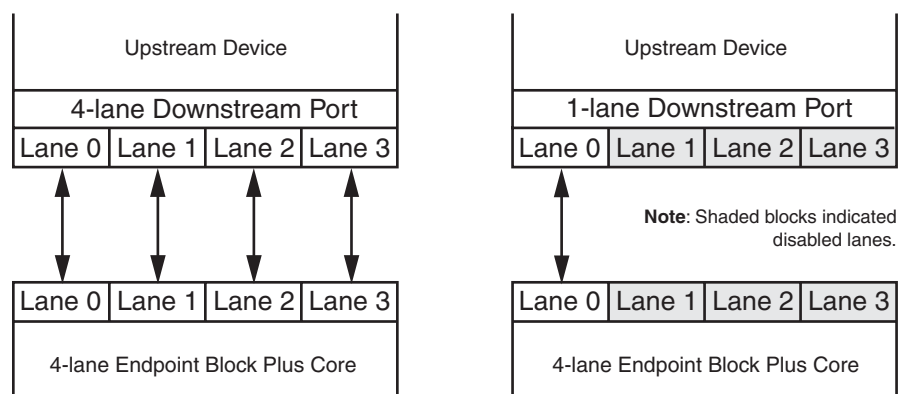


Figure 4-27: Scaling of 4-lane Endpoint Core from 4-lane to 1-lane Operation

Lane Becomes Faulty

If a link becomes faulty after training to the maximum lane width supported by the core and the link partner device, the core attempts to recover and train to a lower lane width, if available. If lane 0 becomes faulty, the link is irrecoverably lost. If any or all of lanes 1–7 become faulty, the link goes into *recovery* and attempts to recover the largest viable link with whichever lanes are still operational.

For example, when using the 8-lane core, loss of lane 1 yields a recovery to 1-lane operation on lane 0, whereas the loss of lane 6 yields a recovery to 4-lane operation on lanes 0-3. After recovery occurs, if the failed lane(s) becomes *alive* again, the core does not attempt to recover to a wider link width. The only way a wider link width can occur is if the link actually goes down and it attempts to retrain from scratch.

The `trn_clk` clock output is a fixed frequency configured in the CORE Generator GUI. `trn_clk` does not shift frequencies in case of link recovery or training down.

Clocking and Reset of the Block Plus Core

Reset

The Endpoint Block Plus core for PCIe uses `sys_reset_n` to reset the system, an asynchronous, active-low reset signal asserted during the PCI Express Fundamental Reset. Asserting this signal causes a hard reset of the entire core, including the RocketIO transceivers. After the reset is released, the core attempts to link train and resume normal operation. In a typical endpoint application, for example, an add-in card, a sideband reset signal is normally present and should be connected to `sys_reset_n`. For endpoint applications that do not have a sideband system reset signal, the initial hardware reset should be generated locally. Three reset events can occur in PCI Express:

- **Cold Reset.** A Fundamental Reset that occurs at the application of power. The signal `sys_reset_n` is asserted to cause the cold reset of the core.
- **Warm Reset.** A Fundamental Reset triggered by hardware without the removal and re-application of power. The `sys_reset_n` signal is asserted to cause the warm reset to the core.
- **Hot Reset:** In-band propagation of a reset across the PCI Express Link through the protocol. In this case, `sys_reset_n` is not used.

The User Application interface of the core has an output signal called `trn_reset_n`. This signal is deasserted synchronously with respect to `trn_clk`. `trn_reset_n` is asserted as a result of any of the following conditions:

- **Fundamental Reset:** Occurs (cold or warm) due to assertion of `sys_reset_n`.
- **PLL within the embedded Block:** Loses lock, indicating an issue with the stability of the clock input.
- **Loss of GTP/GTX PLL Lock:** The Lane 0 transceiver loses lock, indicating an issue with the PCI Express Link.
- **Exit from DL_Active State:** As defined in section 3.2.1 of the *PCI Express Base Specification*.

The `trn_reset_n` signal deasserts synchronously with `trn_clk` after all of the above reasons are resolved, allowing the core to attempt to train and resume normal operation.

Important Note: Systems designed to the PCI Express electro-mechanical specification provide a sideband reset signal, which uses 3.3V signaling levels—see the FPGA device data sheet to understand the requirements for interfacing to such signals.

Clocking

The Endpoint Block Plus core input system clock signal is called `sys_clk`. The core requires a 100 MHz or 250 MHz clock input. The clock frequency used must match the clock frequency selection in the CORE Generator GUI. For more information, see [Answer Record 18329](#).

In a typical PCI Express solution, the PCI Express reference clock is a Spread Spectrum Clock (SSC), provided at 100 MHz. Note that in most commercial PCI Express systems SSC cannot be disabled. For more information regarding SSC and PCI Express, see section 4.3.1.1.1 of the *PCI Express Base Specification*.

Synchronous and Non-synchronous Clocking

There are two ways to clock the PCI Express system:

- Using synchronous clocking, where a shared clock source is used for all devices.
- Using non-synchronous clocking, where each device has its own clock source.

Important Note: Xilinx recommends that designers use synchronous clocking when using the core. All add-in card designs must use synchronous clocking due to the characteristics of the provided reference clock. See the [Virtex-5 GTP Transceiver User Guide](#) (UG196) and device data sheet for additional information regarding reference clock requirements.

For synchronous clocked systems, each link partner device shares the same clock source. [Figures 4-28](#) and [4-30](#) show a system using a 100 MHz reference clock. When using the 250 MHz reference clock option, an external PLL must be used to do a multiply of 5/2 to convert the 100 MHz clock to 250 MHz, as illustrated in [Figures 4-29](#) and [4-31](#). See [Answer Record 18329](#) for more information about clocking requirements.

Further, even if the device is part of an embedded system, if the system uses commercial PCI Express root complexes or switches along with typical mother board clocking schemes, synchronous clocking should still be used as shown in [Figures 4-28](#) and [4-29](#).

[Figures 4-29](#) and [4-31](#) illustrate high-level representations of the board layouts. Designers must ensure that proper coupling, termination, and so forth are used when laying out the board.

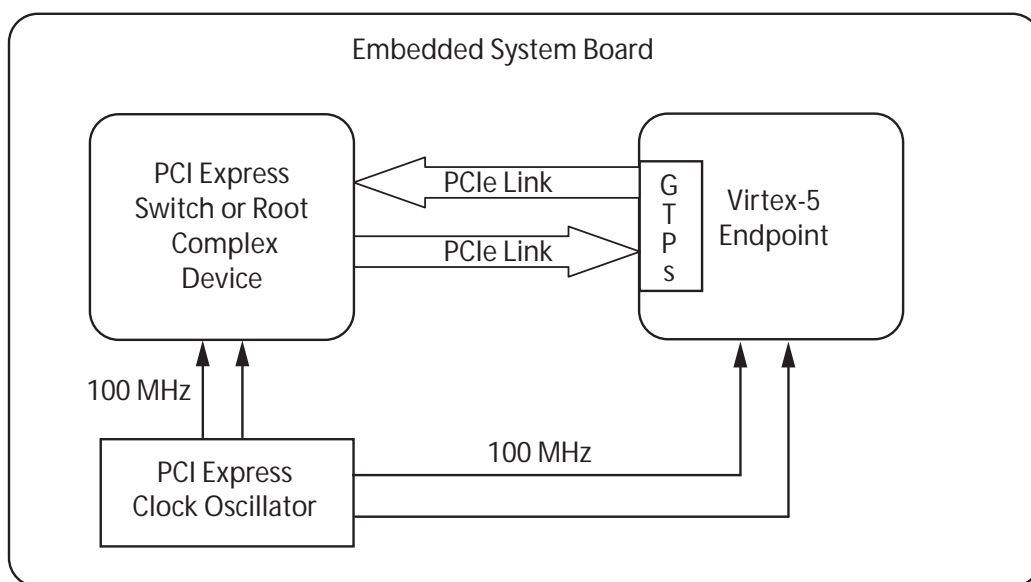


Figure 4-28: Embedded System Using 100 MHz Reference Clock

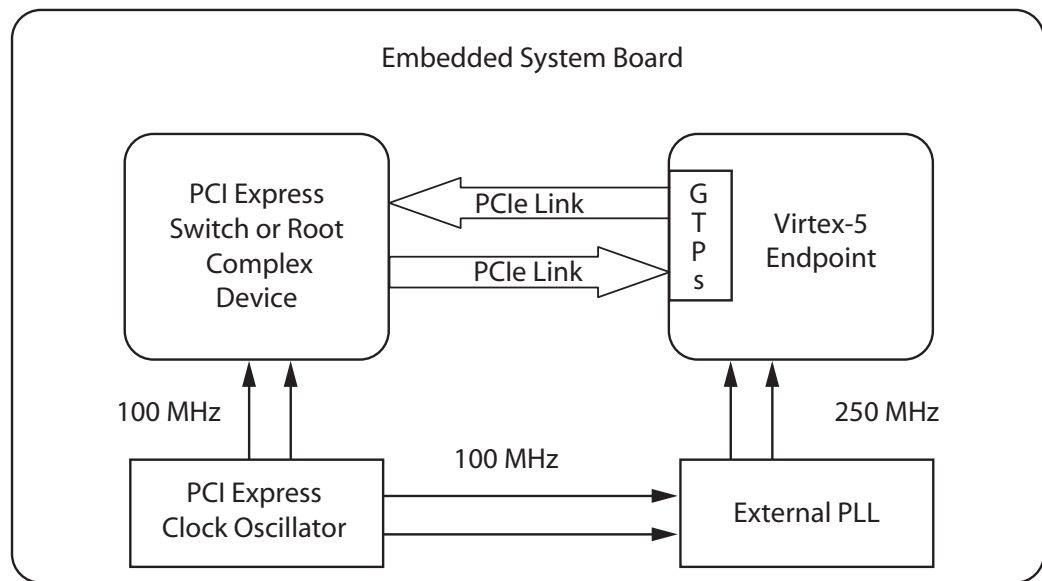


Figure 4-29: Embedded System Using 250 MHz Reference Clock

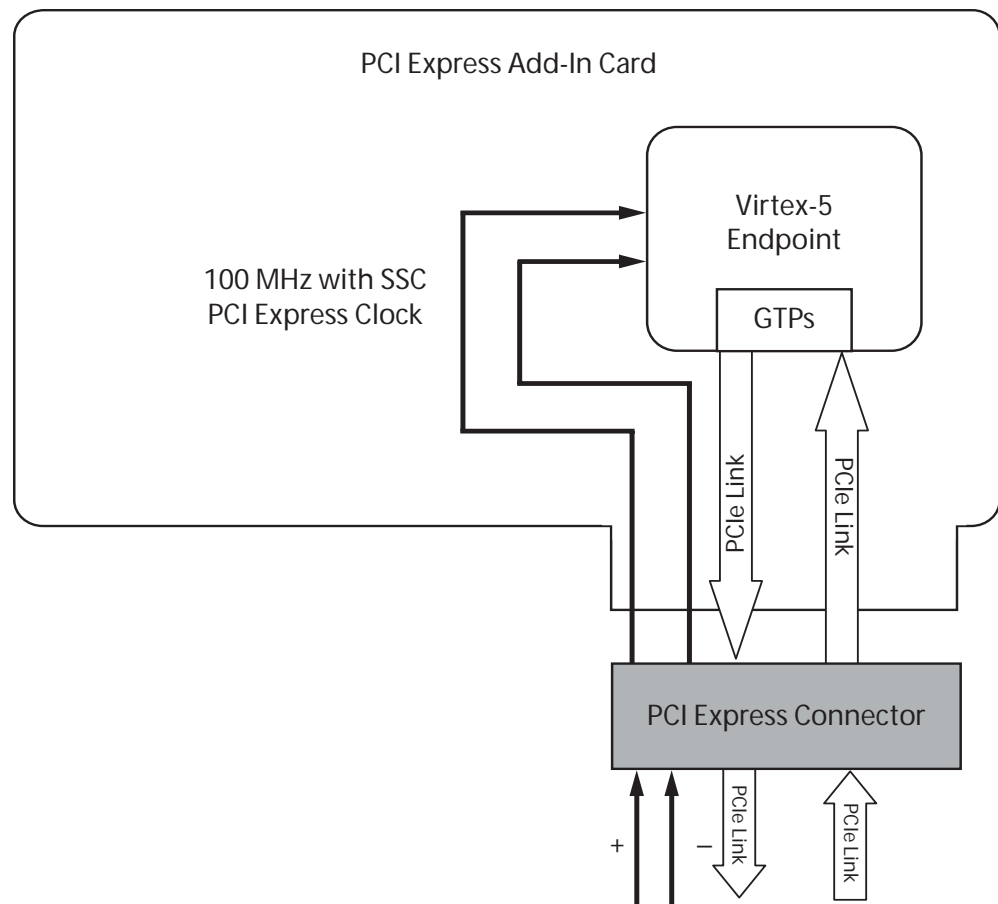


Figure 4-30: Open System Add-In Card Using 100 MHz Reference Clock

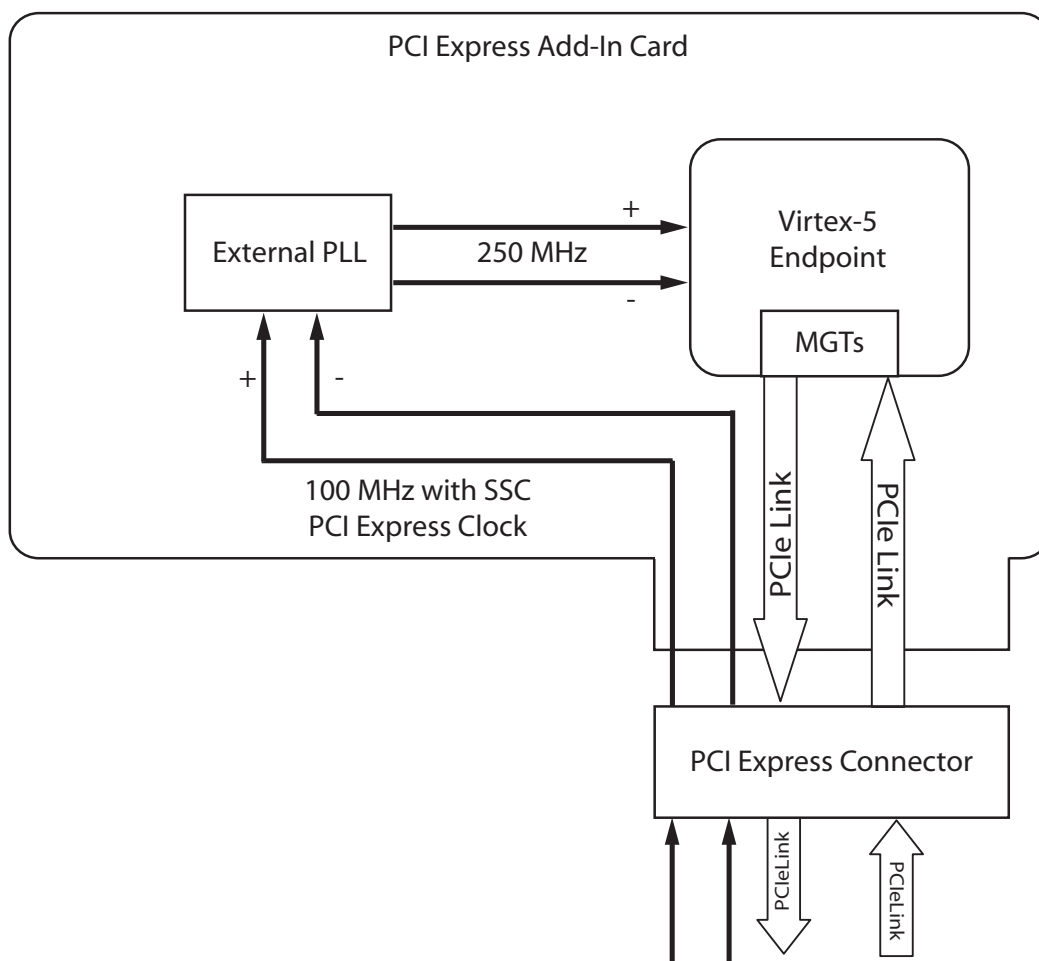


Figure 4-31: Open System Add-In Card Using 250 MHz Reference Clock

Core Constraints

The Endpoint Block Plus for PCI Express solutions require the specification of timing and other physical implementation constraints to meet specified performance requirements for PCI Express. These constraints are provided with the Endpoint Solutions in a User Constraints File (UCF).

To achieve consistent implementation results, a UCF containing these original, unmodified constraints must be used when a design is run through the Xilinx tools. For additional details on the definition and use of a UCF or specific constraints, see the Xilinx Libraries Guide and/or Development System Reference Guide.

Constraints provided with Block Plus solutions have been tested in hardware and provide consistent results. Constraints can be modified, but modifications should only be made with a thorough understanding of the effect of each constraint. Additionally, support is not provided for designs that deviate from the provided constraints.

Contents of the User Constraints File

Although the UCF delivered with each core shares the same overall structure and sequence of information, the content of each core's UCF varies. The sections that follow define the structure and sequence of information in a generic UCF file.

Part Selection Constraints: Device, Package, and Speedgrade

The first section of the UCF specifies the exact device for the implementation tools to target, including the specific part, package, and speed grade. In some cases, device-specific options are included.

User Timing Constraints

The User Timing constraints section is not populated; it is a placeholder for the designer to provide timing constraints on user-implemented logic.

User Physical Constraints

The User Physical constraints section is not populated; it is a placeholder for the designer to provide physical constraints on user-implemented logic.

Core Pinout and I/O Constraints

The Core Pinout and I/O constraints section contains constraints for I/Os belonging to the core's System (SYS) and PCI Express (PCI_EXP) interfaces. It includes location constraints for pins and I/O logic as well as I/O standard constraints.

Core Physical Constraints

Physical constraints are used to limit the core to a specific area of the device and to specify locations for clock buffering and other logic instantiated by the core.

Core Timing Constraints

This Core Timing constraints file defines clock frequency requirements for the core and specifies which nets the timing analysis tool should ignore.

Required Modifications

Several constraints provided in the UCF utilize hierarchical paths to elements within the Block Plus core. These constraints assume an instance name of *ep* for the core. If a different instance name is used, replace *ep* with the actual instance name in all hierarchical constraints.

For example:

Using *xilinx_pcie_ep* as the instance name, the physical constraint

```
INST "ep/BU2/U0/pcie_ep0/pcie_blk/clocking_i/use_pll.pll_adv_i"  
LOC = PLL_ADV_X0Y2;
```

becomes

```
INST "xilinx_pcie_ep/BU2/U0/pcie_ep0/pcie_blk/clocking_i/use_pll.pll_adv_i"  
LOC = PLL_ADV_X0Y2;
```

The provided UCF includes blank sections for constraining user-implemented logic. While the constraints provided adequately constrain the Block Plus core itself, they cannot adequately constrain user-implemented logic interfaced to the core. Additional constraints must be implemented by the designer.

Device Selection

The device selection portion of the UCF informs the implementation tools which part, package, and speed grade to target for the design. Because Block Plus cores are designed for specific part and package combinations, this section should not be modified by the designer.

The device selection section always contains a part selection line, but can also contain part or package-specific options. An example part selection line:

```
CONFIG PART = XC5VLX50T-FF1136-1
```

Core I/O Assignments

This section controls the placement and options for I/Os belonging to the core's System (SYS) interface and PCI Express (PCI_EXP) interface. NET constraints in this section control the pin location and I/O options for signals in the SYS group. Locations and options vary depending on which derivative of the core is used and should not be changed without fully understanding the system requirements.

For example:

```
NET "sys_reset_n" LOC = "AF21" | IOSTANDARD = LVCMOS33 | NODELAY;  
NET "sys_clk_p" LOC = "Y4"
```

```
NET "sys_clk_n" LOC = "Y3"

INST "refclk_ibuf" DIFF_TERM= "TRUE" ;

// The GTP has a fixed internal differential termination, which cannot be
disabled.
```

See “Clocking and Reset of the Block Plus Core,” page 94 for detailed information about reset and clock requirements.

Each GTP_DUAL tile contains two transceivers. The even numbered lane is always placed on transceiver 0 and the odd numbered lane is on transceiver 1 for a given GTP_DUAL in use. For GTP_DUAL pinout information, see “Package Placement Information” in “Chapter 4, Implementation,” of the *Virtex-5 GTP Transceiver User Guide (UG196)*.

INST constraints are used to control placement of signals that belong to the PCI_EXP group. These constraints control the location of the transceiver(s) used, which implicitly controls pin locations for the transmit and receive differential pair. Note that 1-lane cores consume *both* transceivers in a tile even though only one is used.

For example:

```
INST "ep/BU2/U0/pcie_ep0/pcie_blk/SIO/.pcie_gt_wrapper_i/GTD[0].GT_i"
LOC = GTP_DUAL_X0Y0;
```

Tables 5-1 shows how the GTP_DUALs in the INST constraints correspond to the GTP transceiver locations and the Block Plus core lane numbers for x1, x4, and x8 designs.

Table 5-1: Corresponding Lane Numbers and GTP Transceiver Locations

UCF GTP_DUAL Instance Name	GTP Transceiver 0 or 1	x01	x04	x08
GTD[0].GT_i	0	Lane 0	Lane 0	Lane 0
	1	N/C	Lane 1	Lane 1
GTD[2].GT_i	0		Lane 2	Lane 2
	1		Lane 3	Lane 3
GTD[4].GT_i	0			Lane 4
	1			Lane 5
GTD[6].GT_i	0			Lane 6
	1			Lane 7

Core Physical Constraints

Physical constraints are included in the constraints file to control the location of clocking and other elements and to limit the core to a specific area of the FPGA fabric. Specific physical constraints are chosen to match each supported device and package combination—it is very important to leave these constraints unmodified.

Physical constraints example:

```
INST "ep/BU2/U0/pcie_ep0/pcie_blk/clocking_i/use_pll.pll_adv_i"
LOC = PLL_ADV_X0Y2;

AREA_GROUP "GRP0" RANGE = SLICE_X58Y59:SLICE_X59Y56 ;

INST "ep/ep/pcie_blk/SIO/.pcie_gt_wrapper_i/gt_tx_data_k_reg_0" AREA_GROUP
= "GRP0" ;
```

Core Timing Constraints

Timing constraints are provided for all Block Plus solutions, although they differ for each product. In all cases they are crucial and must not be modified, except to specify the top-level hierarchical name. Timing constraints are divided into two categories:

- **TIG constraints.** Used on paths where specific delays are unimportant, to instruct the timing analysis tools to refrain from issuing *Unconstrained Path* warnings.
- **Frequency constraints.** Group clock nets into time groups and assign properties and requirements to those groups.
- **Delay constraints.** Controls delays on internal nets.

TIG constraints example:

```
NET "sys_reset_n" TIG;
```

Clock constraints example:

First, the input reference clock period is specified, which can be either 100 MHz or 250 MHz (selected in the CORE Generator GUI).

```
NET "sys_clk_c" PERIOD = 10ns; # OR
NET "sys_clk_c" PERIOD = 4ns;
```

Next, the internally generated clock net and period is specified, which can be either 100 MHz or 250 MHz. (Note that *both* clock constraints must be specified as either 100 MHz or 250 MHz.)

```
NET "ep/BU2/pcie_ep0/pcie_blk/SIO/.pcie_gt_wrapper_i/gt_refclk_out[0]" TNM_NET =
"MGTCCLK" ;
TIMESPEC "TS_MGTCCLK" = PERIOD "MGTCCLK" 250.00 MHz HIGH 50 % ; # OR
TIMESPEC "TS_MGTCCLK" = PERIOD "MGTCCLK" 100.00 MHz HIGH 50 % ;
```

Delay constraints example:

```
NET "ep/ep/pcie_blk/SIO/.pcie_gt_wrapper_i/gt_tx_elec_idle_reg*" MAXDELAY =
1.0 ns;
```

Relocating the Endpoint Block Plus Core

While Xilinx does not provide technical support for designs whose system clock input, GTP transceivers, or block RAM locations are different from the provided examples, it is possible to relocate the core within the FPGA. The locations selected in the provided examples are the recommended pinouts. These locations have been chosen based on the proximity to the PCIe block, which enables meeting 250 MHz timing, and because they are conducive to layout requirements for add-in card design. If the core is moved, the relative location of all transceivers and clocking resources should be maintained to ensure timing closure.

Supported Core Pinouts

Tables 5-2, 5-3, and 5-4 define the supported core pinouts for various LXT, SXT, FXT, and TXT part and package combinations. The CORE Generator software provides a UCF for the selected part and package that matches the content of this table.

Table 5-2: Supported Core Pinouts Virtex-5 LXT /SXT FPGAs

Package	Virtex-5 FPGA LXT / SXT Part	Lane	x1	x4	x8
FF665	XC5VLX30T XC5VSX35T	Lane 0/1	X0Y3	X0Y3	X0Y3
		Lane 2/3		X0Y2	X0Y2
		Lane 4/5			X0Y1
		Lane 6/7			X0Y0
	XC5VLX50T XC5VSX50T	Lane 0/1	X0Y4	X0Y4	X0Y4
		Lane 2/3		X0Y3	X0Y3
		Lane 4/5			X0Y2
		Lane 6/7			X0Y1
FF1136	XC5VLX50T XC5VLX85T XC5VSX50T	Lane 0/1	X0Y3	X0Y3	X0Y3
		Lane 2/3		X0Y2	X0Y2
		Lane 4/5			X0Y1
		Lane 6/7			X0Y0
	XC5VLX110T XC5VLX155T XC5VSX95T	Lane 0/1	X0Y4	X0Y4	X0Y4
		Lane 2/3		X0Y3	X0Y3
		Lane 4/5			X0Y2
		Lane 6/7			X0Y1
FF1738	XC5VLX110T XC5VLX155T XC5VLX220T	Lane 0/1	X0Y5	X0Y5	X0Y5
		Lane 2/3		X0Y4	X0Y4
		Lane 4/5			X0Y3
		Lane 6/7			X0Y2
	XC5VLX330T	Lane 0/1	X0Y7	X0Y7	X0Y7
		Lane 2/3		X0Y6	X0Y6
		Lane 4/5			X0Y5
		Lane 6/7			X0Y4
FF323	XC5VLX20T XC5VLX30T	Lane 0/1	X0Y2	X0Y2	NA
		Lane 2/3		X0Y1	NA

Table 5-3: Supported Core Pinouts Virtex-5 FXT FPGA

Package	Virtex-5 FPGA FXT Part	PCIe Block Location	Lane	x1	x4	x8
FF665	XC5VFX30T	X0Y0	Lane 0/1	X0Y3	X0Y3	X0Y3
			Lane 2/3		X0Y2	X0Y2
			Lane 4/5			X0Y1
			Lane 6/7			X0Y0
	XC5VFX70T	X0Y1	Lane 0/1	X0Y5	X0Y5	X0Y5
			Lane 2/3		X0Y4	X0Y4
			Lane 4/5			X0Y3
			Lane 6/7			X0Y2
FF1136	XC5VFX70T XC5VFX100T	X0Y0	Lane 0/1	X0Y3	X0Y3	X0Y3
			Lane 2/3		X0Y2	X0Y2
			Lane 4/5			X0Y1
			Lane 6/7			X0Y0
	XC5VFX70T XC5VFX100T	X0Y1	Lane 0/1	X0Y4	X0Y4	X0Y4
			Lane 2/3		X0Y3	X0Y3
			Lane 4/5			X0Y2
			Lane 6/7			X0Y1
	XC5VFX70T XC5VFX100T	X0Y2	Lane 0/1	X0Y7	X0Y7	X0Y7
			Lane 2/3		X0Y6	X0Y6
			Lane 4/5			X0Y5
			Lane 6/7			X0Y4

Table 5-3: Supported Core Pinouts Virtex-5 FXT FPGA (Continued)

Package	Virtex-5 FPGA FXT Part	PCIe Block Location	Lane	x1	x4	x8
FF1738	XC5VFX100T	X0Y0	Lane 0/1	X0Y3	X0Y3	X0Y3
			Lane 2/3		X0Y2	X0Y2
			Lane 4/5			X0Y1
			Lane 6/7			X0Y0
	XC5VFX100T	X0Y1	Lane 0/1	X0Y4	X0Y4	X0Y4
			Lane 2/3		X0Y3	X0Y3
			Lane 4/5			X0Y2
			Lane 6/7			X0Y1
	XC5VFX100T	X0Y2	Lane 0/1	X0Y7	X0Y7	X0Y7
			Lane 2/3		X0Y6	X0Y6
			Lane 4/5			X0Y5
			Lane 6/7			X0Y4
	XC5VFX130T	X0Y0	Lane 0/1	X0Y1	X0Y1	X0Y3
			Lane 2/3		X0Y0	X0Y2
			Lane 4/5			X0Y1
			Lane 6/7			X0Y0
	XC5VFX130T	X0Y1	Lane 0/1	X0Y3	X0Y3	X0Y4
			Lane 2/3		X0Y2	X0Y3
			Lane 4/5			X0Y2
			Lane 6/7			X0Y1
	XC5VFX130T	X0Y2	Lane 0/1	X0Y7	X0Y7	X0Y9
			Lane 2/3		X0Y6	X0Y8
			Lane 4/5			X0Y7
			Lane 6/7			X0Y6
	XC5VFX200T	X0Y0	Lane 0/1	X0Y0	X0Y1	X0Y3
			Lane 2/3		X0Y0	X0Y2
			Lane 4/5			X0Y1
			Lane 6/7			X0Y0

Table 5-3: Supported Core Pinouts Virtex-5 FXT FPGA (Continued)

Package	Virtex-5 FPGA FXT Part	PCIe Block Location	Lane	x1	x4	x8
FF1738	XC5VFX200T	X0Y1	Lane 0/1	X0Y2	X0Y3	X0Y4
			Lane 2/3		X0Y2	X0Y3
			Lane 4/5			X0Y2
			Lane 6/7			X0Y1
	XC5VFX200T	X0Y2	Lane 0/1	X0Y4	X0Y5	X0Y5
			Lane 2/3		X0Y4	X0Y4
			Lane 4/5			X0Y3
			Lane 6/7			X0Y2
	XC5VFX200T	X0Y3	Lane 0/1	X0Y8	X0Y9	X0Y10
			Lane 2/3		X0Y8	X0Y9
			Lane 4/5			X0Y8
			Lane 6/7			X0Y7

Table 5-4: Supported Core Pinouts Virtex-5 TXT FPGA

Package	Virtex-5 FPGA TXT Part	Lane	x1	x4	x8
FF1759	XC5VTX150T XC5VTX240T	Lane 0/1	X1Y5	X1Y5	X1Y5
		Lane 2/3		X1Y4	X1Y4
		Lane 4/5			X1Y3
		Lane 6/7			X1Y2

Hardware Verification

PCI Special Interest Group

Xilinx attends the PCI Special Interest Group ([PCI-SIG](#)) Compliance Workshops to verify the Endpoint Block Plus Wrapper and Integrated Endpoint Blocks compliance and interoperability with various systems available on the market and some not yet released. While Xilinx cannot list the actual systems tested at the PCISIG Compliance Workshops due to requirements of the PCISIG by-laws, Xilinx IP designers can view the [PCISIG integrators list](#).

PCI SIG Integrators List

The integrators list confirms that Xilinx has satisfied the PCI-SIG Compliance Program and that the Endpoint Block Plus wrapper successfully interoperated with other available systems at the event. Virtex-5 entries can be found in the Components and Add-In Cards sections under the company name Xilinx.

Hardware validation testing is performed at Xilinx for each release of the Endpoint Block Plus wrapper with the list of platforms defined in [Table 6-1](#). Hardware testing performed by Xilinx also available to customers includes the PCIECV tool available from the PCI-SIG website, BMD Design, available from XAPP 1052, and the MET design, available from XAPP 1022.

Table 6-1: Platforms Tested

System or Motherboard	Processor	Chipset	OS	Lanes
Dell Poweredge™ 1900 Server	Intel® Xeon® 5110 dual-core 1.60 GHz	Intel E5000	<ul style="list-style-type: none"> Windows® XP Professional v2002 Service Pack 2 Red Hat® Enterprise Linux 4. 	x8
SuperMicro® X6DH8-G2 motherboard	Intel 64-bit Xeon 2.4 GHz, 800 MHz FSB	Intel 7520 (Lindenhurst)	Windows XP Professional Service Pack 2	x8
SuperMicro H8DCE motherboard	Dual-core AMD Opteron™ 200 Series processor	NVIDIA® nForce® Pro 2200 (CK804)	<ul style="list-style-type: none"> Windows XP Professional Service Pack 2 Red Hat Enterprise Linux 4 	x8

Table 6-1: Platforms Tested (Continued)

System or Motherboard	Processor	Chipset	OS	Lanes
ASUS® P5B-VM DO motherboard	Intel Pentium® 4 531 Single Processor 3.0 GHz	Intel Q965 Express / Intel ICH8D0	Windows XP Professional v2002 Service Pack 2	x8
ASUSTeK Computer P5N32-SLI-Deluxe	Intel Pentium D 940 dual-core 3.2 GHz	NVIDIA nForce®4 SLI™ Intel Edition	Windows XP Professional v2002 Service Pack 2	x1, x4
Intel Platform Development Kit	Intel 64-bit dual-core Xeon 3.8 GHz, 1333 MHz FSB	Intel 5400 (Seaburg)	<ul style="list-style-type: none"> Windows XP Professional v2002 Service Pack 2 Red Hat Enterprise Linux 4 	x16

FPGA Configuration

This chapter discusses how to configure the Virtex-5 FPGA so that the device can link up and be recognized by the system. This information is provided so you can choose the correct FPGA configuration method for your system and verify that it will work as expected.

This chapter discusses how specific requirements of the *PCI Express Base Specification* and *PCI Express Card Electromechanical Specification* apply to FPGA configuration. Where appropriate, Xilinx recommends that you read the actual specifications for detailed information. This chapter is divided into three sections:

- “[Configuration Access Time.](#)” Several specification items govern when an Endpoint device needs to be ready to receive configuration accesses from the host (Root Complex).
- “[Board Power in Real-world Systems.](#)” Understanding real-world system constraints related to board power and how they affect the specification requirements.
- “[Recommendations.](#)” Describes methods for FPGA configuration and includes sample problem analysis for FPGA configuration timing issues.

Configuration Terminology

In this chapter, the following terms are used to differentiate between FPGA configuration and configuration of the PCI Express device:

- **Configuration of the FPGA.** *FPGA configuration* is used.
- **Configuration of the PCI Express device.** After the link is active, *configuration* is used.

Configuration Access Time

In standard systems for PCI Express, when the system is powered up, configuration software running on the processor starts scanning the PCI Express bus to discover the machine topology.

The process of scanning the PCI Express hierarchy to determine its topology is referred to as the *enumeration process*. The root complex accomplishes this by initiating configuration transactions to devices as it traverses and determines the topology.

All PCI Express devices are expected to have established the link with their link partner and be ready to accept configuration requests during the enumeration process. As a result, there are requirements as to when a device needs to be ready to accept configuration requests after power up; if the requirements are not met, the following occurs:

- If a device is not ready and does not respond to configuration requests, the root complex does not discover it and treats it as non-existent.
- The operating system will not report the device's existence and the user's application will not be able to communicate with the device.

Choosing the appropriate FPGA configuration method is key to ensuring the device is able to communicate with the system in time to achieve link up and respond to the configuration accesses.

Configuration Access Specification Requirements

Two PCI Express specification items are relevant to configuration access:

1. Section 6.6 of *PCI Express Base Specification*, rev 1.1 states "A system must guarantee that all components intended to be software visible at boot time are ready to receive Configuration Requests within 100 ms of the end of Fundamental Reset at the Root Complex." For detailed information about how this is accomplished, see the specification; it is beyond the scope of this discussion.

Xilinx has verified the Block Plus wrapper using the Virtex-5 FPGA Integrated Block for PCI Express is compliant with this requirement. From the end of the fundamental reset, the Block Plus wrapper is ready to receive configuration accesses within the 100 ms window.

Xilinx compliance to this specification is validated by the PCI Express-CV tests. The [PCI Special Interest Group \(PCI-SIG\)](#) provides the PCI Express Configuration Test Software to verify the device meets the requirement of being able to receive configuration accesses within 100 ms of the end of the fundamental reset. The software, available to any member of the PCI-SIG, generates several resets using the in-band reset mechanism and PERST# toggling to validate robustness and compliance to the specification. The Block Plus wrapper using the Virtex-5 FPGA Integrated Block for PCI Express has successfully passed this test and other requirements for compliance and is located on the [PCI-SIG integrator's list](#).

2. Section 6.6 of *PCI Express Base Specification*, rev 1.1 defines three parameters necessary "where power and PERST# are supplied." The parameter T_{PVPERL} applies to FPGA configuration timing and is defined as:

T_{PVPERL} - PERST# must remain active at least this long after power becomes valid.

The *PCI Express Base Specification* does not give a specific value for T_{PVPERL} – only its meaning is defined. The most common form factor used by designers with the Endpoint Block Plus wrapper is an ATX-based form factor. The *PCI Express Card Electromechanical Specification* focuses on requirements for ATX-based form factors. This applies to most designs targeted to standard desktop or server type motherboards. [Figure 7-1](#) shows the relationship between Power Stable and PERST#. (This figure is based on Figure 2-10 from section 2.1 of *PCI Express Card Electromechanical Specification*, rev 1.1.)

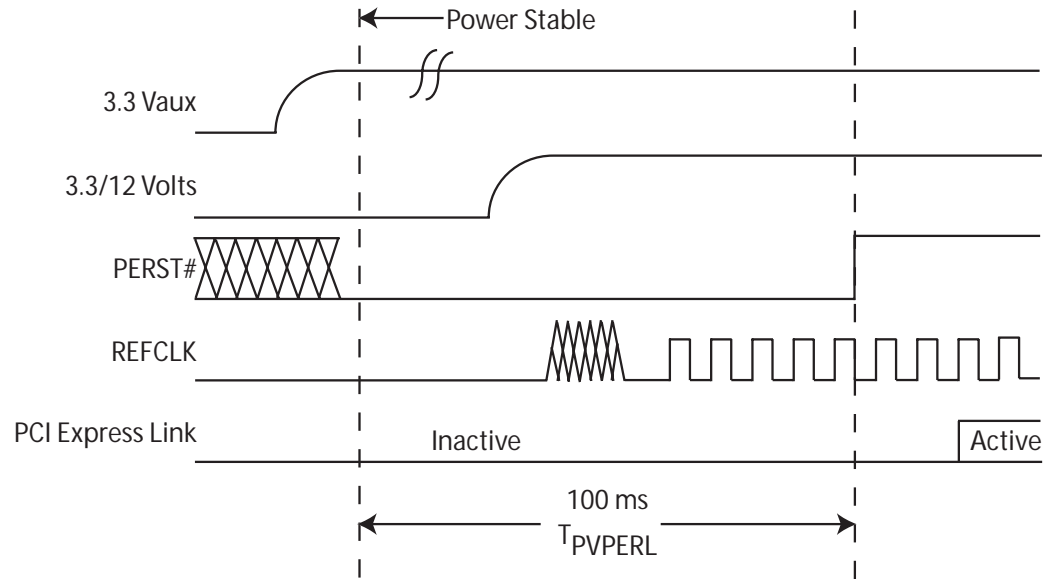


Figure 7-1: Power Up

Section 2.6.2 of the *PCI Express Card Electromechanical Specification* defines T_{PVPERL} as a minimum of 100 ms, indicating that from the time power is stable the system reset will be asserted for at least 100 ms (as shown in [Table 7-1](#)).

Table 7-1: T_{PVPERL} Specification

Symbol	Parameter	Min	Max	Units
T_{PVPERL}	Power stable to PERST# inactive	100		ms

From [Figure 7-1](#) and [Table 7-1](#), it is possible to obtain a simple equation to define the FPGA configuration time as follows:

$$\text{FPGA Configuration Time} \leq T_{\text{PWRVLD}} + T_{\text{PVPERL}}$$

Given that T_{PVPERL} is defined as 100 ms minimum, this becomes:

$$\text{FPGA Configuration Time} \leq T_{\text{PWRVLD}} + 100 \text{ ms}$$

Note: Although T_{PWRVLD} is included in the previous equation, it has yet to be defined in this discussion because it depends on the type of system in use. The next section, “[Board Power in Real-world Systems](#)” defines T_{PWRVLD} for both ATX-based and non ATX-based systems.

FPGA configuration time is only relevant at cold boot; subsequent warm or hot resets do not cause reconfiguration of the FPGA. If you suspect the design is having problems due to FPGA configuration, issue a warm reset as a simple test, which resets the system, including the PCI Express link, but keeps the board powered. If the problem does not appear, the issue could be FPGA configuration time related.

Board Power in Real-world Systems

Several boards are used in PCI Express systems. The *ATX Power Supply Design* specification, endorsed by Intel, is used as a guideline and for this reason followed in the majority of mother boards and 100% of the time if it is an Intel-based motherboard. The relationship between power rails and power valid signaling is described in the [ATX 12V Power Supply Design Guide](#). [Figure 7-2](#), redrawn here and simplified to show the information relevant to FPGA configuration, is based on the information and diagram found in section 3.3 of the *ATX 12V Power Supply Design Guide*. For the entire diagram and definition of all parameters, see the *ATX 12V Power Supply Design Guide*.

[Figure 7-2](#) shows that power stable indication from [Figure 7-1](#) for the PCI Express system is indicated by the assertion of `PWR_OK`. `PWR_OK` asserts high after some delay once the power supply has reached 95% of nominal.

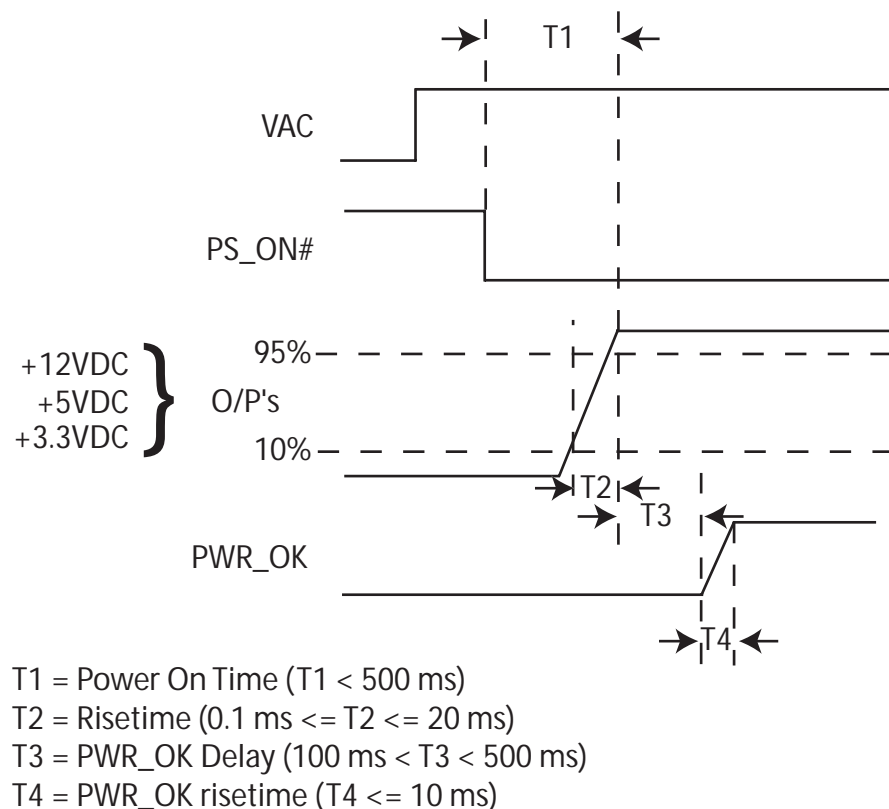


Figure 7-2: ATX Power Supply

Figure 7-2 shows that power is actually valid before PWR_OK is asserted high. This is represented by T3 and is the PWR_OK delay. The *ATX 12V Power Supply Design Guide* defines PWR_OK as $100 \text{ ms} < T3 < 500 \text{ ms}$, indicating the following: From the point at which the power level reaches 95% of nominal, there is a minimum of at least 100 ms but no more than 500 ms of delay before PWR_OK is asserted. Remember, according to the *PCI Express Card Electromechanical Specification*, the PERST# is guaranteed to be asserted a minimum of 100 ms from when power is stable indicated in an ATX system by the assertion of PWR_OK.

Again, the FPGA configuration time equation is:

$$\text{FPGA Configuration Time} \leq T_{\text{PWRVLD}} + 100 \text{ ms}$$

T_{PWRVLD} is defined as PWR_OK delay period, that is, T_{PWRVLD} represents the amount of time that power is valid in the system before PWR_OK is asserted. This time can be added to the amount of time the FPGA has to configure. The minimum values of T2 and T4 are negligible and considered zero for purposes of these calculations. For ATX-based motherboards, which represent the majority of real-world motherboards in use, T_{PWRVLD} can be defined as:

$$100 \text{ ms} \leq T_{\text{PWRVLD}} \leq 500 \text{ ms}$$

This provides the following requirement for FPGA configuration time in both ATX and non-ATX-based motherboards:

- FPGA Configuration Time ≤ 200 ms (for ATX based motherboard)
- FPGA Configuration Time ≤ 100 ms (for non-ATX based motherboard)

The second equation for the non-ATX based motherboards assumes a T_{PWRVLD} value of 0 ms because it is not defined in this context. Designers with non-ATX based motherboards should evaluate their own power supply design to obtain a value for T_{PWRVLD} .

Recommendations

Xilinx recommends using the [Platform Flash XL High-Density Storage and Configuration Device](#) (XCF128X) in Slave Map x16 Mode with a CCLK frequency of 50 MHz, which allows time for FPGA configuration on any Virtex-5 FPGA in ATX-based motherboards. Other valid configuration options are represented by green cells in [Table 7-2](#) and [Table 7-3](#) depending on the type of system in use. This section discusses these recommendations and includes sample analysis of potential problems that may arise during FPGA configuration.

FPGA Configuration Times for Virtex-5 Devices

During power up, the FPGA configuration sequence is performed in four steps:

1. Wait for POR (Power on Reset) for all voltages (V_{CCInt} , V_{CCAUX} , and V_{CC0}) in the FPGA to trip, referred to as POR Trip Time
2. Wait for completion (deassertion) of INIT to allow the FPGA to initialize before accepting a bitstream transfer.

Note: As a general rule, steps 1 and 2 require ≤ 50 ms

3. Wait for assertion of DONE, the actual time required for a bitstream to transfer, and depends on the following:
 - ♦ Bitstream size
 - ♦ Clock frequency
 - ♦ Transfer mode used in the Flash Device
 - SPI = Serial Peripheral Interface
 - BPI = Byte Peripheral Interface
 - PFP = Platform Flash PROMs

For detailed information about the configuration process, see the *Virtex-5 FPGA Configuration User Guide* ([UG191](#)).

[Tables 7-2](#) and [7-3](#) show the comparative data for all Virtex-5 FPGA LXT and SXT devices with respect to a variety of flash devices and programming modes. The default clock rate for configuring the device is always 2 Mhz. Any reference to a different clock rate implies a change in the settings of the device being used to program the FPGA. The configuration clock (CCLK), when driven by the FPGA, has variation and is not exact. See the *Virtex-5 FPGA Configuration Guide* (UG191) for more information on CCLK tolerances.

Configuration Time Matrix: ATX Motherboards

Table 7-2 shows the configuration methods that allow the device to be configured before the end of the fundamental reset in ATX-based systems. The table values represent the bitstream transfer time only. The matrix is color-coded to show which configuration methods allow the device to configure within 200 ms once the FPGA initialization time is included. Choose a configuration method shaded in green when using ATX-based systems to ensure that the device is recognized.

Table 7-2: Configuration Time Matrix (ATX Motherboards): Virtex-5 FPGA
Bitstream Transfer Time in Milliseconds

Virtex-5 FPGA	Bitstream (Bits)	SPIx1 ⁽¹⁾	BPIx16 ⁽²⁾ (Page mode)	PFPx8 ⁽³⁾	XCF128X (Master-BPIx16)	XCF128X ⁽⁴⁾ (Slave-SMAPx16)
XC5VLX20T	6,251,200	125	57	24	25	8
XC5VLX30T	9,371,136	187	85	35	38	12
XC5VLX50T	14,052,352	281	128	53	57	18
XC5VLX85T	23,341,312	467	213	88	94	29
XC5VLX110T	31,118,848	622	284	118	125	39
XC5VLX155T	43,042,304	861	392	163	174	54
XC5VLX220T	55,133,696	1103	503	209	222	69
XC5VLX330T	82,696,192	1654	754	313	333	103
XC5VSX35T	13,349,120	267	122	51	54	17
XC5VSX50T	20,019,328	400	182	76	81	25
XC5VSX95T	35,716,096	714	326	135	144	45
XC5VSX240T	79,610,368	1592	726	302	321	100
XC5VFX30T	13,517,056	270	123	51	55	17
XC5VFX70T	27,025,408	541	246	102	109	34
XC5VFX100T	39,389,696	788	359	149	159	49
XC5VFX130T	49,234,944	985	449	186	199	62
XC5VFX200T	70,856,704	1417	646	268	286	89
XC5VTX150T	43,278,464	866	394	164	175	54
XC5VTX240T	65,755,648	1315	599	249	265	82
<div> <div></div> Bitstream Transfer Time + FPGA INIT Time (50 ms) ≤ 200 ms. <div></div> Bitstream Transfer Time + FPGA INIT Time (50 ms) > 200 ms <div></div> Bitstream Transfer Time > 200 ms </div>						

1. SPI flash assumptions: 50 MHz MAX
2. BPIx16 assumptions: P30 4-page read with 4-cycle first page read (4-1-1-1), maximum configuration time
3. PFP assumptions: 33 MHz MAX
4. XCF128X Slave-SMAPx16 assumptions: CCLK=50 MHz

Configuration Time Matrix: Non-ATX-Based Motherboards

Table 7-3 shows the configuration methods that allow the device to be configured before the end of the fundamental reset in non-ATX-based systems. This assumes T_{PWRVLD} is zero. The table values represent the bitstream transfer time only. The matrix is color-coded to show which configuration methods allow the device to configure within 100 ms once the FPGA initialization time is included. Choose a configuration method shaded in green when using non-ATX-based systems to ensure that the device is recognized.

It is also obvious that for some of the larger FPGAs, it may not be possible to configure within the 100 ms window. In these cases, evaluate your system to see if any margin is available that can be assigned to T_{PWRVLD} .

Table 7-3: Configuration Time Matrix (Generic Platforms: Non-ATX Motherboards): Virtex-5 FPGA Bitstream Transfer Time in Milliseconds

Virtex-5 FPGA	Bitstream (Bits)	SP1x1 ⁽¹⁾	BPIx16 ⁽²⁾ (Page mode)	PFPx8 ⁽³⁾	XCF128X (Master-BPIx16)	XCF128X ⁽⁴⁾ (Slave-SMAPx16)
XC5VLX20T	6,251,200	125	57	24	25	8
XC5VLX30T	9,371,136	187	85	35	38	12
XC5VLX50T	14,052,352	281	128	53	57	18
XC5VLX85T	23,341,312	467	213	88	94	29
XC5VLX110T	31,118,848	622	284	118	125	39
XC5VLX155T	43,042,304	861	392	163	174	54
XC5VLX220T	55,133,696	1103	503	209	222	69
XC5VLX330T	82,696,192	1654	754	313	333	103
XC5VSX35T	13,349,120	267	122	51	54	17
XC5VSX50T	20,019,328	400	182	76	81	25
XC5VSX95T	35,716,096	714	326	135	144	45
XC5VSX240T	79,610,368	1592	726	302	321	100
XC5VFX30T	13,517,056	270	123	51	55	17
XC5VFX70T	27,025,408	541	246	102	109	34
XC5VFX100T	39,389,696	788	359	149	159	49
XC5VFX130T	49,234,944	985	449	186	199	62
XC5VFX200T	70,856,704	1417	646	268	286	89
XC5VTX150T	43,278,464	866	394	164	175	54
XC5VTX240T	65,755,648	1315	599	249	265	82
<div> <div></div> Bitstream Transfer Time + FPGA INIT Time (50 ms) ≤ 100 ms. Note: These entries adhere to the configuration time requirement; equation [4] defined in “Board Power in Real-world Systems.” For these calculations, $T_{95\% PWR}$ is negligible or close to zero. <div></div> Bitstream Transfer Time + FPGA INIT Time (50 ms) > 100 ms <div></div> Bitstream Transfer Time > 100 ms </div>						

1. SPI flash assumptions: 50 MHz MAX
2. BPIx16 assumptions: P30 4-page read with 4-cycle first page read (4-1-1-1), maximum configuration time
3. PFP assumptions: 33 MHz MAX
4. XCF128X Slave-SMAPx16 assumptions: CCLK=50 MHz

Sample Problem Analysis

This section presents data from an ASUS PL5 system to demonstrate the relationships between Power Valid, FPGA Configuration, and PERST#. [Figure 7-3](#) shows a case where the endpoint failed to be recognized due to a FPGA configuration time issue. [Figure 7-4](#) shows a successful FPGA configuration with the endpoint being recognized by the system.

Failed FPGA Recognition

[Figure 7-3](#) illustrates a failed cold boot test using the default configuration time on an LX50T FPGA. In this example, the host failed to recognize the Xilinx FPGA. Although a second PERST# pulse assists in allowing more time for the FPGA to configure, the slowness of the FPGA configuration clock (2 MHz) causes configuration to complete well after this second deassertion. During this time, the system enumerated the bus and did not recognize the FPGA.

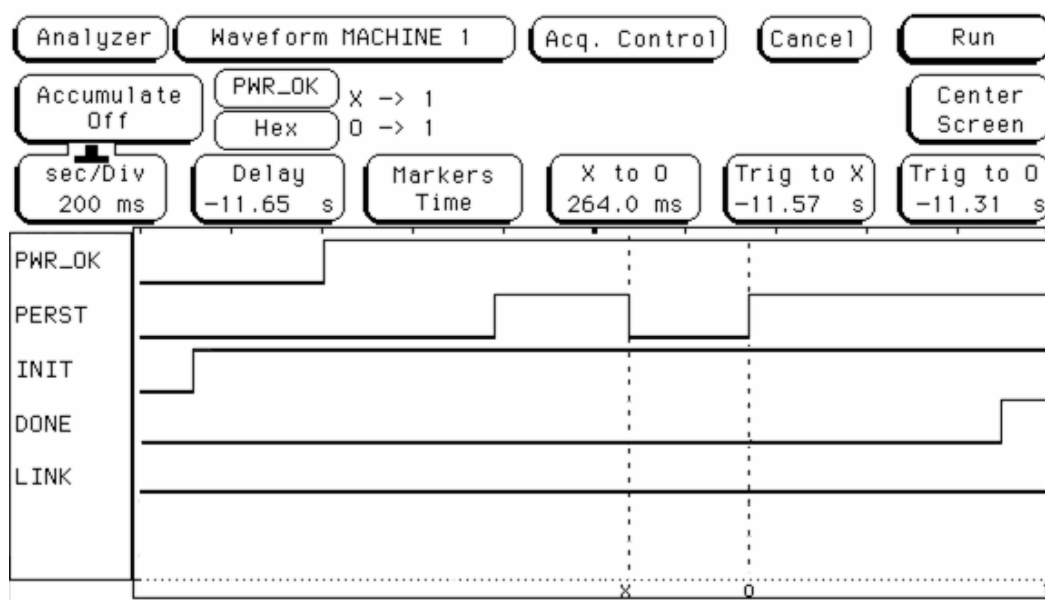


Figure 7-3: Default Configuration Time on LX50T Device (2 MHz Clock)

Successful FPGA Recognition

Figure 7-4 illustrates a successful cold boot test on the same system. In this test, the CCLK was running at 50 MHz, allowing the FPGA to configure in time to be enumerated and recognized. The figure shows that the FPGA began initialization approximately 250 ns before PWR_OK. DONE going high shows that the FPGA was configured even before PWR_OK was asserted.

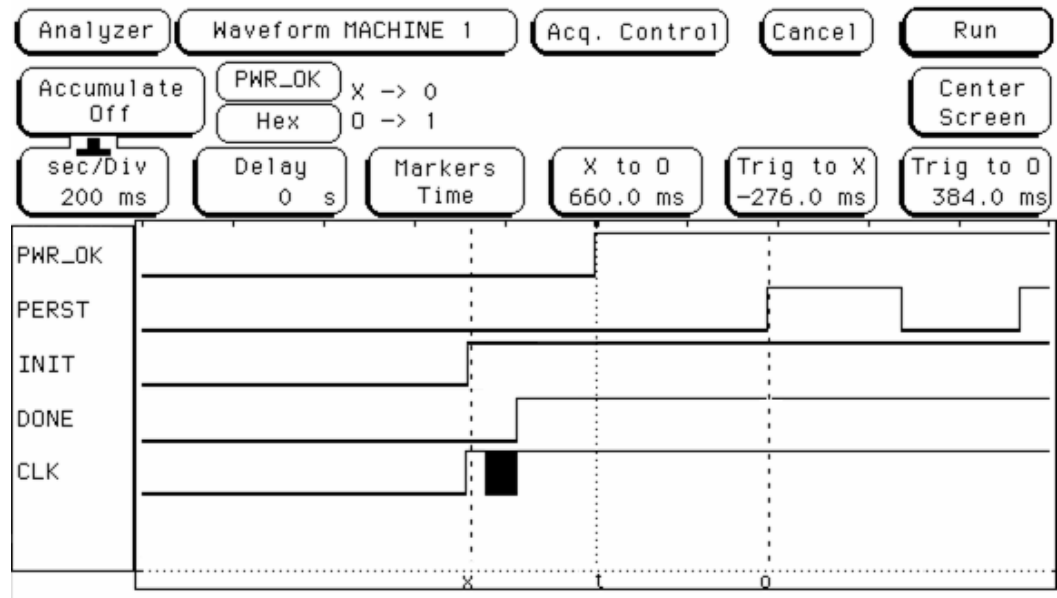


Figure 7-4: Fast Configuration Time on LX50T Device (50 MHz Clock)

Workarounds for Closed Systems

For failing FPGA configuration combinations, as represented by pink cells and yellow cells in Table 7-2 and Table 7-3, designers may be able to work around the problem in closed systems or systems where they can guarantee behavior. These options are not recommended for products where the targeted end system is unknown.

1. Check if the motherboard and BIOS generate multiple PERST# pulses at startup. This can be determined by capturing the signal on the board using an oscilloscope. (This is similar to what is shown in Figure 7-3. If multiple PERST#s are generated, this typically adds extra time for FPGA configuration.

Define $T_{\text{PERSTPERIOD}}$ as the total sum of the pulse width of PERST# and deassertion period before the next PERST# pulse arrives. Because the FPGA is not power cycled or reconfigured with additional PERST# assertions, the $T_{\text{PERSTPERIOD}}$ number can be added to the FPGA configuration equation.

$$\text{FPGA Configuration Time} \leq T_{\text{PWRVLD}} + T_{\text{PERSTPERIOD}} + 100 \text{ ms}$$

2. In closed systems, it may be possible to create scripts to force the system to perform a warm reset after the FPGA is configured, after the initial power up sequence. This resets the system along with the PCI Express sub-system allowing the device to be recognized by the system.

Programmed Input Output Example Design

Programmed Input Output (PIO) transactions are generally used by a PCI Express system host CPU to access Memory Mapped Input Output (MMIO) and Configuration Mapped Input Output (CMIO) locations in the PCI Express fabric. Endpoints for PCI Express accept Memory and IO Write transactions and respond to Memory and IO Read transactions with Completion with Data transactions.

The PIO example design (PIO design) is included with the Endpoint for PCIe generated by the CORE Generator, which allows users to easily bring up their system board with a known established working design to verify the link and functionality of the board.

Note: The PIO design Port Model is shared by the Endpoint for PCI Express, Endpoint Block Plus for PCI Express, and Endpoint PIPE for PCI Express solutions. This appendix represents all the solutions generically using the name Endpoint for PCI Express (or Endpoint for PCIe).

System Overview

The PIO design is a simple target-only application that interfaces with the Endpoint for PCIe core's Transaction (TRN) interface and is provided as a starting point for customers to build their own designs. The following features are included:

- Four transaction-specific 2 kB target regions using the internal Xilinx FPGA block RAMs, providing a total target space of 8192 bytes
- Supports single DWORD payload Read and Write PCI Express transactions to 32/64 bit address memory spaces and IO space with support for completion TLPs
- Utilizes the core's `trn_rbar_hit_n[6:0]` signals to differentiate between TLP destination Base Address Registers
- Provides separate implementations optimized for 32-bit and 64-bit TRN interfaces

Figure A-1 illustrates the PCI Express system architecture components, consisting of a Root Complex, a PCI Express switch device, and an Endpoint for PCIe. PIO operations move data *downstream* from the Root Complex (CPU register) to the Endpoint, and/or *upstream* from the Endpoint to the Root Complex (CPU register). In either case, the PCI Express protocol request to move the data is initiated by the host CPU.

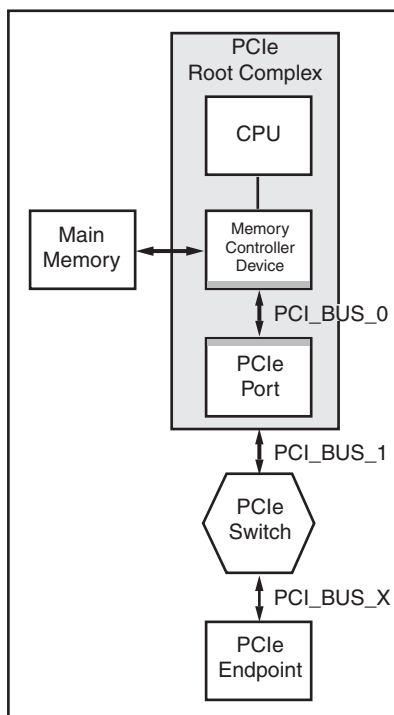


Figure A-1: System Overview

Data is moved downstream when the CPU issues a store register to a MMIO address command. The Root Complex typically generates a Memory Write TLP with the appropriate MMIO location address, byte enables and the register contents. The transaction terminates when the Endpoint receives the Memory Write TLP and updates the corresponding local register.

Data is moved upstream when the CPU issues a load register from a MMIO address command. The Root Complex typically generates a Memory Read TLP with the appropriate MMIO location address and byte enables. The Endpoint generates a Completion with Data TLP once it receives the Memory Read TLP. The Completion is steered to the Root Complex and payload is loaded into the target register, completing the transaction.

PIO Hardware

The PIO design implements a 8192 byte target space in FPGA block RAM, behind the Endpoint for PCIe. This 32-bit target space is accessible through single DWORD IO Read, IO Write, Memory Read 64, Memory Write 64, Memory Read 32, and Memory Write 32 TLPs.

The PIO design generates a completion with 1 DWORD of payload in response to a valid Memory Read 32 TLP, Memory Read 64 TLP, or IO Read TLP request presented to it by the core. In addition, the PIO design returns a completion without data with successful status for IO Write TLP request.

The PIO design processes a Memory or IO Write TLP with 1 DWORD payload by updating the payload into the target address in the FPGA block RAM space.

Base Address Register Support

The PIO design supports four discrete target spaces, each consisting of a 2 kB block of memory represented by a separate Base Address Register (BAR). Using the default parameters, the CORE Generator produces a core configured to work with the PIO design defined in this section, consisting of the following:

- One 64-bit addressable Memory Space BAR
- One 32-bit Addressable Memory Space BAR

Users may change the default parameters used by the PIO design; however, in some cases they may need to change the back-end user application depending on their system. See “[Changing CORE Generator Default BAR Settings](#)” for information about changing the default CORE Generator parameters and the affect on the PIO design.

Each of the four 2 kB address spaces represented by the BARs corresponds to one of four 2 kB address regions in the PIO design. Each 2 kB region is implemented using a 2 kB dual-port block RAM. As transactions are received by the core, the core decodes the address and determines which of the four regions is being targeted. The core presents the TLP to the PIO design and asserts the appropriate bits of `trn_rbar_hit_n[6:0]`, as defined in [Table A-1](#).

Note: Because the endpoint_blk_plus core does not support Expansion ROM TLP accesses, the PIO design’s block RAM dedicated for EROM TLP transactions is disabled.

Table A-1: TLP Traffic Types

Block RAM	TLP Transaction Type	Default BAR	trn_rbar_hit_n[6:0]
ep_mem0	IO TLP transactions	Disabled	Disabled
ep_mem1	32-bit address Memory TLP transactions	2	111_1011b
ep_mem2	64-bit address Memory TLP transactions	0-1	111_1100b
ep_mem3	Disabled	Disabled	Disabled

Changing CORE Generator Default BAR Settings

Users can change the CORE Generator parameters and continue to use the PIO design to create customized Verilog or VHDL source to match the selected BAR settings. However, because the PIO design parameters are more limited than the core parameters, consider the following example design limitations when changing the default CORE Generator parameters:

- The example design supports one IO space BAR, one 32-bit Memory space (that cannot be the Expansion ROM space), and one 64-bit Memory space. If these limits are exceeded, only the first space of a given type will be active—accesses to the other spaces will not result in completions.
- Each space is implemented with a 2 kB memory. If the corresponding BAR is configured to a wider aperture, accesses beyond the 2 kB limit wrap around and overlap the 2 kB memory space.
- The PIO design supports one IO space BAR, which by default is disabled, but can be changed if desired.

Although there are limitations to the PIO design, Verilog or VHDL source code is provided so the user can tailor the example design to their specific needs.

TLP Data Flow

This section defines the data flow of a TLP successfully processed by the PIO design. For detailed information about the interface signals within the sub-blocks of the PIO design, see “Receive Path,” page 126 and “Transmit Path,” page 128.

The PIO design successfully processes single DWORD payload Memory Read and Write TLPs and IO Read and Write TLPs. Memory Read or Memory Write TLPs of lengths larger than one DWORD are not processed correctly by the PIO design; however, the core *does* accept these TLPs and passes them along to the PIO design. If the PIO design receives a TLP with a length of greater than 1 DWORD, the TLP is received completely from the core and discarded. No corresponding completion is generated.

Memory/IO Write TLP Processing

When the Endpoint for PCIe receives a Memory or IO Write TLP, the TLP destination address and transaction type are compared with the values in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive TRN interface of the PIO design. Note that the PIO design handles Memory writes and IO TLP writes in different ways: the PIO design responds to *IO writes* by generating a Completion Without Data (cpl), a requirement of the PCI Express specification.

Along with the start of packet, end of packet, and ready handshaking signals, the Receive TRN interface also asserts the appropriate `trn_rbar_hit_n[6:0]` signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design's RX State Machine processes the incoming Write TLP and extracts the TLP's data and relevant address fields so that it can pass this along to the PIO design's internal block RAM write request controller.

Based on the specific `trn_rbar_hit_n[6:0]` signal asserted, the RX State Machine indicates to the internal write controller the appropriate 2 kB block RAM to use prior to asserting the write enable request. For example, if an IO Write Request is received by the core targeting BAR0, the core passes the TLP to the PIO design and asserts `trn_rbar_hit_n[0]`. The RX State machine extracts the lower address bits and the data field from the IO Write TLP and instructs the internal Memory Write controller to begin a write to the block RAM.

In this example, the assertion of `trn_rbar_hit_n[0]` instructed the PIO memory write controller to access `ep_mem0` (which by default represents 2 kB of IO space). While the write is being carried out to the FPGA block RAM, the PIO design RX state machine deasserts the `trn_rdst_rdy_n` signal, causing the Receive TRN interface to stall receiving any further TLPs until the internal Memory Write controller completes the write to the block RAM. Note that deasserting `trn_rdst_rdy_n` in this way is not required for all designs using the core—the PIO design uses this method to simplify the control logic of the RX state machine.

Memory/IO Read TLP Processing

When the Endpoint for PCIe receives a Memory or IO Read TLP, the TLP destination address and transaction type are compared with the values programmed in the core BARs. If the TLP passes this comparison check, the core passes the TLP to the Receive TRN interface of the PIO design.

Along with the start of packet, end of packet, and ready handshaking signals, the Receive TRN interface also asserts the appropriate `trn_rbar_hit_n[6:0]` signal to indicate to the PIO design the specific destination BAR that matched the incoming TLP. On reception, the PIO design's state machine processes the incoming Read TLP and extracts the relevant TLP information and passes it along to the PIO design's internal block RAM read request controller.

Based on the specific `trn_rbar_hit_n[6:0]` signal asserted, the RX state machine indicates to the internal read request controller the appropriate 2 KB block RAM to use before asserting the read enable request. For example, if a Memory Read 32 Request TLP is received by the core targeting the default MEM32 BAR2, the core passes the TLP to the PIO design and asserts `trn_rbar_hit_n[2]`. The RX State machine extracts the lower address bits from the Memory 32 Read TLP and instructs the internal Memory Read Request controller to start a read operation.

In this example, the assertion of `trn_rbar_hit_n[2]` instructs the PIO memory read controller to access the Mem32 space, which by default represents 2 KB of memory space. A notable difference in handling of memory write and read TLPs is the requirement of the receiving device to return a Completion with Data TLP in the case of memory or IO read request.

While the read is being processed, the PIO design RX state machine deasserts `trn_rdst_rdy_n`, causing the Receive TRN interface to stall receiving any further TLPs until the internal Memory Read controller completes the read access from the block RAM and generates the completion. Note that deasserting `trn_rst_rdy_n` in this way is not required for all designs using the core. The PIO design uses this method to simplify the control logic of the RX state machine.

PIO File Structure

Table A-2 defines the PIO design file structure. Note that based on the specific core targeted, not all files delivered by CORE Generator are necessary, and some files may or may not be delivered. The major difference is that some of the Endpoint for PCIe solutions use a 32-bit user data path, others use a 64-bit data path, and the PIO design works with both. The width of the data path depends on the specific core being targeted.

Table A-2: PIO Design File Structure

File	Description
PIO.[v vhd]	Top-level design wrapper
PIO_EP.[v vhd]	PIO application module
PIO_TO_CTRL.[v vhd]	PIO turn-off controller module
PIO_32.v	32b interface macro define
PIO_64.v	64b macro define
PIO_32_RX_ENGINE.[v vhd]	32b Receive engine
PIO_32_TX_ENGINE.[v vhd]	32b Transmit engine
PIO_64_RX_ENGINE.[v vhd]	64b Receive engine
PIO_64_TX_ENGINE.[v vhd]	64b Transmit engine

Table A-2: PIO Design File Structure (Continued)

File	Description
PIO_EP_MEM_ACCESS.[v vhd]	Endpoint memory access module
EP_MEM.[v vhd]	Endpoint memory

Two configurations of the PIO Design are provided: PIO_32 and PIO_64, with 32 and 64-bit TRN interfaces, respectively. The PIO configuration generated depends on the selected endpoint type (that is, PIPE, PCI Express, and Block Plus) as well as the number of PCI Express lanes selected by the user. Table A-3 identifies the PIO configuration generated based on the user's selection.

Table A-3: PIO Configuration

Core	x1	x4	x8
Endpoint for PIPE	PIO_32	NA	NA
Endpoint for PCI Express	PIO_32	PIO_64	PIO_64
Endpoint for PCI Express Block Plus	PIO_64	PIO_64	PIO_64

Figure A-2 shows the various components of the PIO design, which is separated into four main parts: the TX Engine, RX Engine, Memory Access Controller, and Power Management Turn-Off Controller.

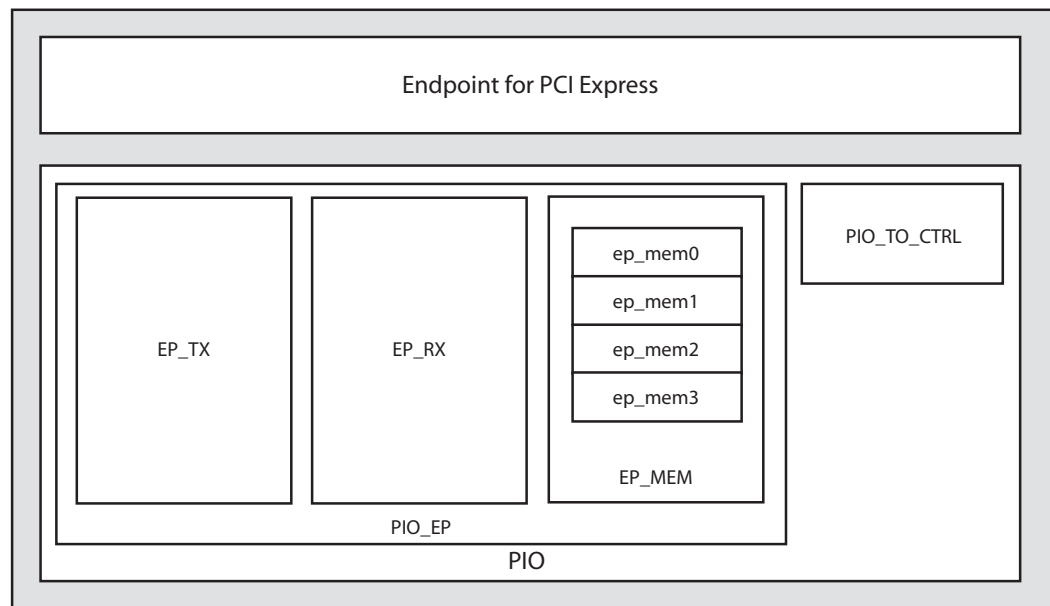


Figure A-2: PIO Design Components

PIO Application

Figures A-3 and A-4 depict 64-bit and 32-bit PIO application top-level connectivity, respectively. The data path width, either 32-bits or 64-bits, depends on which Endpoint for PCIe core is used. The PIO_EP module contains the PIO FPGA block RAM memory modules and the transmit and receive engines. The PIO_TO_CTRL module is the Endpoint Turn-Off controller unit, which responds to power turn-off message from the host CPU with an acknowledgement.

The PIO_EP module connects to the Endpoint Transaction (trn) and Configuration (cfg) interfaces.

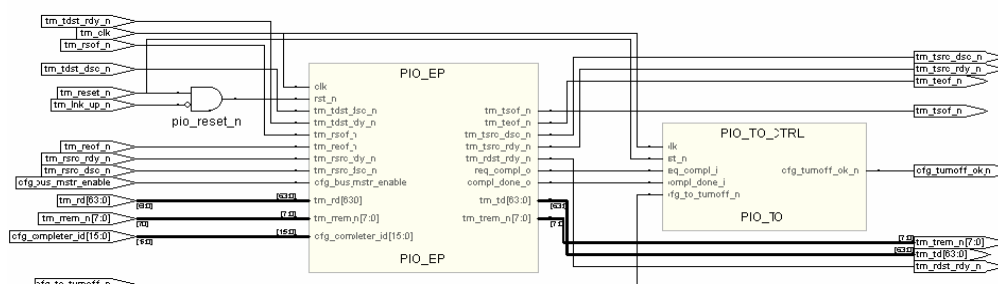


Figure A-3: PIO 64-bit Application

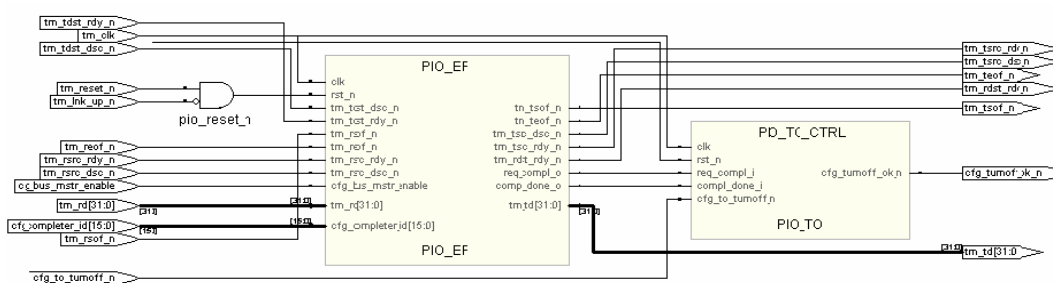


Figure A-4: PIO 32-bit Application

Receive Path

Figure A-5 illustrates the PIO_32_RX_ENGINE and PIO_64_RX_ENGINE modules. The data path of the module must match the data path of the core being used. These modules connect with Endpoint for PCIe Transaction Receive (trn_r*) interface.

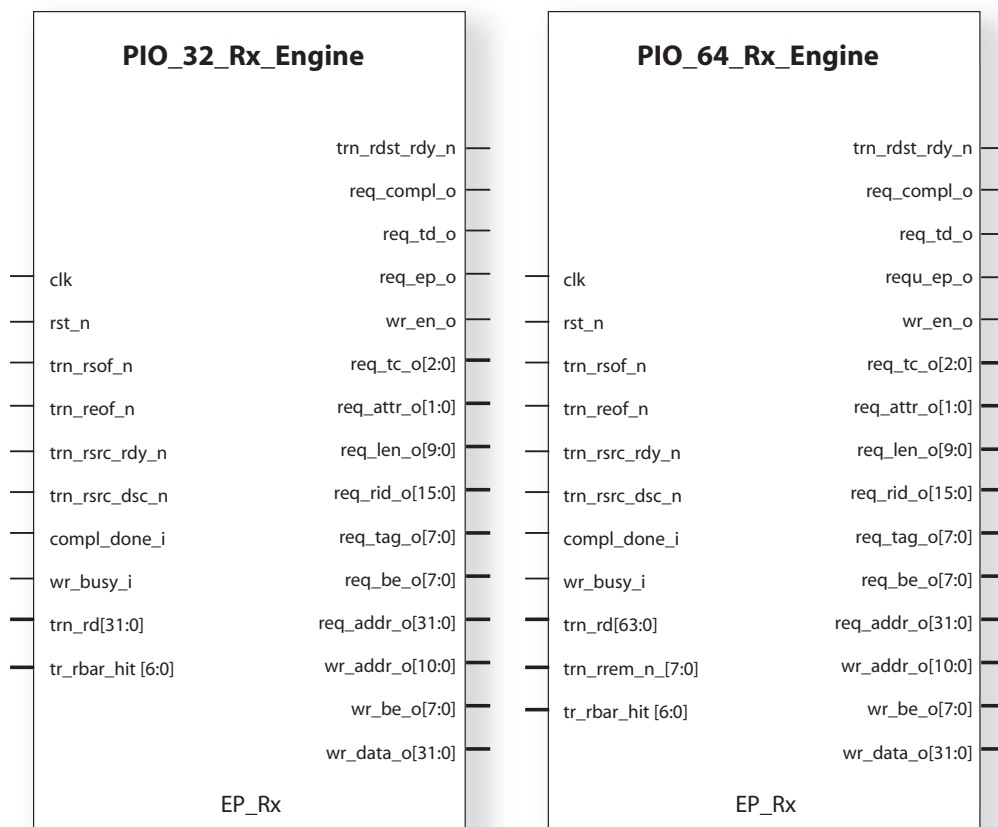


Figure A-5: Rx Engines

The PIO_32_RX_ENGINE and PIO_64_RX_ENGINE modules receive and parse incoming read and write TLPs.

The RX engine parses 1 DWORD 32 and 64-bit addressable memory and IO read requests. The RX state machine extracts needed information from the TLP and passes it to the memory controller, as defined in Table A-4.

Table A-4: Rx Engine: Read Outputs

Port	Description
req_compl_o	Completion request (active high)
req_td_o	Request TLP Digest bit
req_ep_o	Request Error Poisoning bit
req_tc_o[2:0]	Request Traffic Class
req_attr_o[1:0]	Request Attributes

Table A-4: Rx Engine: Read Outputs (Continued)

Port	Description
req_len_o[9:0]	Request Length
req_rid_o[15:0]	Request Requester Identifier
req_tag_o[7:0]	Request Tag
req_be_o[7:0]	Request Byte Enable
req_addr_o[10:0]	Request Address

The RX Engine parses 1 DWORD 32- and 64-bit addressable memory and IO write requests. The RX state machine extracts needed information from the TLP and passes it to the memory controller, as defined in [Table A-5](#).

Table A-5: Rx Engine: Write Outputs

Port	Description
wr_en_o	Write received
wr_addr_o[10:0]	Write address
wr_be_o[7:0]	Write byte enable
wr_data_o[31:0]	Write data

The read data path stops accepting new transactions from the core while the application is processing the current TLP. This is accomplished by `trn_rdst_rdy_n` deassertion. For an ongoing Memory or IO Read transaction, the module waits for `compl_done_i` input to be asserted before it accepts the next TLP, while an ongoing Memory or IO Write transaction is deemed complete after `wr_busy_i` is deasserted.

Transmit Path

Figure A-6 shows the PIO_32_TX_ENGINE and PIO_64_TX_ENGINE modules. The data path of the module must match the data path of the core being used. These modules connect with the core Transaction Transmit (trn_r*) interface.

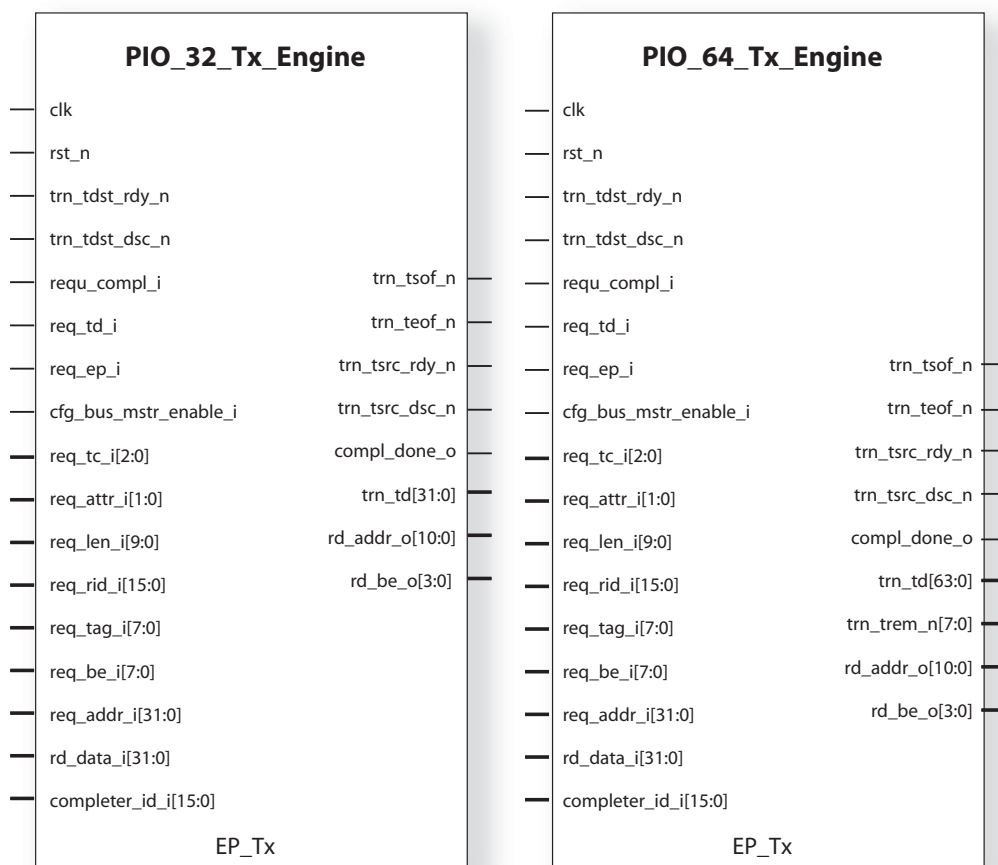


Figure A-6: Tx Engines

The PIO_32_TX_ENGINE and PIO_64_TX_ENGINE modules generate completions for received memory and IO read TLPs. The PIO design does not generate outbound read or write requests. However, users can add this functionality to further customize the design.

The PIO_32_TX_ENGINE and PIO_64_TX_ENGINE modules generate completions in response to 1 DWORD 32 and 64-bit addressable memory and IO read requests. Information necessary to generate the completion is passed to the TX Engine, as defined in Table A-6.

Table A-6: Tx Engine Inputs

Port	Description
req_compl_i	Completion request (active high)
req_td_i	Request TLP Digest bit
req_ep_i	Request Error Poisoning bit

Table A-6: Tx Engine Inputs (Continued)

Port	Description
req_tc_i[2:0]	Request Traffic Class
req_attr_i[1:0]	Request Attributes
req_len_i[9:0]	Request Length
req_rid_i[15:0]	Request Requester Identifier
req_tag_i[7:0]	Request Tag
req_be_i[7:0]	Request Byte Enable
req_addr_i[10:0]	Request Address

After the completion is sent, the TX engine asserts the `compl_done_i` output indicating to the RX engine that it can assert `trn_rdst_rdy_n` and continue receiving TLPs.

Endpoint Memory

Figure A-7 displays the `PIO_EP_MEM_ACCESS` module. This module contains the Endpoint memory space.

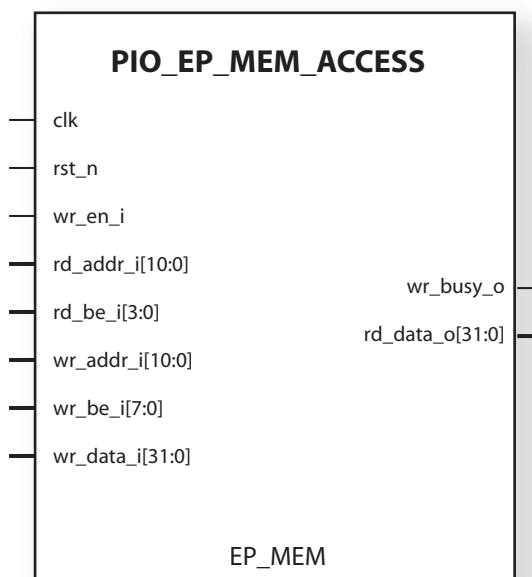


Figure A-7: EP Memory Access

The `PIO_EP_MEM_ACCESS` module processes data written to the memory from incoming Memory and IO Write TLPs and provides data read from the memory in response to Memory and IO Read TLPs.

The `EP_MEM` module processes 1 DWORD 32- and 64-bit addressable Memory and IO Write requests based on the information received from the RX Engine, as defined in

Table A-7. While the memory controller is processing the write, it asserts the `wr_busy_o` output indicating it is busy.

Table A-7: EP Memory: Write Inputs

Port	Description
<code>wr_en_i</code>	Write received
<code>wr_addr_i[10:0]</code>	Write address
<code>wr_be_i[7:0]</code>	Write byte enable
<code>wr_data_i[31:0]</code>	Write data

Both 32 and 64-bit Memory and IO Read requests of one DWORD are processed based on the following inputs, as defined in **Table A-8**. After the read request is processed, the data is returned on `rd_data_o[31:0]`.

Table A-8: EP Memory: Read Inputs

Port	Description
<code>req_be_i[7:0]</code>	Request Byte Enable
<code>req_addr_i[31:0]</code>	Request Address

PIO Operation

PIO Read Transaction

Figure A-8 depicts a Back-To-Back Memory Read request to the PIO design. The receive engine deasserts `trn_rdst_rdy_n` as soon as the first TLP is completely received. The next Read transaction is accepted only after `compl_done_o` is asserted by the transmit engine, indicating that Completion for the first request was successfully transmitted.

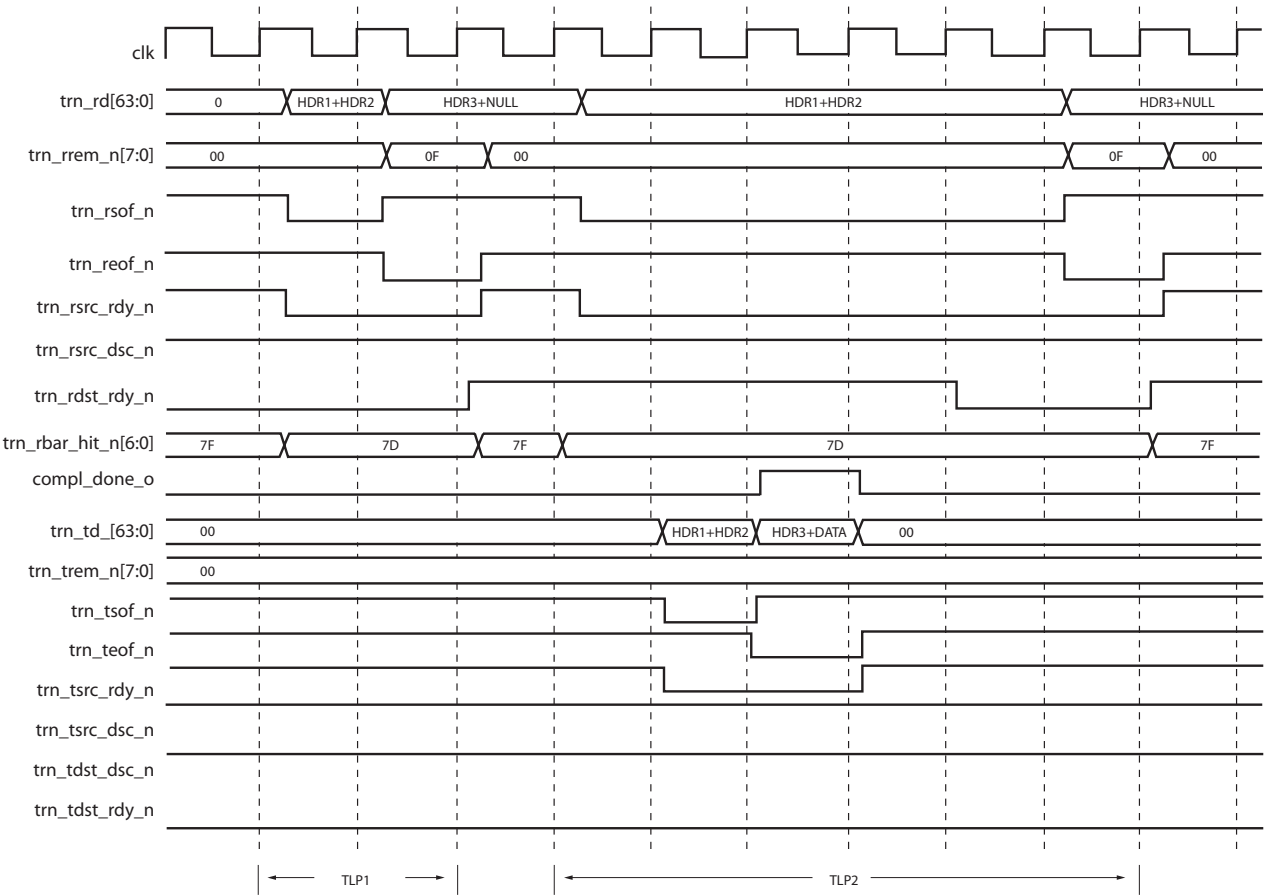


Figure A-8: Back-to-Back Read Transactions

PIO Write Transaction

Figure A-9 depicts a back-to-back Memory Write to the PIO design. The next Write transaction is accepted only after `wr_busy_o` is deasserted by the memory access unit, indicating that data associated with the first request was successfully written to the memory aperture.

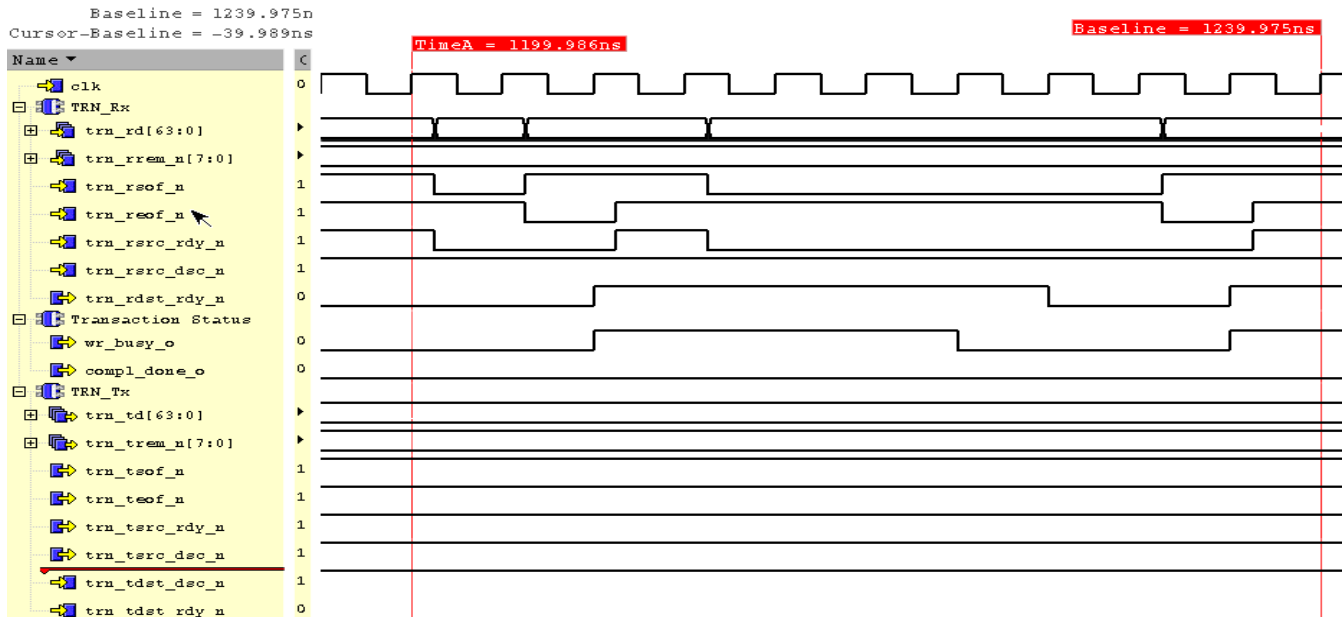


Figure A-9: Back-to-Back Write Transactions

Device Utilization

Table A-9 shows the PIO design FPGA resource utilization.

Table A-9: PIO Design FPGA Resources

Resources	Utilization
LUTs	300
Flip-Flops	500
block RAMs	4

Dual Core Example Design

All Virtex-5 FXT FPGAs support multiple PCI Express Endpoint Blocks. The Virtex-5 FXT FPGA dual core example design is a reference design intended for customers designing with multiple endpoint cores for PCI Express. The dual core example design consists of Verilog and VHDL source code as well as simulation and implementation scripts. For detailed information about the dual core example design, see “Dual Core Example Design,” in “Chapter 3, Quickstart Example Design” of the *Endpoint for PCI Express Block Plus Getting Started Guide* ([GSG343](#)).

Summary

The PIO design demonstrates the Endpoint for PCIe and its interface capabilities. In addition, it enables rapid bring-up and basic validation of end user endpoint add-in card FPGA hardware on PCI Express platforms. Users may leverage standard operating system utilities that enable generation of read and write transactions to the target space in the reference design.

Downstream Port Model Test Bench

The Endpoint for PCI Express Downstream Port Model is a robust test bench environment that provides a test program interface that can be used with the provided PIO design or with your own design. The purpose of the Downstream Port Model is to provide a source mechanism for generating downstream PCI Express TLP traffic to stimulate the customer design, and a destination mechanism for receiving upstream PCI Express TLP traffic from the customer design in a simulation environment.

Note: The Downstream Port Model is shared by the Endpoint for PCI Express, Endpoint Block Plus for PCI Express, and Endpoint PIPE for PCI Express solutions. This appendix represents all the solutions generically using the name Endpoint for PCI Express or Endpoint (or Endpoint for PCIe).

Source code for the Downstream Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the core's configuration space, creating TLP transactions, generating TLP logs, and providing an interface for creating and verifying tests are complete, allowing you to dedicate your efforts to verifying the correct functionality of the design rather than spending time developing an Endpoint core test bench infrastructure.

The Downstream Port Model consists of the following:

- Test Programming Interface (TPI), which allows the user to stimulate the Endpoint device for the PCI Express
- Example tests that illustrate how to use the test program TPI
- Verilog or VHDL source code for all Downstream Port Model components, which allow the user to customize the test bench

Figure B-1 illustrates the illustrates the Downstream Port Model coupled with the PIO design.

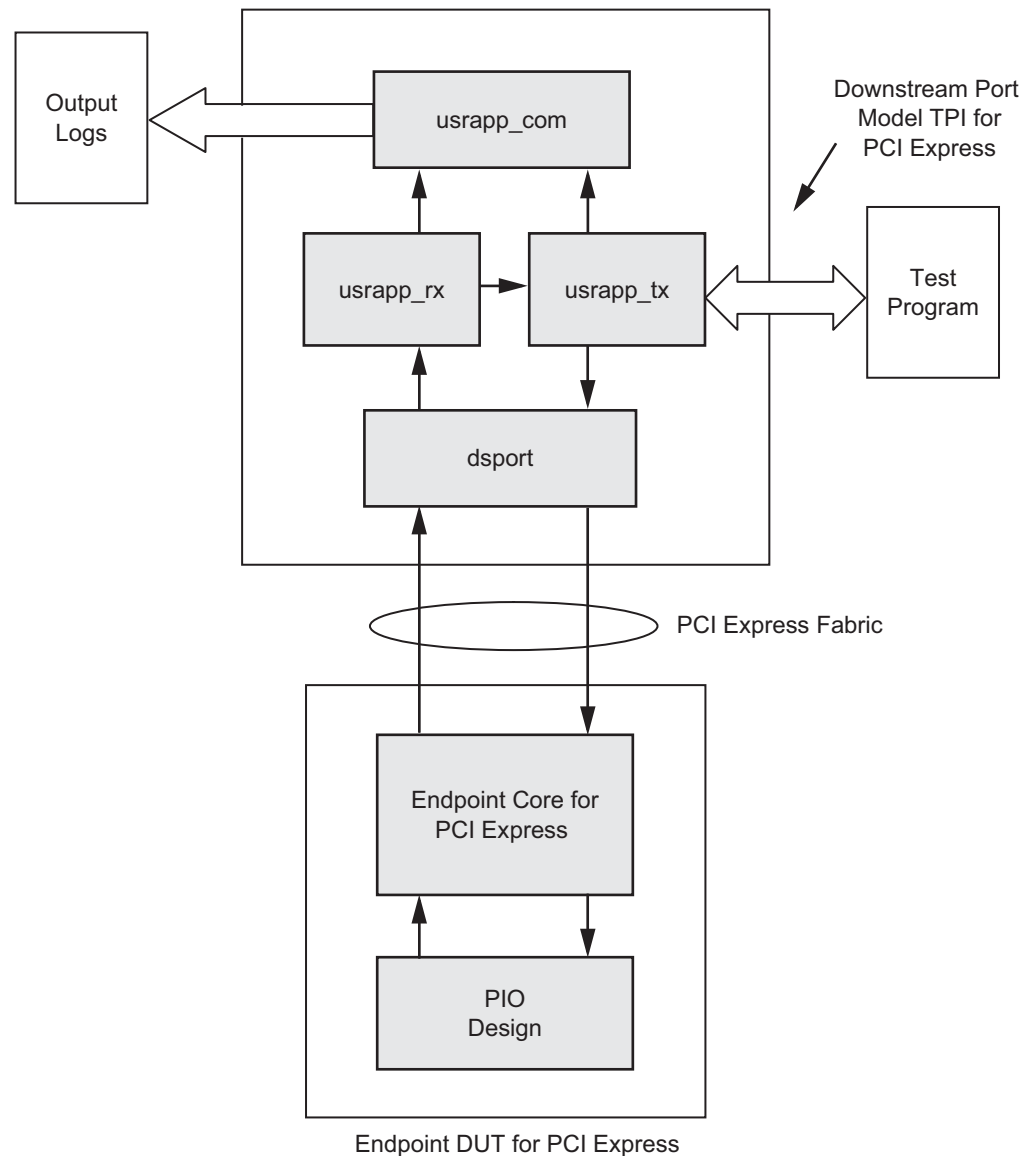


Figure B-1: Downstream Port Model and Top-level Endpoint

Architecture

The Downstream Port Model consists of the following blocks, illustrated in Figure B-1.

- dsport (downstream port)
- usrapp_tx
- usrapp_rx
- usrapp_com (Verilog only)

The `usrapp_tx` and `usrapp_rx` blocks interface with the `dsport` block for transmission and reception of TLPs to/from the Endpoint Design Under Test (DUT). The Endpoint DUT consists of the Endpoint for PCIe and the PIO design (displayed) or customer design.

The `usrapp_tx` block sends TLPs to the `dsport` block for transmission across the PCI Express Link to the Endpoint DUT. In turn, the Endpoint DUT device transmits TLPs across the PCI Express Link to the `dsport` block, which are subsequently passed to the `usrapp_rx` block. The `dsport` and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express fabric. Both the `usrapp_tx` and `usrapp_rx` utilize the `usrapp_com` block for shared functions, for example, TLP processing and log file outputting. Transaction sequences or test programs are initiated by the `usrapp_tx` block to stimulate the endpoint device's fabric interface. TLP responses from the endpoint device are received by the `usrapp_rx` block. Communication between the `usrapp_tx` and `usrapp_rx` blocks allow the `usrapp_tx` block to verify correct behavior and act accordingly when the `usrapp_rx` block has received TLPs from the endpoint device.

The Downstream Port Model has a 128-byte MPS capability in the receive direction and a 512-byte MPS capability in the transmit direction.

Simulating the Design

Three simulation script files are provided with the model to facilitate simulation with Synopsys® VCS, Cadence® IUS, and Mentor Graphics® ModelSim® simulators:

- `simulate_vcs.sh` (Verilog only)
- `simulate_ncsim.sh`
- `simulate_mti.do`

The example simulation script files are located in the following directory:

```
<project_dir>/<component_name>/simulation/functional
```

Instructions for simulating the PIO design using the Downstream Port Model are provided in the *LogiCORE IP Endpoint for PCI Express Getting Started Guide*.

Note: For VHDL users using the IUS flow, the environment variable `LMC_TIMEUNIT` must be set to `-12`, which sets the timing resolution to 1 ps (required for the LogiCORE IP PCI Express simulation SmartModel to operate correctly).

For Linux, use the following command line:

```
> setenv LMC_TIMEUNIT -12
```

Note: For IUS users, the work construct must be manually inserted into your CDS.LIB file:

```
DEFINE WORK WORK
```

Test Selection

Table B-1 describes the tests provided with the Downstream Port Model, followed by specific sections for VHDL and Verilog test selection.

Table B-1: Downstream Port Model Provided Tests

Test Name	Test in VHDL/Verilog	Description
sample_smoke_test0	Verilog and VHDL	Issues a PCI Type 0 Configuration Read TLP and waits for the completion TLP; then compares the value returned with the expected Device/Vendor ID value.
sample_smoke_test1	Verilog	Performs the same operation as sample_smoke_test0 but makes use of expectation tasks. This test uses two separate test program threads: one thread issues the PCI Type 0 Configuration Read TLP and the second thread issues the Completion with Data TLP expectation task. This test illustrates the form for a parallel test that uses expectation tasks. This test form allows for confirming reception of any TLPs from the customer's design. Additionally, this method can be used to confirm reception of TLPs when ordering is unimportant.
pio_writeReadBack_test0	Verilog and VHDL	Transmits a 1 DWORD Write TLP followed by a 1 DWORD Read TLP to each of the example design's active BARs, and then waits for the Completion TLP and verifies that the write and read data match. The test will send the appropriate TLP to each BAR based on the BARs address type (for example, 32 bit or 64 bit) and space type (for example, IO or Memory).
pio_testByteEnables_test0	Verilog	Issues four sequential Write TLPs enabling a unique byte enable for each Write TLP, and then issues a 1 DWORD Read TLP to confirm that the data was correctly written to the example design. The test will send the appropriate TLP to each BAR based on the BARs address-type (for example, 32 bit or 64 bit) and space type (for example, IO or Memory).
pio_memTestDataBus	Verilog	Determines if the PIO design's FPGA block RAMs data bus interface is correctly connected by performing a 32-bit walking ones data test to the first available BAR in the example design.
pio_memTestAddrBus	Verilog	Determines whether the PIO design's FPGA block RAM's address bus interface is correctly connected by performing a walking ones address test. This test should only be called after successful completion of pio_memTestDataBus.
pio_memTestDevice	Verilog	Checks the integrity of each bit of the PIO design's FPGA block RAM by performing an increment/decrement test. This test should only be called after successful completion of pio_memTestAddrBus.

Table B-1: Downstream Port Model Provided Tests (Continued)

pio_timeoutFailureExpected	Verilog	Sends a Memory 32 Write TLP followed by Memory 32 Read TLP to an invalid address and waits for a Completion with data TLP. This test anticipates that waiting for the completion TLP times out and illustrates how the test programmer can gracefully handle this event.
pio_tlp_test0 (illustrative example only)	Verilog	Issues a sequence of Read and Write TLPs to the example design's RX interface. Some of the TLPs, for example, burst writes, are not supported by the PIO design.

VHDL Test Selection

Test selection is implemented in the VHDL Downstream Port Model by means of overriding the `test_selector` generic within the `tests` entity. The `test_selector` generic is a string with a one-to-one correspondence to each test within the `tests` entity.

The user can modify the generic mapping of the instantiation of the `tests` entity within the `pci_exp_usrapp_tx` entity. The default generic mapping is to override the `test_selector` with the test name `pio_writeReadBack_test0`. Currently there are two tests defined inside the `tests` entity, `sample_smoke_test0` and `pio_writeReadBack_test0`. Additional customer-defined tests should be added inside `tests.vhd`. Currently, specific tests cannot be selected from the VHDL simulation scripts.

Verilog Test Selection

The Verilog test model used for the Downstream Port Model lets you specify the name of the test to be run as a command line parameter to the simulator. For example, the `simulate_ncsim.sh` script file, used to start the IUS simulator explicitly specifies the test `sample_smoke_test0` to be run using the following command line syntax:

```
ncsim work.boardx01 +TESTNAME=sample_smoke_test0
```

You can change the test to be run by changing the value provided to `TESTNAME` defined in the test files `sample_tests1.v` and `pio_tests.v`. The same mechanism is used for VCS and ModelSim.

VHDL and Verilog Downstream Port Model Differences

The following sections identify differences between the VHDL and Verilog Downstream Port Model.

Verilog Expectation Tasks

The most significant difference between the Verilog and the VHDL test bench is that the Verilog test bench has Expectation Tasks. Expectation tasks are API calls used in conjunction with a bus mastering customer design. The test program issues a series of expectation task calls, that is, the task calls expect a memory write tlp and a memory read tlp. If the customer design does not respond with the expected tlps, the test program fails. This functionality was implemented using the fork-join construct in Verilog, which is not available in VHDL and subsequently not implemented.

Verilog Command Line versus VHDL tests.vhd Module

The Verilog test bench allows test programs to be specified at the command line, while the VHDL test bench specifies test programs within the `tests.vhd` module.

Generating Wave Files

- The Verilog test bench uses `recordvars` and `dumpfile` commands within the code to generate wave files.
- The VHDL test bench leaves the generating wave file functionality up to the simulator.

Speed Differences

The VHDL test bench is slower than the Verilog test bench, especially when testing the x8 core. For initial design simulation and speed enhancement, you may want to use the x1 core, identify basic functionality issues, and then move to x4 or x8 simulation when testing design performance.

Waveform Dumping

Table B-2 describes the available simulator waveform dump file formats, each of which is provided in the simulator's native file format. The same mechanism is used for VCS and ModelSim.

Table B-2: Simulator Dump File Format

Simulator	Dump File Format
Synopsys VCS	.vpd
ModelSim	.vcd
Cadence IUS	.trn

VHDL Flow

Waveform dumping in the VHDL flow does not use the `+dump_all` mechanism described in the Verilog flow section. Because the VHDL language itself does not provide a common interface for dumping waveforms, each VHDL simulator has its own interface for supporting waveform dumping. For both the supported ModelSim and IUS flows, dumping is supported by invoking the VHDL simulator command line with a command line option that specifies the respective waveform command file, `wave.do` (ModelSim) and `wave.sv` (IUS). This command line can be found in the respective simulation script files `simulate_mti.do` and `simulate_ncsim.sh`.

ModelSim

The following command line initiates waveform dumping for the ModelSim flow using the VHDL test bench:

```
>vsim +notimingchecks -do wave.do -L unisim -L work work.board
```

IUS

The following command line initiates waveform dumping for the IUS flow using the VHDL test bench:

```
>ncsim -gui work.board -input @"simvision -input wave.sv"
```

Verilog Flow

The Downstream Port Model provides a mechanism for outputting the simulation waveform to file by specifying the `+dump_all` command line parameter to the simulator.

For example, the script file `simulate_ncsim.sh` (used to start the IUS simulator) can indicate to the Downstream Port Model that the waveform should be saved to a file using the following command line:

```
ncsim work.boardx01 +TESTNAME=sample_smoke_test0 +dump_all
```

Output Logging

When a test fails on the example or customer design, the test programmer debugs the offending test case. Typically, the test programmer inspects the wave file for the simulation and cross-reference this to the messages displayed on the standard output. Because this approach can be very time consuming, the Downstream Port Model offers an output logging mechanism to assist the tester with debugging failing test cases to speed the process.

The Downstream Port Model creates three output files (`tx.dat`, `rx.dat`, and `error.dat`) during each simulation run. Log files `rx.dat` and `tx.dat` each contain a detailed record of every TLP that was received and transmitted, respectively, by the Downstream Port Model. With an understanding of the expected TLP transmission during a specific test case, the test programmer may more easily isolate the failure.

The log file `error.dat` is used in conjunction with the expectation tasks. Test programs that utilize the expectation tasks will generate a general error message to standard output. Detailed information about the specific comparison failures that have occurred due to the expectation error is located within `error.dat`.

Parallel Test Programs

There are two classes of tests supported by the Downstream Port Model:

- **Sequential tests.** Tests that exist within one process and behave similarly to sequential programs. The test depicted in "[Test Program: pio_writeReadBack_test0](#)" is an example of a sequential test. Sequential tests are very useful when verifying behavior that have events with a known order.
- **Parallel tests.** Tests involving more than one process thread. The test `sample_smoke_test1` is an example of a parallel test with two process threads. Parallel tests are very useful when verifying that a specific set of events have occurred, however the order of these events are not known.

A typical parallel test uses the form of one command thread and one or more expectation threads. These threads work together to verify a device's functionality. The role of the command thread is to create the necessary TLP transactions that cause the device to receive and generate TLPs. The role of the expectation threads is to verify the reception of an expected TLP. The Downstream Port Model TPI has a complete set of expectation tasks to be used in conjunction with parallel tests.

Note that because the example design is a target-only device, only Completion TLPs can be expected by parallel test programs while using the PIO design. However, the full library of

expectation tasks can be used for expecting any TLP type when used in conjunction with the customer's design (which may include bus-mastering functionality). Currently the VHDL version of the Downstream Port Model Test Bench does not support Parallel tests.

Test Description

The Downstream Port Model provides a Test Program Interface (TPI). The TPI provides the means to create tests by simply invoking a series of Verilog tasks. All Downstream Port Model tests should follow the same six steps:

1. Perform conditional comparison of a unique test name
2. Set up master timeout in case simulation hangs
3. Wait for Reset and link-up
4. Initialize the configuration space of the endpoint
5. Transmit and receive TLPs between the Downstream Port Model and the Endpoint DUT
6. Verify that the test succeeded

“Test Program: pio_writeReadBack_test0” displays the listing of a simple test program *pio_writeReadBack_test0*, written for use in conjunction with the PIO endpoint. This test program is located in the file *pio_tests.v*. As the test name implies, this test performs a one DWORD write operation to the PIO Design followed by a 1 DWORD read operation from the PIO Design, after which it compares the values to confirm that they are equal. The test is performed on the first location in each of the active Mem32 BARs of the PIO Design. For the default configuration, this test performs the write and read back to BAR2 and to the EROM space (BAR6) (Block Plus only). The following section outlines the steps performed by the test program.

- Line 1 of the sample program determines if the user has selected the test program *pio_writeReadBack_test1* when invoking the Verilog simulator.
- Line 4 of the sample program invokes the TPI call *TSK_SIMULATION_TIMEOUT* which sets the master timeout value to be long enough for the test to complete.
- Line 5 of the sample program invokes the TPI call *TSK_SYSTEM_INITIALIZATION*. This task will cause the test program to wait for the system reset to deassert as well as the endpoint's *trn_lnk_up_n* signal to assert. This is an indication that the endpoint is ready to be configured by the test program via the Downstream Port Model.
- Line 6 of the sample program uses the TPI call *TSK_BAR_INIT*. This task will perform a series of Type 0 Configuration Writes and Reads to the Endpoint core's PCI Configuration Space, determine the memory and IO requirements of the endpoint, and then program the endpoint's Base Address Registers so that it is ready to receive TLPs from the Downstream Port Model.
- Lines 7, 8, and 9 of the sample program work together to cycle through all the endpoint's BARs and determine whether they are enabled, and if so to determine their type, for example, Mem32, Mem64, or IO).

Note that all PIO tests provided with the Downstream Port Model are written in a form that does not assume that a specific BAR is enabled or is of a specific type (for example, Mem32, Mem64, IO). These tests perform on-the-fly BAR determination and execute TLP transactions dependent on BAR types (that is, Memory32 TLPs to Memory32 Space, IO TLPs to IO Space, and so forth). This means that if a user reconfigures the BARs of the Endpoint, the PIO continues to work because it dynamically explores and configures the

BARs. Users are not required to follow the form used and can create tests that assume their own specific BAR configuration.

- Line 7 sets a counter to increment through all of the endpoint's BARs.
- Line 8 determines whether the BAR is enabled by checking the global array `BAR_INIT_P_BAR_ENABLED[]`. A non-zero value indicates that the corresponding BAR is enabled. If the BAR is not enabled then test program flow will move on to check the next BAR. The previous call to `TSK_BAR_INIT` performed the necessary configuration TLP communication to the endpoint device and filled in the appropriate values into the `BAR_INIT_P_BAR_ENABLED[]` array.
- Line 9 performs a case statement on the same global array `BAR_INIT_P_BAR_ENABLED[]`. If the array element is enabled (that is, non-zero), the element's value indicates the BAR type. A value of 1, 2, and 3 indicates IO, Memory 32, and Memory 64 spaces, respectively.

If the BAR type is either IO or Memory 64, then the test does not perform any TLP transactions. If the BAR type is Memory 32, program control continues to line 16 and starts transmitting Memory 32 TLPs.

- Lines 21-26 use the TPI call `TSK_TX_MEMORY_WRITE_32` and transmits a Memory 32 Write TLP with the payload DWORD '01020304' to the PIO endpoint.
- Lines 32-33 use the TPI calls `TSK_TX_MEMORY_READ_32` followed by `TSK_WAIT_FOR_READ_DATA` in order to transmit a Memory 32 Read TLP and then wait for the next Memory 32 Completion with Data TLP. In case the Downstream Port Model never receives the Completion with Data TLP, the TPI call `TSK_WAIT_FOR_READ_DATA` would locally timeout and display an error message.
- Line 34 compares the DWORD received from the Completion with Data TLP with the DWORD that was transmitted to the PIO endpoint and displays the appropriate success or failure message.

Test Program: pio_writeReadBack_test0

```

1.  else if(testname == "pio_writeReadBack_test1"
2.  begin
3.  // This test performs a 32 bit write to a 32 bit Memory space and performs a read back
4.  TSK_SIMULATION_TIMEOUT(10050);
5.  TSK_SYSTEM_INITIALIZATION;
6.  TSK_BAR_INIT;
7.  for (ii = 0; ii <= 6; ii = ii + 1) begin
8.  if (BAR_INIT_P_BAR_ENABLED[ii] > 2'b00) // bar is enabled
9.  case(BAR_INIT_P_BAR_ENABLED[ii])
10. 2'b01 : // IO SPACE
11.  begin
12.  $display("[%t] : NOTHING: to IO 32 Space BAR %x", $realtime, ii);
13.  end
14. 2'b10 : // MEM 32 SPACE
15.  begin
16.  $display("[%t] : Transmitting TLPs to Memory 32 Space BAR %x",
17.  $realtime, ii);
18.  //-----
19.  // Event : Memory Write 32 bit TLP
20.  //-----
21.  DATA_STORE[0] = 8'h04;
22.  DATA_STORE[1] = 8'h03;
23.  DATA_STORE[2] = 8'h02;
24.  DATA_STORE[3] = 8'h01;
25.  P_READ_DATA = 32'hffff_ffff; // make sure P_READ_DATA has known initial value
26.  TSK_TX_MEMORY_WRITE_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0] , 4'hF,
4'hF, 1'b0);
27.  TSK_TX_CLK_EAT(10);
28.  DEFAULT_TAG = DEFAULT_TAG + 1;
29.  //-----
30.  // Event : Memory Read 32 bit TLP
31.  //-----
32.  TSK_TX_MEMORY_READ_32(DEFAULT_TAG, DEFAULT_TC, 10'd1, BAR_INIT_P_BAR[ii][31:0], 4'hF,
4'hF);
33.  TSK_WAIT_FOR_READ_DATA;
34.  if (P_READ_DATA != {DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0] })
35.  begin
36.  $display("[%t] : Test FAILED --- Data Error Mismatch, Write Data %x != Read Data %x",
$realtime,{DATA_STORE[3], DATA_STORE[2], DATA_STORE[1], DATA_STORE[0]}, P_READ_DATA);
37.  end
38.  else
39.  begin
40.  $display("[%t] : Test PASSED --- Write Data: %x successfully received", $realtime,
P_READ_DATA);
41.  end
42.  TSK_TX_CLK_EAT(10);
43.  DEFAULT_TAG = DEFAULT_TAG + 1;
44.  end
45. 2'b11 : // MEM 64 SPACE
46.  begin
47.  $display("[%t] : NOTHING: to Memory 64 Space BAR %x", $realtime, ii);
48.  end
49.  default : $display("Error case in usrapp_tx\n");
50.  endcase
51.  end
52.  $display("[%t] : Finished transmission of PCI-Express TLPs", $realtime);
53.  $finish;
54.  end

```


Expanding the Downstream Port Model

The Downstream Port Model was created to work with the PIO design, and for this reason is tailored to make specific checks and warnings based on the limitations of the PIO design. These checks and warnings are enabled by default when the Downstream Port Model is generated by the CORE Generator. However, these limitations can easily be disabled so that they do not affect the customer's design.

Because the PIO design was created to support at most one IO BAR, one Mem64 BAR, and two Mem32 BARs (one of which must be the EROM space), the Downstream Port Model by default makes a check during device configuration that verifies that the core has been configured to meet this requirement. A violation of this check will cause a warning message to be displayed as well as for the offending BAR to be gracefully disabled in the test bench. This check can be disabled by setting the `pio_check_design` variable to zero in the `pci_exp_usrapp_tx.v` file.

Downstream Port Model TPI Task List

The Downstream Port Model TPI tasks include the following, which are further defined in [Tables B-3 through B-7](#).

- “Test Setup Tasks”
- “TLP Tasks”
- “BAR Initialization Tasks”
- “Example PIO Design Tasks”
- “Expectation Tasks”

Table B-3: Test Setup Tasks

Name	Input(s)		Description
TSK_SYSTEM_INITIALIZATION	None		Waits for transaction interface reset and link-up between the Downstream Port Model and the Endpoint DUT. This task must be invoked prior to the Endpoint core initialization.
TSK_USR_DATA_SETUP_SEQ	None		Initializes global 4096 byte DATA_STORE array entries to sequential values from zero to 4095.
TSK_TX_CLK_EAT	clock count	31:30	Waits clock_count transaction interface clocks.
TSK_SIMULATION_TIMEOUT	timeout	31:0	Sets master simulation timeout value in units of transaction interface clocks. This task should be used to ensure that all DUT tests complete.

Table B-4: TLP Tasks

Name	Input(s)	Description
TSK_TX_TYPE0_CONFIGURATION_READ	tag_ 7:0 reg_addr_ 11:0 first_dw_be_ 3:0	<p>Waits for transaction interface reset and link-up between the Downstream Port Model and the Endpoint DUT.</p> <p>This task must be invoked prior to Endpoint core initialization.</p>
TSK_TX_TYPE1_CONFIGURATION_READ	tag_ 7:0 reg_addr_ 11:0 first_dw_be_ 3:0	<p>Sends a Type 1 PCI Express Config Read TLP from Downstream Port Model to reg_addr_ of Endpoint DUT with tag_ and first_dw_be_ inputs.</p> <p>CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_TYPE0_CONFIGURATION_WRITE	tag_ 7:0 reg_addr_ 11:0 reg_data_ 31:0 first_dw_be_ 3:0	<p>Sends a Type 0 PCI Express Config Write TLP from Downstream Port Model to reg_addr_ of Endpoint DUT with tag_ and first_dw_be_ inputs.</p> <p>Cpl returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_TYPE1_CONFIGURATION_WRITE	tag_ 7:0 reg_addr_ 11:0 reg_data_ 31:0 first_dw_be_ 3:0	<p>Sends a Type 1 PCI Express Config Write TLP from Downstream Port Model to reg_addr_ of Endpoint DUT with tag_ and first_dw_be_ inputs.</p> <p>Cpl returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_MEMORY_READ_32	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 31:0 last_dw_be_ 3:0 first_dw_be_ 3:0	<p>Sends a PCI Express Memory Read TLP from downstream port to 32 bit memory address addr_ of Endpoint DUT.</p> <p>CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>
TSK_TX_MEMORY_READ_64	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 63:0 last_dw_be_ 3:0 first_dw_be_ 3:0	<p>Sends a PCI Express Memory Read TLP from Downstream Port Model to 64 bit memory address addr_ of Endpoint DUT.</p> <p>CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p>

Table B-4: TLP Tasks (Continued)

Name	Input(s)	Description
TSK_TX_MEMORY_WRITE_32	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 31:0 last_dw_be_ 3:0 first_dw_be_ 3:0 ep_ –	Sends a PCI Express Memory Write TLP from Downstream Port Model to 32 bit memory address addr_ of Endpoint DUT. CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_MEMORY_WRITE_64	tag_ 7:0 tc_ 2:0 len_ 9:0 addr_ 63:0 last_dw_be_ 3:0 first_dw_be_ 3:0 ep_ –	Sends a PCI Express Memory Write TLP from Downstream Port Model to 64 bit memory address addr_ of Endpoint DUT. CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID. The global DATA_STORE byte array is used to pass write data to task.
TSK_TX_COMPLETION	tag_ 7:0 tc_ 2:0 len_ 9:0 comp_status_ 2:0	Sends a PCI Express Completion TLP from Downstream Port Model to Endpoint DUT using global COMPLETE_ID_CFG as completion ID.
TSK_TX_COMPLETION_DATA	tag_ 7:0 tc_ 2:0 len_ 9:0 byte_count 11:0 lower_addr 6:0 comp_status 2:0 ep_ –	Sends a PCI Express Completion with Data TLP from Downstream Port Model to Endpoint DUT using global COMPLETE_ID_CFG as completion ID. The global DATA_STORE byte array is used to pass completion data to task.
TSK_TX_MESSAGE	tag_ 7:0 tc_ 2:0 len_ 9:0 data 63:0 message_rtg 2:0 message_code 7:0	Sends a PCI Express Message TLP from Downstream Port Model to Endpoint DUT. Completion returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_MESSAGE_DATA	tag_ 7:0 tc_ 2:0 len_ 9:0 data 63:0 message_rtg 2:0 message_code 7:0	Sends a PCI Express Message with Data TLP from Downstream Port Model to Endpoint DUT. The global DATA_STORE byte array is used to pass message data to task. Completion returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.

Table B-4: TLP Tasks (Continued)

Name	Input(s)	Description
TSK_TX_IO_READ	tag_ 7:0 addr_ 31:0 first_dw_be_ 3:0	Sends a PCI Express IO Read TLP from Downstream Port Model to IO address addr_[31:2] of Endpoint DUT. CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_IO_WRITE	tag_ 7:0 addr_ 31:0 first_dw_be_ 3:0 data 31:0	Sends a PCI Express IO Write TLP from Downstream Port Model to IO address addr_[31:2] of Endpoint DUT. CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.
TSK_TX_BAR_READ	bar_index 2:0 byte_offset 31:0 tag_ 7:0 tc_ 2:0	Sends a PCI Express 1 DWORD Memory 32, Memory 64, or IO Read TLP from the Downstream Port Model to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT. This task sends the appropriate Read TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed. CplID returned from Endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.

Table B-4: TLP Tasks (Continued)

Name	Input(s)	Description
TSK_TX_BAR_WRITE	bar_index 2:0 byte_offset 31:0 tag_ 7:0 tc_ 2:0 data_ 31:0	<p>Sends a PCI Express 1 DWORD Memory 32, Memory 64, or IO Write TLP from the Downstream Port to the target address corresponding to offset byte_offset from BAR bar_index of the Endpoint DUT.</p> <p>This task sends the appropriate Write TLP based on how BAR bar_index has been configured during initialization. This task can only be called after TSK_BAR_INIT has successfully completed.</p>
TSK_WAIT_FOR_READ_DATA	None	<p>Waits for the next completion with data TLP that was sent by the Endpoint DUT. On successful completion, the first DWORD of data from the CplD will be stored in the global P_READ_DATA. This task should be called immediately following any of the read tasks in the TPI that request Completion with Data TLPs to avoid any race conditions.</p> <p>By default this task will locally time out and terminate the simulation after 1000 transaction interface clocks. The global cpld_to_finish can be set to zero so that local time out returns execution to the calling test and does not result in simulation timeout. For this case test programs should check the global cpld_to, which when set to one indicates that this task has timed out and that the contents of P_READ_DATA are invalid.</p>

Table B-5: BAR Initialization Tasks

Name	Input(s)	Description
TSK_BAR_INIT	None	<p>Performs a standard sequence of Base Address Register initialization tasks to the Endpoint device using the PCI Express fabric. Performs a scan of the Endpoint's PCI BAR range requirements, performs the necessary memory and IO space mapping calculations, and finally programs the Endpoint so that it is ready to be accessed.</p> <p>On completion, the user test program may begin memory and IO transactions to the device. This function displays to standard output a memory/IO table that details how the Endpoint has been initialized. This task also initializes global variables within the Downstream Port Model that are available for test program usage. This task should only be called after TSK_SYSTEM_INITIALIZATION.</p>
TSK_BAR_SCAN	None	<p>Performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads using the PCI Express fabric in order to determine the memory and IO requirements for the Endpoint.</p> <p>The task stores this information in the global array BAR_INIT_P_BAR_RANGE[]. This task should only be called after TSK_SYSTEM_INITIALIZATION.</p>
TSK_BUILD_PCIE_MAP	None	<p>Performs memory/IO mapping algorithm and allocates Memory 32, Memory 64, and IO space based on the Endpoint requirements.</p> <p>This task has been customized to work in conjunction with the limitations of the PIO design and should only be called after completion of TSK_BAR_SCAN.</p>
TSK_DISPLAY_PCIE_MAP	None	<p>Displays the memory mapping information of the Endpoint core's PCI Base Address Registers. For each BAR, the BAR value, the BAR range, and BAR type is given. This task should only be called after completion of TSK_BUILD_PCIE_MAP.</p>

Table B-6: Example PIO Design Tasks

Name	Input(s)		Description
TSK_TX_READBACK_CONFIG	None		Performs a sequence of PCI Type 0 Configuration Reads to the Endpoint device's Base Address Registers, PCI Command Register, and PCIe Device Control Register using the PCI Express fabric. This task should only be called after TSK_SYSTEM_INITIALIZATION.
TSK_MEM_TEST_DATA_BUS	bar_index	2:0	Tests whether the PIO design FPGA block RAM data bus interface is correctly connected by performing a 32-bit walking ones data test to the IO or memory address pointed to by the input bar_index. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.
TSK_MEM_TEST_ADDR_BUS	bar_index nBytes	2:0 31:0	Tests whether the PIO design FPGA block RAM address bus interface is accurately connected by performing a walking ones address test starting at the IO or memory address pointed to by the input bar_index. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.
TSK_MEM_TEST_DEVICE	bar_index nBytes	2:0 31:0	Tests the integrity of each bit of the PIO design FPGA block RAM by performing an increment/decrement test on all bits starting at the block RAM pointed to by the input bar_index with the range specified by input nBytes. For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design. Additionally, the nBytes input should specify the entire size of the individual block RAM.

Table B-7: Expectation Tasks

Name	Input(s)	Output	Description
TSK_EXPECT_CPLD	traffic_class 2:0 td - ep - attr 1:0 length 9:0 completer_id 15:0 completer_status 2:0 bcm - byte_count 11:0 requester_id 15:0 tag 7:0 address_low 6:0	expect status	Waits for a Completion with Data TLP that matches traffic_class, td, ep, attr, length, and payload. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_CPL	traffic_class 2:0 td - ep - attr 1:0 completer_id 15:0 completer_status 2:0 bcm - byte_count 11:0 requester_id 15:0 tag 7:0 address_low 6:0	Expect status	Waits for a Completion without Data TLP that matches traffic_class, td, ep, attr, and length. Returns a 1 on successful completion; 0 otherwise.
TSK_EXPECT_MEMRD	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 29:0	Expect status	Waits for a 32-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. Note that this task can only be used in conjunction with Bus Master designs.

Table B-7: Expectation Tasks (Continued)

Name	Input(s)	Output	Description
TSK_EXPECT_MEMRD64	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 61:0	Expect status	Waits for a 64-bit Address Memory Read TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. Note that this task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 29:0	Expect status	Waits for a 32 bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. Note that this task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_MEMWR64	traffic_class 2:0 td - ep - attr 1:0 length 9:0 requester_id 15:0 tag 7:0 last_dw_be 3:0 first_dw_be 3:0 address 61:0	Expect status	Waits for a 64 bit Address Memory Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. Note that this task can only be used in conjunction with Bus Master designs.
TSK_EXPECT_IOWR	td - ep - requester_id 15:0 tag 7:0 first_dw_be 3:0 address 31:0 data 31:0	Expect status	Waits for an IO Write TLP with matching header fields. Returns a 1 on successful completion; 0 otherwise. Note that this task can only be used in conjunction with Bus Master designs.

Migration Considerations

For users migrating to Endpoint Block Plus for PCI Express from the Endpoint for PCI Express core for Virtex-II Pro and Virtex-4 FX devices, the following list describes the differences in behaviors and options between the Block Plus core and the Endpoint core (versions 3.6 and earlier). Many of the differences are illustrated in the changes to the CORE Generator GUI.

Transaction Interfaces

- The `trn_terrfd_n` signal does not exist. User applications that require transmitting poisoned TLPs must set the EP bit in the TLP header.
- Bit definitions for `trn_tbuf_av_n` vary between the two cores. See “[Transaction Interface](#),” [page 28](#) for detailed information.
- MPS-violation checking and filtering is not implemented for transmitted TLPs. User applications that erroneously transmit TLPs that violate the MPS setting can cause a fatal error to be detected by the connected PCI Express port.
- `trn_tdst_rdy_n` signal can be deasserted at any time during TLP transmission. User applications are required to monitor `trn_tdst_rdy_n` and respond appropriately. See “[Transaction Interface](#),” [page 28](#) for detailed information.
- Values reported on `trn_rfc_npd_av` are the actual credit values presented to the link partner, not the available space in the receive buffers. The value is 0, representing infinite non-posted credit.
- The `trn_rfc_cplh_av` and `trn_rfc_cpld_av` signals do not exist. User applications must follow the completion-limiting mechanism described in “[Performance Considerations on Receive Transaction Interface](#),” [page 74](#).

Configuration Interface

The Block Plus core automatically responds to PME-turnoff messages with `PME-turnoff_ack` after a short delay. There is no `cfg_turnoff_ok_n` input to the core.

System and PCI Express Interfaces

- The `trn_clk` output is a fixed frequency configured in the CORE Generator GUI. `trn_clk` does not shift frequencies in case of link recovery or training down.
- The reference clock (`sys_clk`) frequency is selectable: 100 MHz or 250 MHz. The reference clock input is a single-ended signal.
- A free running clock output (`refclkout`) is provided, based on the selection of the Reference clock (100 or 250 MHz)
- The 4-lane and 8-lane cores can train down to a 2-lane link when appropriate.

Configuration Space

- Slot Clock Configuration support is available using the CORE Generator GUI.
- The Expansion ROM BAR is always disabled.
- Accesses to non-supported configuration space result in successful completions with null data.
- There is no support for user-implemented Extended PCI Configuration Space or PCI Express Extended Capabilities.
- Power budgeting values are fixed at 0.
- There is no support for D1 or D2 PPM power-management states or the L1 as part of ASPM power-management state.