

LogiCORE™ PCI Express® Endpoint Block Plus v1.2

User Guide

UG341 February 15, 2007





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.

© 2007 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 | 1.2 | Update core to version 1.2; Xilinx tools 9.1i. |

Table of Contents

Preface: About This Guide

| | |
|----------------------------|----|
| Contents | 11 |
| Additional Resources | 12 |
| Conventions | 12 |
| Typographical | 12 |
| Online Document | 13 |

Chapter 1: Introduction

| | |
|-------------------------------------|----|
| About the Core | 15 |
| Recommended Design Experience | 15 |
| Additional Core Resources | 16 |
| Technical Support | 16 |
| Feedback | 16 |
| Core | 16 |
| Document | 16 |

Chapter 2: Core Overview

| | |
|---------------------------------------|----|
| Overview | 17 |
| Protocol Layers | 18 |
| PCI Express Configuration Space | 19 |
| Core Interfaces | 21 |
| Transaction Interface | 25 |
| Configuration Interface (CFG) | 28 |
| Error Reporting Signals | 30 |

Chapter 3: Generating and Customizing the Core

| | |
|---|----|
| Using the CORE Generator | 33 |
| Basic Parameter Settings | 34 |
| Component Name | 35 |
| Reference Clock Frequency | 35 |
| Number of Lanes | 35 |
| Interface Frequency | 36 |
| ID Initial Values | 36 |
| Class Code | 36 |
| Cardbus CIS Pointer | 37 |
| Base Address Registers | 37 |
| Base Address Register Overview | 38 |
| Managing Base Address Register Settings | 39 |
| Configuration Register Settings | 40 |
| Capabilities Register | 41 |
| Device Capabilities Register | 41 |
| Link Capabilities Register | 42 |
| Slot Clock Configuration | 42 |

| | |
|----------------------------------|----|
| Advanced Settings | 43 |
| Transaction Layer Module | 44 |
| Advanced Physical Layer | 44 |
| Power Management Registers | 44 |
| Power Consumption | 45 |
| Power Dissipated | 45 |

Chapter 4: Designing with the Core

| | |
|---|----|
| TLP Format on the User Application Interface | 48 |
| Transmitting Outbound Packets | 49 |
| Receiving Inbound Packets | 58 |
| Accessing Configuration Space Registers | 73 |
| Additional Packet Handling Requirements | 77 |
| Reporting User Error Conditions | 78 |
| Power Management | 81 |
| Generating Interrupt Requests | 83 |
| Link Training: 4-Lane and 8-Lane PCIe Block Plus Cores | 86 |
| Upstream Partner Supports Fewer Lanes | 86 |
| Lane Becomes Faulty | 86 |
| Clocking and Reset of the PCI Express Block Plus Core | 87 |
| Reset | 87 |
| Clocking | 87 |

Chapter 5: Core Constraints

| | |
|---|----|
| Contents of the User Constraints File | 91 |
| Part Selection Constraints: Device, Package, and Speedgrade | 91 |
| User Timing Constraints | 91 |
| User Physical Constraints | 91 |
| Core Pinout and I/O Constraints | 91 |
| Core Physical Constraints | 92 |
| Core Timing Constraints | 92 |
| Required Modifications | 92 |
| Device Selection | 92 |
| Core I/O Assignments | 92 |
| Core Physical Constraints | 93 |
| Core Timing Constraints | 94 |
| Relocating the PCI Express Block Plus Core | 94 |

Appendix A: Managing Receive-Buffer Space for Inbound Completions

| | |
|---|----|
| General Considerations and Concepts | 95 |
| Completion Space | 95 |
| Maximum Request Size | 95 |
| Read Completion Boundary | 96 |
| Methods of Managing Completion Space | 97 |
| The LIMIT_FC Method | 97 |
| The PACKET_FC Method | 98 |
| The RCB_FC Method | 98 |
| The DATA_FC Method | 99 |

Appendix B: PCI Express Endpoint PIO Example Design

| | |
|-------------------------------------|-----|
| System Overview | 101 |
| PIO Hardware | 102 |
| Base Address Register Support | 103 |
| TLP Data Flow | 104 |
| PIO File Structure | 105 |
| PIO Application | 106 |
| Receive Path | 107 |
| Transmit Path | 109 |
| Endpoint Memory | 110 |
| PIO Operation | 111 |
| PIO Read Transaction | 111 |
| PIO Write Transaction | 112 |
| Device Utilization | 113 |
| Summary | 113 |

Appendix C: PCI Express Endpoint Downstream Model Test Bench

| | |
|---|-----|
| Architecture | 116 |
| Simulating the Design | 117 |
| Test Selection | 117 |
| Waveform Dumping | 119 |
| Output Logging | 119 |
| Parallel Test Programs | 120 |
| Test Description | 120 |
| Test Program: pio_writeReadBack_test0 | 122 |
| Expanding the PCI Express Downstream Port Model | 123 |
| PCI Express Downstream Port Model TPI Task List | 124 |

Appendix D: Migration Considerations

| | |
|---|-----|
| Transaction Interfaces | 133 |
| Configuration Interface | 133 |
| System and PCI Express Interfaces | 133 |
| Configuration Space | 134 |

Schedule of Figures

Chapter 2: Core Overview

| | |
|--|----|
| <i>Figure 2-1: PCI Express Top-level Functional Blocks and Interfaces.</i> | 18 |
|--|----|

Chapter 3: Generating and Customizing the Core

| | |
|---|----|
| <i>Figure 3-1: PCIe Block Plus Parameters: Screen 1</i> | 34 |
| <i>Figure 3-2: PCIe Block Plus Parameters: Screen 2</i> | 35 |
| <i>Figure 3-3: PCIe Block Plus BAR Options: Screen 3</i> | 37 |
| <i>Figure 3-4: PCIe Block Plus BAR Options: Screen 4</i> | 38 |
| <i>Figure 3-5: PCIe Block Plus Configuration Settings: Screen 5</i> | 40 |
| <i>Figure 3-6: PCIe Block Plus Configuration Settings: Screen 6</i> | 41 |
| <i>Figure 3-7: PCIe Block Plus Advanced Settings: Screen 7</i> | 43 |
| <i>Figure 3-8: PCIe Block Plus Advanced Settings: Screen 8</i> | 44 |

Chapter 4: Designing with the Core

| | |
|--|----|
| <i>Figure 4-1: PCI Express Base Specification Byte Order</i> | 48 |
| <i>Figure 4-2: PCI Express Endpoint Byte Order.</i> | 48 |
| <i>Figure 4-3: TLP 3-DW Header without Payload</i> | 50 |
| <i>Figure 4-4: TLP with 4-DW Header without Payload</i> | 51 |
| <i>Figure 4-5: TLP with 3-DW Header with Payload</i> | 52 |
| <i>Figure 4-6: TLP with 4-DW Header with Payload</i> | 53 |
| <i>Figure 4-7: Back-to-back Transaction on Transmit Interface</i> | 54 |
| <i>Figure 4-8: Source Throttling on the Transmit Data Path</i> | 55 |
| <i>Figure 4-9: Destination Throttling of the Endpoint Transaction Transmit Interface</i> | 56 |
| <i>Figure 4-10: Source Driven Transaction Discontinue on Transmit Interface</i> | 57 |
| <i>Figure 4-11: TLP 3-DW Header without Payload</i> | 59 |
| <i>Figure 4-12: TLP 4-DW Header without Payload</i> | 60 |
| <i>Figure 4-13: TLP 3-DW Header with Payload</i> | 61 |
| <i>Figure 4-14: TLP 4-DW Header with Payload</i> | 62 |
| <i>Figure 4-15: User Application Throttling Receive TLP</i> | 63 |
| <i>Figure 4-16: Receive Back-To-Back Transactions</i> | 64 |
| <i>Figure 4-17: User Application Throttling Back-to-Back TLPs</i> | 64 |
| <i>Figure 4-18: Packet Re-ordering on Transaction Receive Interface</i> | 65 |
| <i>Figure 4-19: Receive Transaction Data Poisoning</i> | 67 |
| <i>Figure 4-20: BAR Target Determination using trn_rbar_hit</i> | 68 |
| <i>Figure 4-21: Receive Transaction Discontinue</i> | 69 |
| <i>Figure 4-22: Example Configuration Space Access</i> | 77 |
| <i>Figure 4-23: Signaling Unsupported Request for Non-Posted TLP</i> | 80 |
| <i>Figure 4-24: Signaling Unsupported Request for Posted TLP</i> | 80 |

| | |
|---|----|
| <i>Figure 4-25: Requesting Interrupt Service: MSI and Legacy Mode</i> | 85 |
| <i>Figure 4-26: Scaling of 4-lane Endpoint Core from 4-lane to 1-lane Operation</i> | 86 |
| <i>Figure 4-27: Embedded System Using 100 MHz Reference Clock</i> | 88 |
| <i>Figure 4-28: Embedded System Using 250 MHz Reference Clock</i> | 89 |
| <i>Figure 4-29: Open System Add-In Card Using 100 MHz Reference Clock</i> | 89 |
| <i>Figure 4-30: Open System Add-In Card Using 250 MHz Reference Clock</i> | 90 |

Appendix B: PCI Express Endpoint PIO Example Design

| | |
|--|-----|
| <i>Figure B-1: PCI Express System Overview</i> | 102 |
| <i>Figure B-2: PIO Design Components</i> | 106 |
| <i>Figure B-3: PIO 64-bit Application</i> | 106 |
| <i>Figure B-4: PIO 32-bit Application</i> | 107 |
| <i>Figure B-5: Rx Engines</i> | 107 |
| <i>Figure B-6: Tx Engines</i> | 109 |
| <i>Figure B-7: EP Memory Access</i> | 110 |
| <i>Figure B-8: Back-to-Back Read Transactions</i> | 112 |
| <i>Figure B-9: Back-to-Back Write Transactions</i> | 113 |

Appendix C: PCI Express Endpoint Downstream Model Test Bench

| | |
|---|-----|
| <i>Figure C-1: PCI Express Downstream Port Model and Top-level Endpoint</i> | 116 |
|---|-----|

Schedule of Tables

Chapter 2: Core Overview

| | |
|---|----|
| Table 2-1: Product Overview | 17 |
| Table 2-1: System Interface Signals | 21 |
| Table 2-2: PCI Express Interface Signals for the 1-lane Endpoint Core | 21 |
| Table 2-3: PCI Express Interface Signals for the 4-Lane Endpoint Core | 22 |
| Table 2-4: PCI Express Interface Signals for the 8-Lane Endpoint Core | 23 |
| Table 2-5: Transaction Transmit Interface Signals | 25 |
| Table 2-6: Transaction Receive Interface Signals | 26 |
| Table 2-7: Configuration Interface Signals | 28 |
| Table 2-8: User Application Error-Reporting Signals | 30 |

Chapter 3: Generating and Customizing the Core

| | |
|---|----|
| Table 3-1: Lane Width | 36 |
| Table 3-2: Default and Alternate Lane Width Frequency | 36 |

Chapter 4: Designing with the Core

| | |
|--|----|
| Table 4-1: <code>trn_tbuf_av[2:0]</code> Bits | 58 |
| Table 4-2: <code>trn_rbar_hit_n</code> to Base Address Register Mapping | 67 |
| Table 4-3: Transaction Receiver Credits Available Initial Values | 72 |
| Table 4-4: Command and Status Registers Mapped to the Configuration Port | 73 |
| Table 4-5: Bit Mapping on Header Status Register | 74 |
| Table 4-6: Bit Mapping on Header Command Register | 74 |
| Table 4-7: Bit Mapping on Extended Device Control Register | 75 |
| Table 4-8: Bit Mapping of PCI Express Device Status Register | 75 |
| Table 4-9: Bit Mapping of PCI Express Device Control Register | 76 |
| Table 4-10: Bit Mapping of PCI Express Link Status Register | 76 |
| Table 4-11: User-indicated Error Signaling | 79 |
| Table 4-12: Possible Error Conditions for TLPs Received by the User Application | 79 |
| Table 4-13: Interrupt Signalling | 84 |

Appendix A: Managing Receive-Buffer Space for Inbound Completions

| | |
|--|----|
| Table A-1: Receiver Buffer Completion Space | 95 |
| Table A-2: Max Request Size Settings | 95 |
| Table A-3: Read Completion Boundary Settings | 96 |
| Table A-4: Managing Receive Completion Space Methods | 97 |

Appendix B: PCI Express Endpoint PIO Example Design

| | |
|---|-----|
| <i>Table B-1: TLP Traffic Types</i> | 103 |
| <i>Table B-2: PIO Design File Structure</i> | 105 |
| <i>Table B-3: Rx Engine: Read Outputs</i> | 108 |
| <i>Table B-4: Rx Engine: Write Outputs</i> | 108 |
| <i>Table B-5: Tx Engine Inputs</i> | 109 |
| <i>Table B-6: EP Memory: Write Inputs</i> | 111 |
| <i>Table B-7: EP Memory: Read Inputs</i> | 111 |
| <i>Table B-8: PIO Design FPGA Resources</i> | 113 |

Appendix C: PCI Express Endpoint Downstream Model Test Bench

| | |
|---|-----|
| <i>Table C-1: Tests Provided with Downstream Port Model</i> | 117 |
| <i>Table C-2: Simulator Dump File Format</i> | 119 |
| <i>Table C-3: Test Setup Tasks</i> | 124 |
| <i>Table C-4: TLP Tasks</i> | 124 |
| <i>Table C-5: BAR Initialization Tasks</i> | 128 |
| <i>Table C-6: Example PIO Design Tasks</i> | 129 |
| <i>Table C-7: Expectation Tasks</i> | 130 |

About This Guide

This *PCI Express® Endpoint Block Plus v1.2 User Guide* describes the function and operation of the PCI Express Endpoint Block Plus (PCIe® Block Plus) 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, a list of additional resources, 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 PCIe Block Plus core architecture.
- [Chapter 3, “Generating and Customizing the Core,”](#) describes how to modify the PCIe Block Plus interface using the Xilinx CORE Generator™.
- [Chapter 4, “Designing with the Core,”](#) provides instructions on how to design a device using the PCIe Block Plus core.
- [Chapter 5, “Core Constraints,”](#) discusses the required and optional constraints for the PCIe Block Plus core.
- [Appendix A, “Managing Receive-Buffer Space for Inbound Completions,”](#) provides steps to track receive-buffer space for inbound completions.
- [Appendix B, “PCI Express Endpoint PIO Example Design,”](#) describes the PCIe Block Plus Programmed Input Output (PIO) transactions and the PIO example design (included with the PCIe Block Plus core when generated by the CORE Generator).
- [Appendix C, “PCI Express Endpoint Downstream Model Test Bench,”](#) describes the test bench environment, which provides a test program interface for use with the PIO example design.
- [Appendix D, “Migration Considerations”](#) defines the differences in behaviors and options between the PCIe Block Plus and PCI Express core (versions 3.3 and earlier).

Additional Resources

For additional information, go to www.xilinx.com/support. The following table lists some of the resources you can access from this website or by using the provided URLs.

| Resource | Description/URL |
|-------------------|---|
| Tutorials | Tutorials covering Xilinx design flows, from design entry to verification and debugging www.xilinx.com/support/techsup/tutorials/index.htm |
| Answer Browser | Database of Xilinx solution records www.xilinx.com/xlnx/xil_ans_browser.jsp |
| Application Notes | Descriptions of device-specific design techniques and approaches www.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes |
| Data Sheets | Device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging www.xilinx.com/xlnx/xweb/xil_publications_index.jsp |
| Problem Solvers | Interactive tools that allow you to troubleshoot your design issues www.xilinx.com/support/troubleshoot/psolvers.htm |
| Tech Tips | Latest news, design tips, and patch information for the Xilinx design environment www.xilinx.com/xlnx/xil_tt_home.jsp |

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 |

| Convention | Meaning or Use | Example |
|----------------------------------|--|--|
| <i>Italic font</i> | Variables in a syntax statement for which you must supply values | See the <i>Development System Reference Guide</i> for more information. |
| | 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 not supported or reserved | 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 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 PCIe Block Plus core and provides related information including system requirements, recommended design experience, additional core resources, technical support, and submitting feedback to Xilinx.

About the Core

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

The PCIe Block Plus solution 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 PCIe Block Plus [product page](#). For information about licensing options, see “Chapter 2, Licensing the Core,” in the *PCI Express Block Plus Getting Started Guide*.

Recommended Design Experience

Although the PCIe 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 PCIe Block Plus core, see the following documents:

- *PCI Express Endpoint Block Plus Data Sheet* (available from the product page)
- *PCI Express Endpoint Block Plus Getting Started Guide* (available from the product lounge)
- *PCI Express Endpoint Block Plus Release Notes* (available from the product lounge)
- [Virtex-5 PCI Express Endpoint Block 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 PCI Express core.

Xilinx will provide technical support for use of this product as described in the [PCI Express Endpoint Block User Guide](#) and the *PCI Express Endpoint Block Plus 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 PCIe Block Plus core and the accompanying documentation.

Core

For comments or suggestions about the PCIe Block Plus 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 PCIe Block Plus core architecture.

Overview

Table 2-1 defines the PCIe Block Plus solutions.

Table 2-1: Product Overview

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

1. See “[Link Training: 4-Lane and 8-Lane PCIe Block Plus Cores](#)” for additional information.

The PCIe Block Plus core internally instances the Virtex-5 PCI Express Endpoint Block. See the [Virtex-5 PCI Express Endpoint Block User Guide](#) (UG 197) for information regarding the internal architecture of the Block. The PCIe 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 PCIe Block Plus core, as defined below:

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

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.

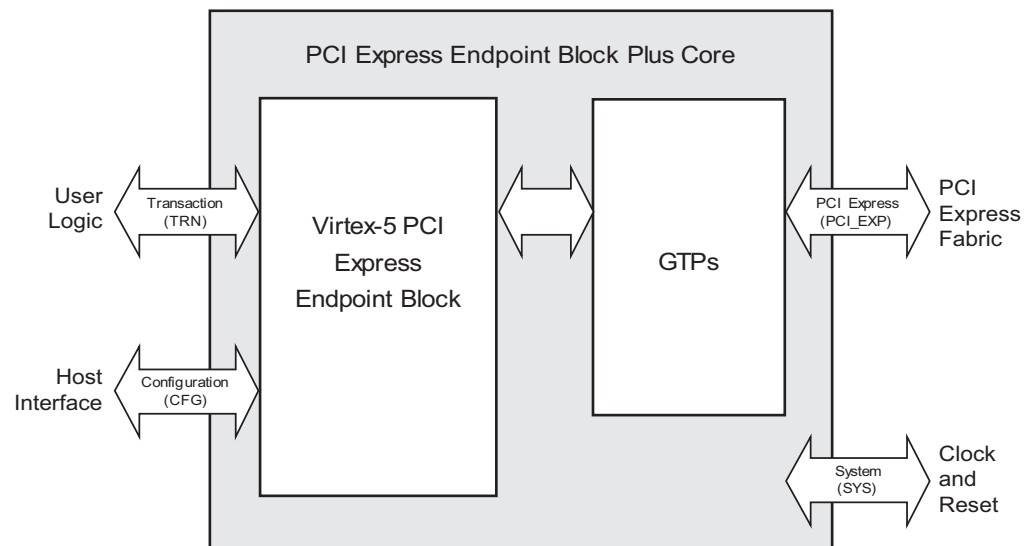


Figure 2-1: PCI Express Top-level Functional Blocks and Interfaces

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 PCIe 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.

Configuration Management

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

- Implements PCIe 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 Express Configuration Space

The configuration space consists of three primary parts, illustrated in [Table 2-2](#). 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 0xFFFF is reserved. The core returns 0x00000000 for reads to this range; writes are ignored.

Table 3: PCI Express 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 |
| 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 PCIe Block Plus core includes top-level signal interfaces that have sub-groups for the receive direction, transmit direction, and signals common to both directions.

System Interface (SYS)

The System Interface consists of the system reset signal, `sys_reset_n`, and the system clock signal, `sys_clk`, as described in [Table 2-1](#).

Table 2-1: 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. |

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 Graphical User Interface (GUI). For important information about resetting and clocking the PCIe Block Plus core, see [“Clocking and Reset of the PCI Express Block Plus Core,” page 87](#) and the [Virtex-5 PCI Express Endpoint Block User Guide](#).

PCI Express Interface

The PCIe interface (PCI_EXP) consists of differential transmit and receive pairs organized in multiple lanes. A PCIe 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 Endpoint core supports only Lane 0, the 4-Lane Endpoint core supports lanes 0-3, and the 8-Lane Endpoint core supports lanes 0-7. Transmit and receive signals of the PCI_EXP interface are defined in [Tables 2-2](#), [2-3](#), and [2-4](#).

Table 2-2: 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 2-3: 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 2-4: 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 2-4: PCI Express Interface Signals for the 8-Lane Endpoint Core (Continued)

| 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 Interface (TRN) 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 2-5, 2-6, 2-7, and 2-8](#).

Transmit (Tx) TRN Interface

[Table 2-5](#) defines the Transmit Interface signal names and descriptions.

Table 2-5: Transaction Transmit Interface Signals

| Name | Direction | Description |
|-------------------|-----------|--|
| trn_tsof_n | Input | Transmit Start-of-Frame (SOF): Active low. Signals the start of a packet. |
| trn_teof_n | Input | Transmit End-of-Frame (EOF): Active low. Signals the end of a packet. |
| 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: 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]. |
| 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. |
| trn_tbuf_av [2:0] | Output | <p>Transmit Buffers Available: Indicates transmit buffer availability in the core. Each bit of trn_tbuf_av corresponds to one of the following credit queues:</p> <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 <p>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.</p> |

Receive (Rx) TRN Interface

Table 2-6 lists the Rx Interface signal names and descriptions.

Table 2-6: Transaction Receive Interface Signals

| Name | Direction | Description |
|----------------------|-----------|--|
| trn_rsof_n | Output | Receive Start-of-Frame (SOF): Active low. Signals the start of a packet. |
| trn_reof_n | Output | Receive End-of-Frame (EOF): Active low. Signals the end of a packet. |
| trn_rd[63:0] | Output | Receive Data: Packet data being received. |
| 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: 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. |
| 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, allowing Posted and Completion packets to bypass Non-Posted packets in the inbound queue, if necessitated by the User Application. 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 last Non-Posted TLP the User Application can accept. |
| trn_rcpl_streaming_n | Input | Receive Completion Streaming: Active low. Asserted to enable Upstream Memory Read transmission without the need for throttling. See “Completion Streaming on Transaction Receive Interface,” page 70 for details. |

Table 2-6: Transaction Receive Interface Signals (Continued)

| Name | Direction | Description |
|-----------------------------------|-----------|--|
| trn_rbar_hit_n[6:0] | Output | Receive BAR Hit: Active low. Indicates BAR(s) targeted by the current receive transaction. trn_rbar_hit_n[0] => BAR0 trn_rbar_hit_n[1] => BAR1 trn_rbar_hit_n[2] => BAR2 trn_rbar_hit_n[3] => BAR3 trn_rbar_hit_n[4] => BAR4 trn_rbar_hit_n[5] => BAR5 trn_rbar_hit_n[6] => 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 (CFG)

The configuration interface enables the user design to inspect the state of the PCIe Endpoint configuration space. The user provides a 10-bit configuration address, which selects one of the 1024 configuration space double word (DWORD) registers. The endpoint will return the state of the selected register over the 32-bit data output port. [Table 2-7](#) defines the configuration interface signals. See “[Accessing Configuration Space Registers](#),” [page 73](#) for usage.

Table 2-7: 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 PCIe endpoint. |
| 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. |
| cfg_interrupt_n | Input | Configuration Interrupt: Active-low interrupt-request signal. The User Application can 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_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 |

Table 2-7: Configuration Interface Signals (Continued)

| Name | Direction | Description |
|--------------------------|-----------|---|
| cfg_bus_number[7:0] | Output | Configuration Bus Number: This output provides the assigned bus number for the device. The User Application must use this information in the Bus Number field of outgoing TLP request. Default value after reset is 00h. Refreshed whenever a Type 0 Configuration packet is received. |
| cfg_device_number[4:0] | Output | Configuration Device Number: This output provides the assigned device number for the device. The User Application must use this information in the Device Number field of outgoing TLP request. Default value after reset is 00000b. 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: The status register from the Configuration Space Header. |
| cfg_command[15:0] | Output | Configuration Command: The command register from the Configuration Space Header. |
| cfg_dstatus[15:0] | Output | Configuration Device Status: The device status register from the PCI Express Extended Capability Structure. |
| cfg_dcommand[15:0] | Output | Configuration Device Command: The 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. |

Table 2-7: Configuration Interface Signals (Continued)

| Name | Direction | Description |
|----------------------------|-----------|--|
| cfg_pcie_link_state_n[2:0] | Output | PCI Express Link State: This one-hot encoded bus 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" |
| 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 PCI Express Device Serial Number extended capability. |

- Writing to the configuration space through the user configuration port is not supported at this time. These ports are on the Endpoint core to allow for future enhancement of this feature.

Error Reporting Signals

Table 2-8 defines the User Application error-reporting signals.

Table 2-8: 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. |

Table 2-8: User Application Error-Reporting Signals (Continued)

| Port Name | Direction | Description |
|------------------------------|-----------|--|
| 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. |
| cfg_err_tlp_cpl_header[47:0] | Input | 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. The following information should be extracted from the received error TLP and presented in the format below: [47:41] Lower Address [40:29] Byte Count [28:26] TC [25:24] Attr [23:8] Requester ID [7:0] Tag |

Generating and Customizing the Core

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

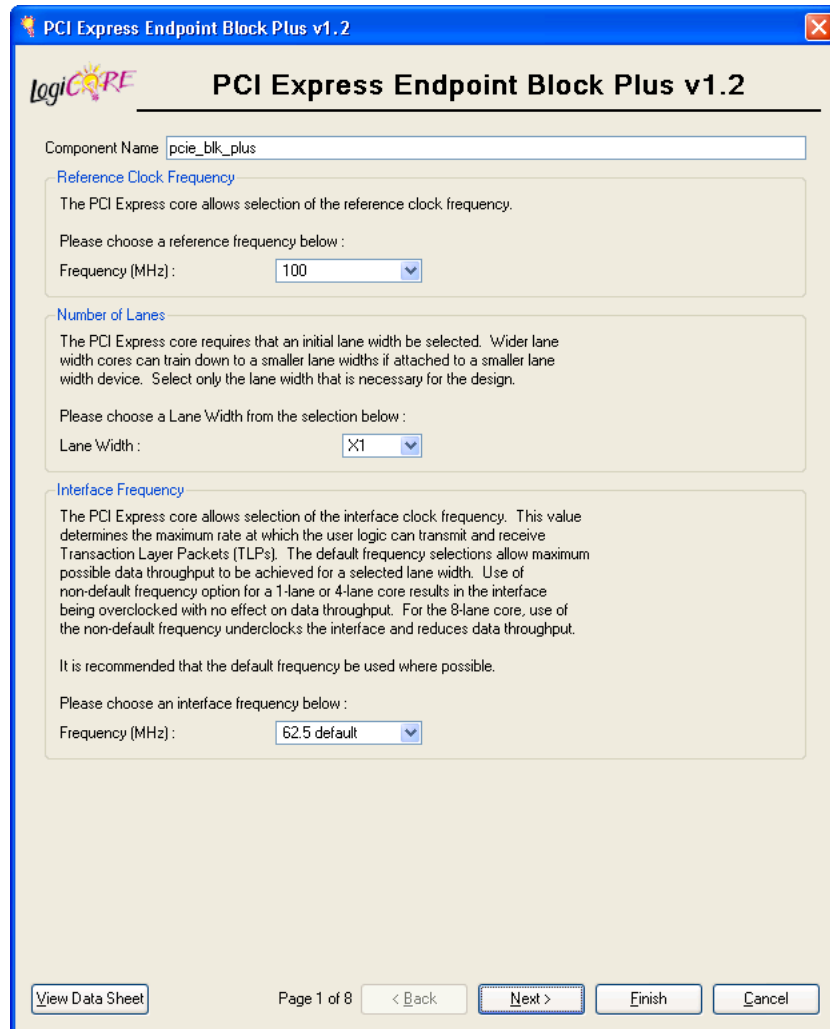
Using the CORE Generator

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



PCI Express Endpoint Block Plus v1.2

LogiCORE

PCI Express Endpoint Block Plus v1.2

Component Name: pcie_blk_plus

Reference Clock Frequency

The PCI Express core allows selection of the reference clock frequency.

Please choose a reference frequency below :

Frequency (MHz): 100

Number of Lanes

The PCI Express core requires that an initial lane width be selected. Wider lane width cores can train down to a smaller lane widths if attached to a smaller lane width device. Select only the lane width that is necessary for the design.

Please choose a Lane Width from the selection below :

Lane Width: X1

Interface Frequency

The PCI Express 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.

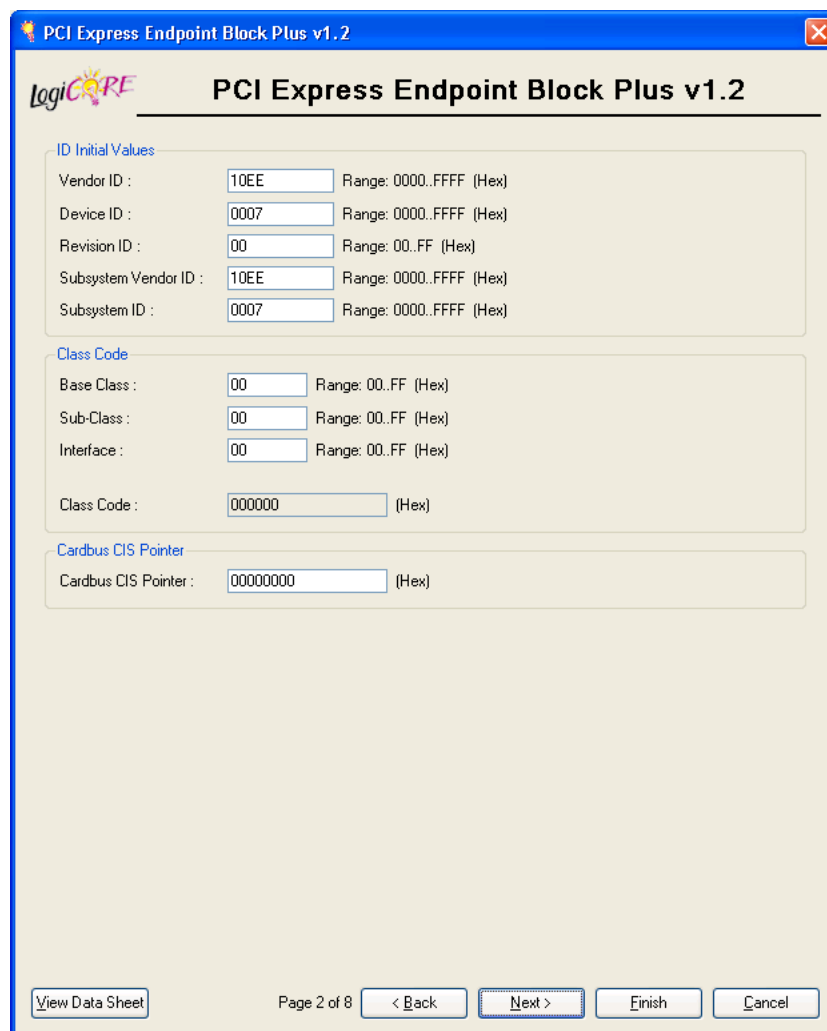
It is recommended that the default frequency be used where possible.

Please choose an interface frequency below :

Frequency (MHz): 62.5 default

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

Figure 3-1: PCIe Block Plus Parameters: Screen 1



PCI Express Endpoint Block Plus v1.2

ID Initial Values

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

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

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

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

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

Class Code

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

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

Interface : Range: 00..FF (Hex)

Class Code : (Hex)

Cardbus CIS Pointer

Cardbus CIS Pointer : (Hex)

[View Data Sheet](#) Page 2 of 8 [< Back](#) [Next >](#) [Finish](#) [Cancel](#)

Figure 3-2: PCIe 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 PCIe Block Plus core, see ["Clocking and Reset of the PCI Express Block Plus Core," page 87](#) and the [Virtex-5 PCI Express Endpoint Block User Guide](#) (UG197).

Number of Lanes

The PCIe Block Plus core requires that the initial lane width be selected. The choices and the associated generated products are shown in [Table 3-1](#). 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 PCIe Block Plus Cores](#),” page 86 for more information.

Table 3-1: Lane Width

| Lane Width | Product Generated |
|------------|----------------------|
| x1 | PCIe 1 Lane Endpoint |
| x4 | PCIe 4 Lane Endpoint |
| x8 | PCIe 8 Lane Endpoint |

Interface Frequency

The clock frequency of the core's user interface is selectable. For each lane width, there are two available frequency choices: a default frequency and an alternate frequency, as defined in [Table 3-2](#). Where possible, Xilinx recommends that the default frequency be used. 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.
- **Subvendor ID.** Further qualifies the manufacturer of the device or application. This value is typically the same as the Vendor 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.

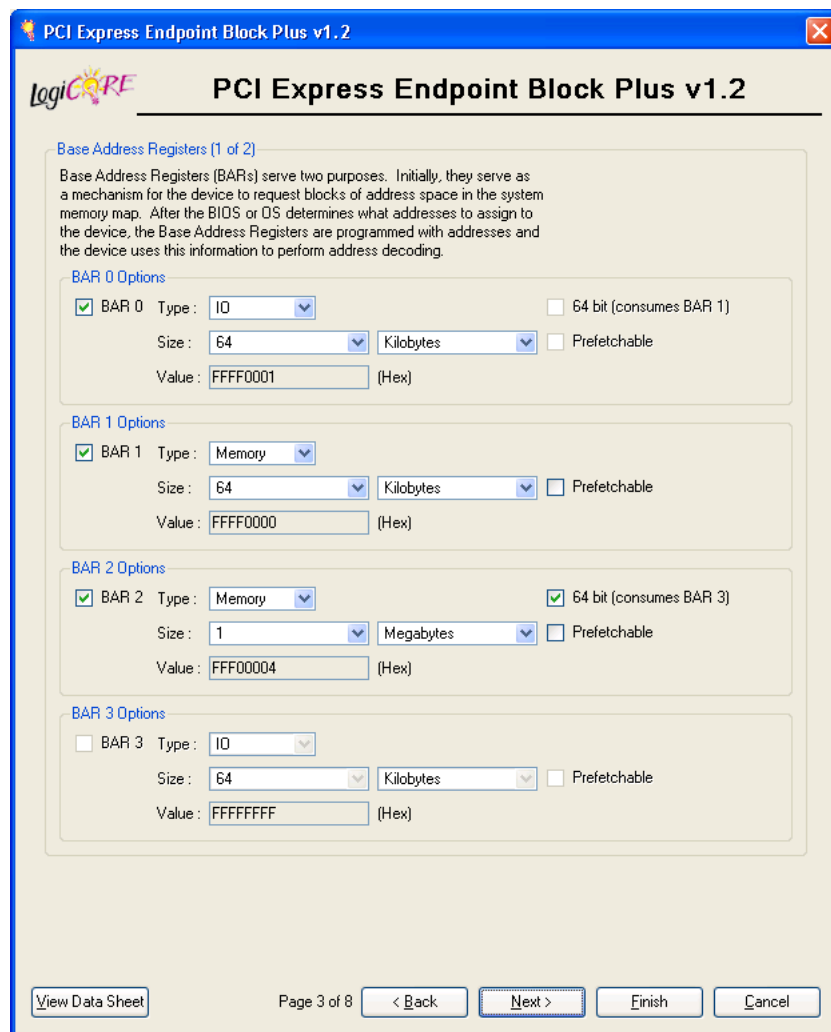
Class code encoding can be found at www.pcisig.com.

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.



PCI Express Endpoint Block Plus v1.2

LogiCORE

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: IO ☐ 64 bit (consumes BAR 1)
 Size: 64 Kilobytes ☐ Prefetchable
 Value: FFFF0001 (Hex)

BAR 1 Options

☒ BAR 1 Type: Memory ☐ Prefetchable
 Size: 64 Kilobytes
 Value: FFFF0000 (Hex)

BAR 2 Options

☒ BAR 2 Type: Memory ☒ 64 bit (consumes BAR 3) ☐ Prefetchable
 Size: 1 Megabytes
 Value: FFF00004 (Hex)

BAR 3 Options

☐ BAR 3 Type: IO ☐ Prefetchable
 Size: 64 Kilobytes
 Value: FFFFFFFF (Hex)

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

Figure 3-3: PCIe Block Plus BAR Options: Screen 3

PCI Express Endpoint Block Plus v1.2

LogiCORE

PCI Express Endpoint Block Plus v1.2

Base Address Registers (2 of 2)

BAR 4 Options

☐ BAR 4 Type: IO 64 bit (consumes BAR 5)

Size: 64 Kilobytes Prefetchable

Value: 00000000 (Hex)

BAR 5 Options

☐ BAR 5 Type: IO

Size: 64 Kilobytes Prefetchable

Value: 00000000 (Hex)

Expansion ROM Base Address Register

☒ Expansion Rom Size: 1 Megabytes

Value: FFF00001 (Hex)

View Data Sheet Page 4 of 8 < Back Next > Finish Cancel

Figure 3-4: PCIe Block Plus BAR Options: Screen 4

Base Address Register Overview

The PCIe Block Plus core 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

A 1-megabyte Expansion ROM BAR is always enabled in the PCIe Block Plus core.

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 (non-Legacy) Endpoint, 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.

PCI Express Endpoint Block Plus v1.2

LogiCORE

PCI Express Endpoint Block Plus v1.2

Configuration Register Settings (1 of 2)

Capabilities Register

Capability Version : 1 Range: 0..F (Hex)

Device Port/Type : PCI Express Endpoint device

Capabilities Register : 0001 (Hex)

Device Capabilities Register

Max Payload Size : 512 bytes

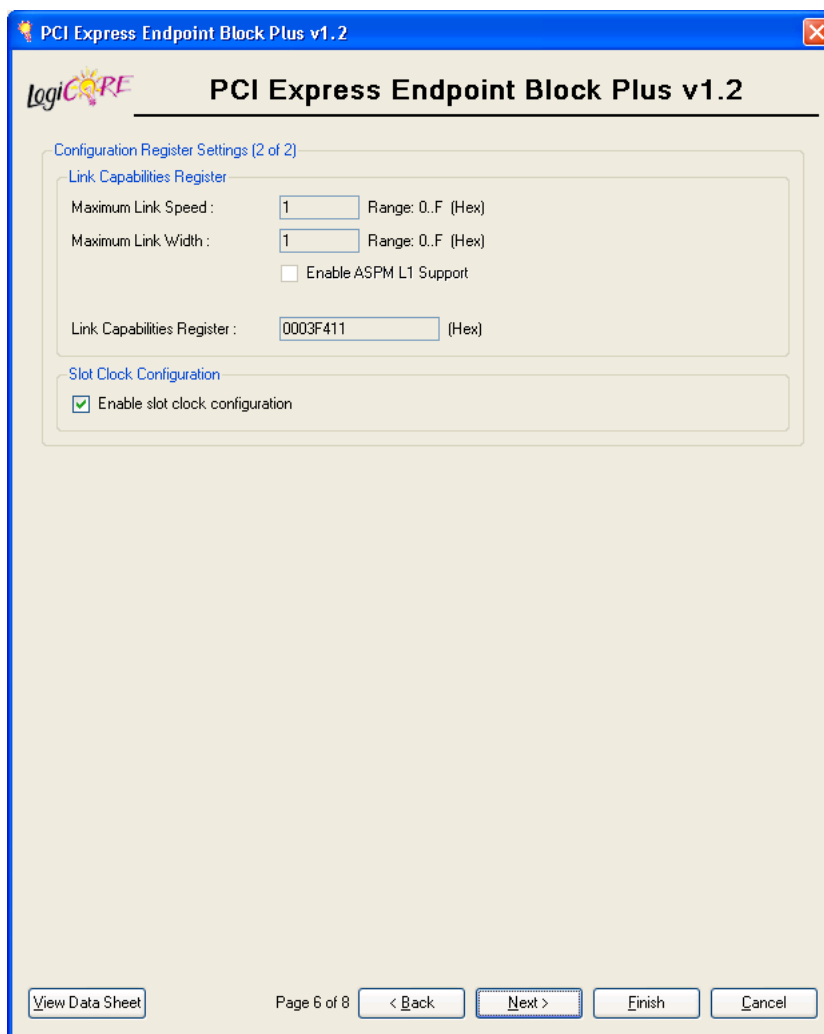
Acceptable L0s Latency : No limit

Acceptable L1 Latency : No limit

Device Capabilities Register : 00000FC5 (Hex)

View Data Sheet Page 5 of 8 < Back Next > Finish Cancel

Figure 3-5: PCIe Block Plus Configuration Settings: Screen 5



PCI Express Endpoint Block Plus v1.2

LogiCORE

PCI Express Endpoint Block Plus v1.2

Configuration Register Settings (2 of 2)

Link Capabilities Register

Maximum Link Speed : Range: 0..F (Hex)

Maximum Link Width : Range: 0..F (Hex)

☐ Enable ASPM L1 Support

Link Capabilities Register : (Hex)

Slot Clock Configuration

☒ Enable slot clock configuration

[View Data Sheet](#) Page 6 of 8 [< Back](#) [Next >](#) [Finish](#) [Cancel](#)

Figure 3-6: PCIe Block Plus 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 PCIe 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.

Advanced Settings

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

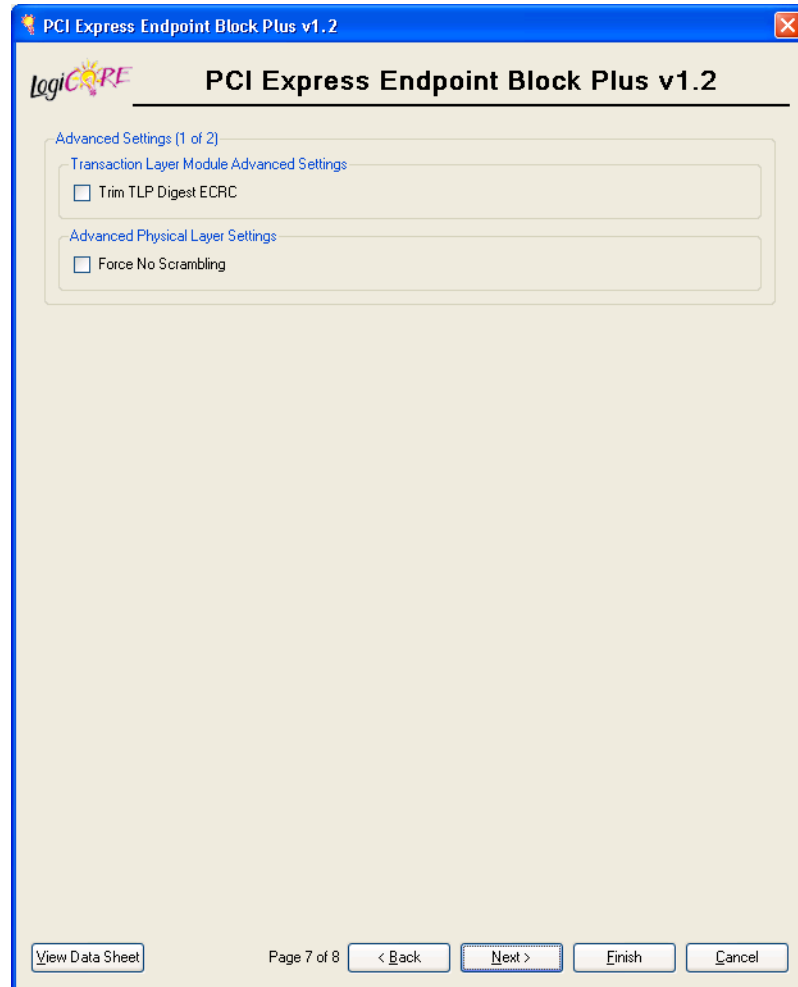


Figure 3-7: PCIe Block Plus Advanced Settings: Screen 7

PCI Express Endpoint Block Plus v1.2

LogiCORE

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 | 1 | = 0.000 (watts) |
| D1: | 0 | 1 | = 0.000 (watts) |
| D2: | 0 | 1 | = 0.000 (watts) |
| D3: | 0 | 1 | = 0.000 (watts) |

Power Dissipation

| | Power Dissipated | Scale Factor | Total Power |
|-----|------------------|--------------|-----------------|
| D0: | 0 | 1 | = 0.000 (watts) |
| D1: | 0 | 1 | = 0.000 (watts) |
| D2: | 0 | 1 | = 0.000 (watts) |
| D3: | 0 | 1 | = 0.000 (watts) |

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

Figure 3-8: PCIe Block Plus 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 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.

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 PCIe Block Plus 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 PCIe Block Plus 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 PCIe Block Plus core. See section 3.2.3 of the *PCI Bus Power Management Interface Specification Revision 1.2*.
- **AUX Max Current.** The PCIe Block Plus 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 PCIe Block Plus core 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 PCIe Block Plus core 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 PCIe Block Plus 64-bit 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 PCIe Block Plus Cores”
- ◆ “Clocking and Reset of the PCI Express Block Plus Core”

TLP Format on the User Application 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).

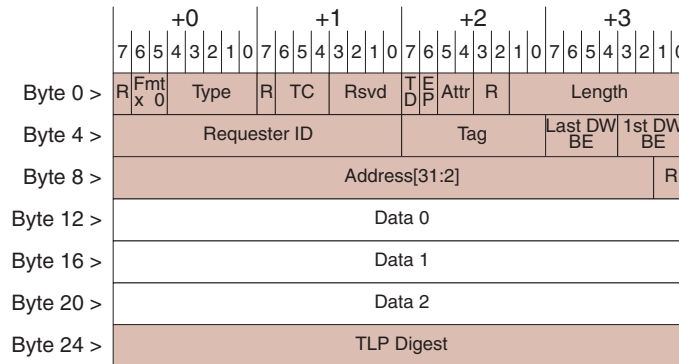


Figure 4-1: PCI Express Base Specification Byte Order

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 User Application 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*.

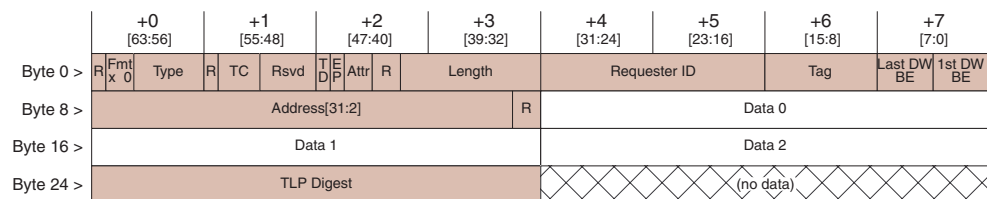


Figure 4-2: PCI Express Endpoint Byte Order

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 PCIe Block Plus core does not check the TLP Digest for incoming packets.

Transmitting Outbound Packets

Basic TLP Transmit Operation

The PCIe Block Plus core 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.

Table 2-5, page 25 defines the transmit User Application signals. To transmit a TLP, the transmit User Application must perform the following sequence of events on the Transaction transmit 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 transmit 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 transmit 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 transmit 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.

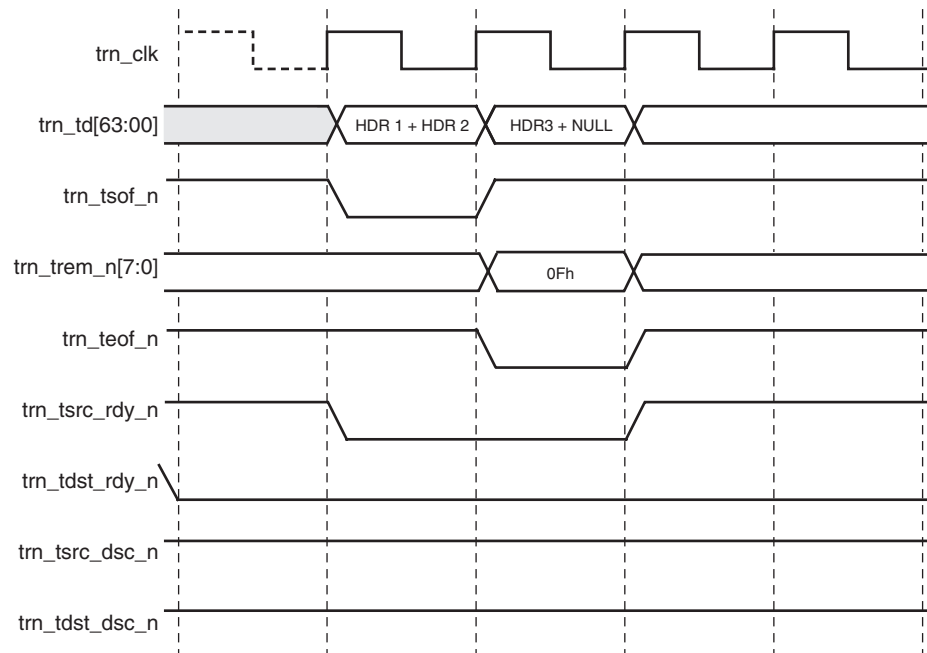


Figure 4-3: TLP 3-DW Header without Payload

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.

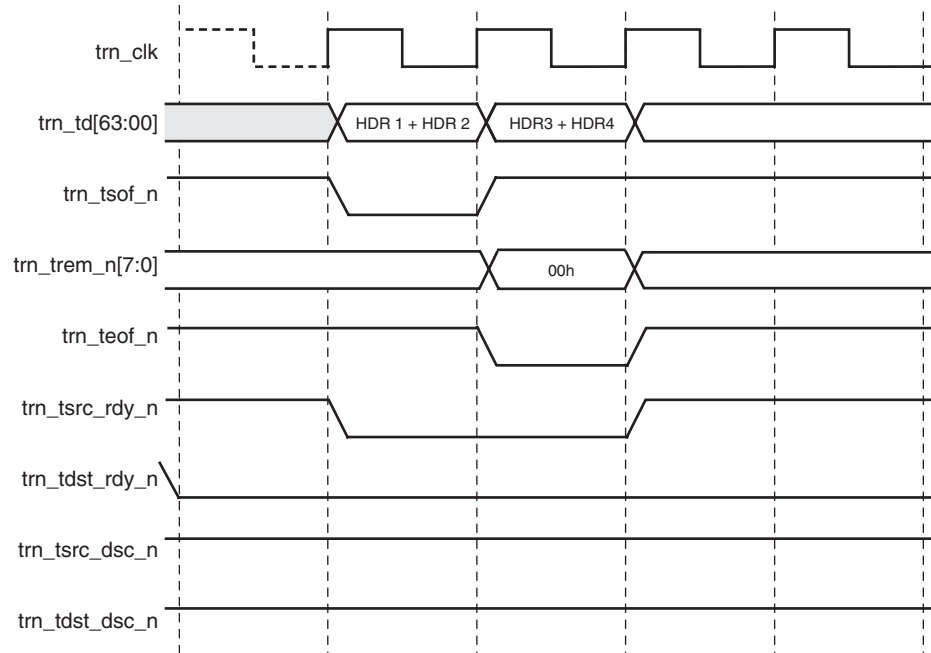


Figure 4-4: TLP with 4-DW Header without Payload

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.

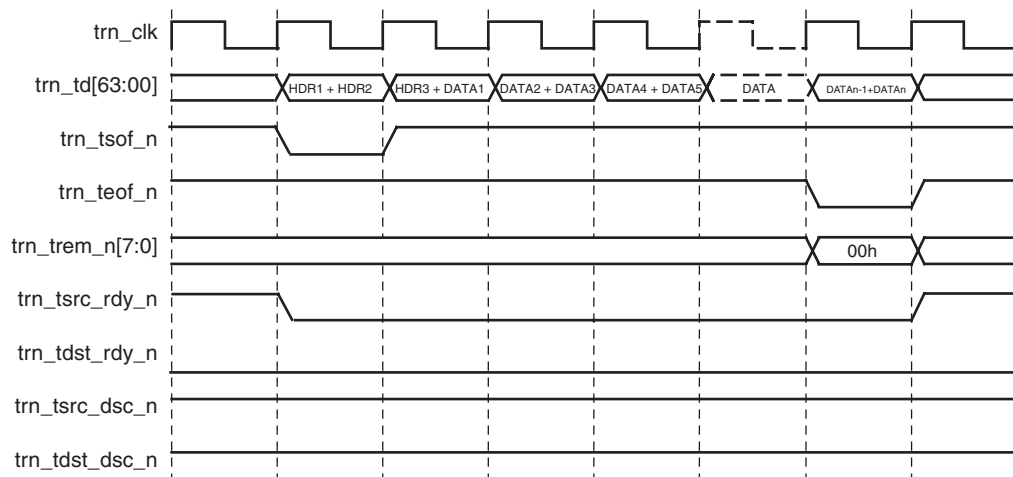


Figure 4-5: TLP with 3-DW Header with Payload

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 selected for the final data cycle creates a packet of length equivalent to the length field in the packet header.

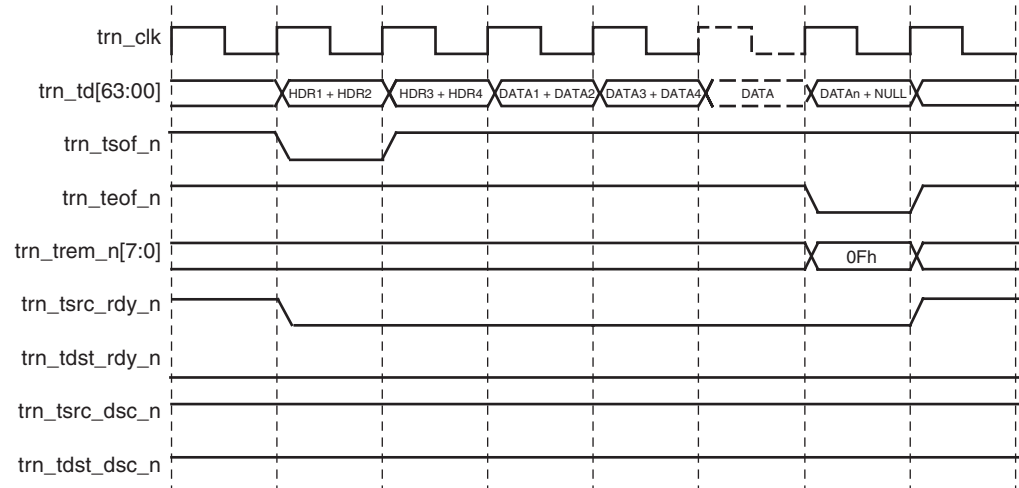


Figure 4-6: TLP with 4-DW Header with Payload

Presenting Back-to-Back Transactions on the Transmit Interface

The transmit User Application can present back-to-back TLPs on the Transaction transmit interface to maximize bandwidth utilization. Figure 4-7 illustrates back-to-back TLPs presented on the transmit interface. The transmit 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.

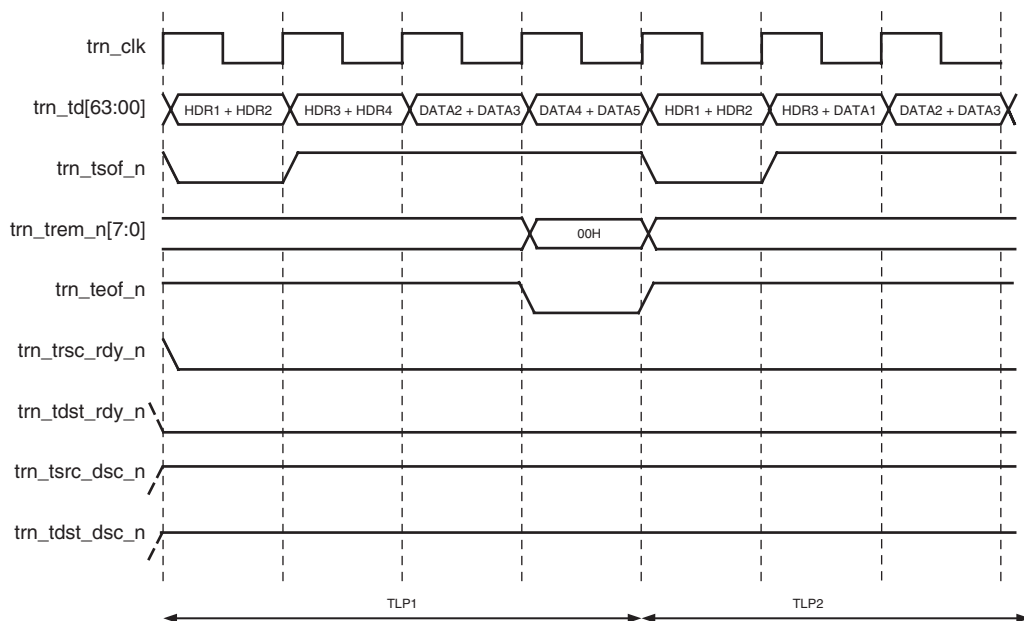


Figure 4-7: Back-to-back Transaction on Transmit Interface

Source Throttling on the Transmit Data Path

The Endpoint core Transaction transmit 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 Endpoint core Transaction transmit 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.

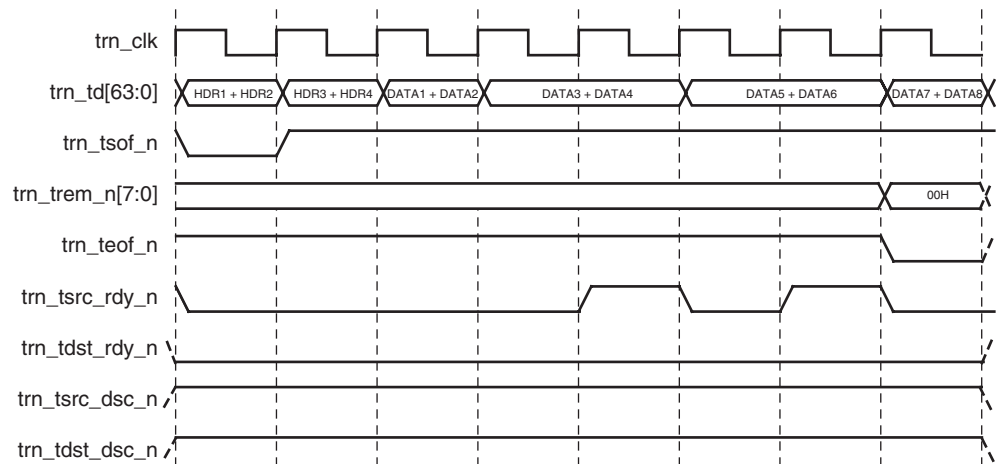


Figure 4-8: Source Throttling on the Transmit Data Path

Destination Throttling of the Transmit Data Path

The Endpoint core Transaction transmit 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 transmit User Application is presenting TLPs. Figure 4-9 illustrates the deassertion of `trn_tdst_rdy_n` to throttle the transmit User Application when the PCIe Block Plus core's internal transmit buffers are full.

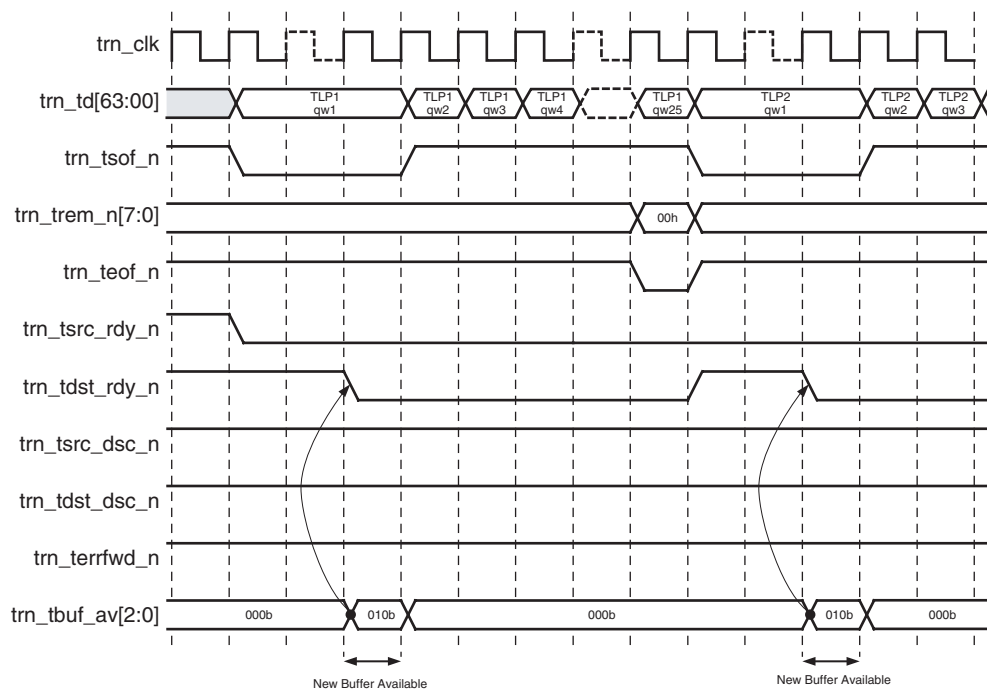


Figure 4-9: Destination Throttling of the Endpoint Transaction Transmit Interface

The Endpoint core 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 ongoing TLP is accepted completely and **trn_tdst_rdy_n** is subsequently deasserted, disallowing the transmit 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 Endpoint core Transaction transmit 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_tsof_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.

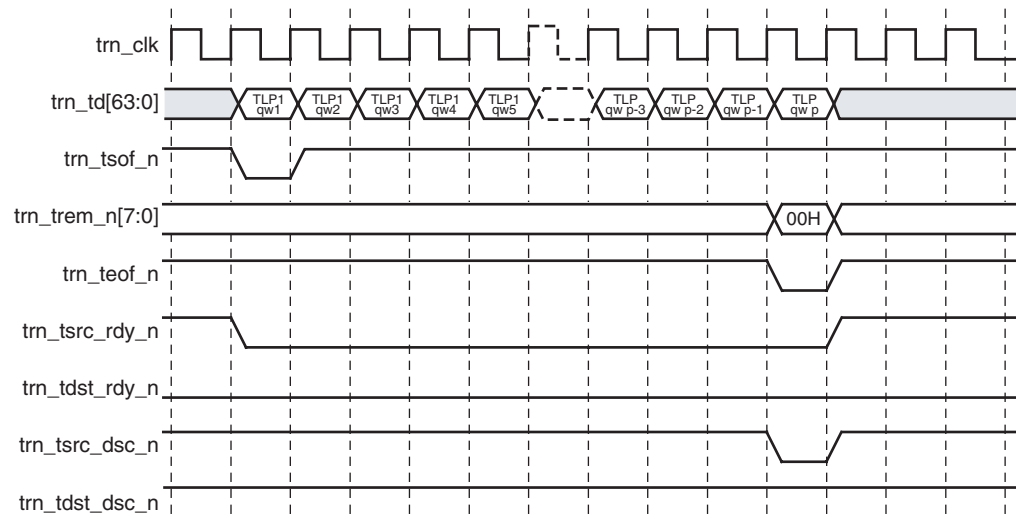


Figure 4-10: Source Driven Transaction Discontinue on Transmit Interface

Packet Data Poisoning on the Transaction Transmit 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 Endpoint 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 1-DWORD TLP Digest is properly computed and appended 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 plus a data payload equal to the MAX_PAYLOAD_SIZE of the core (as defined in the Device Capability register) plus 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 PCIe 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 PCIe Block Plus core 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 PCIe Block Plus 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[2:0]`) signal enables the transmit 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[2:0]` 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.

Table 4-1: trn_tbuf_av[2:0] 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 |

Receiving Inbound Packets

Basic TLP Receive Operation

[Table 2-6, page 26](#) defines the receive User Application signals. The following sequence of events must occur on the Transaction receive 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_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]`.

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.

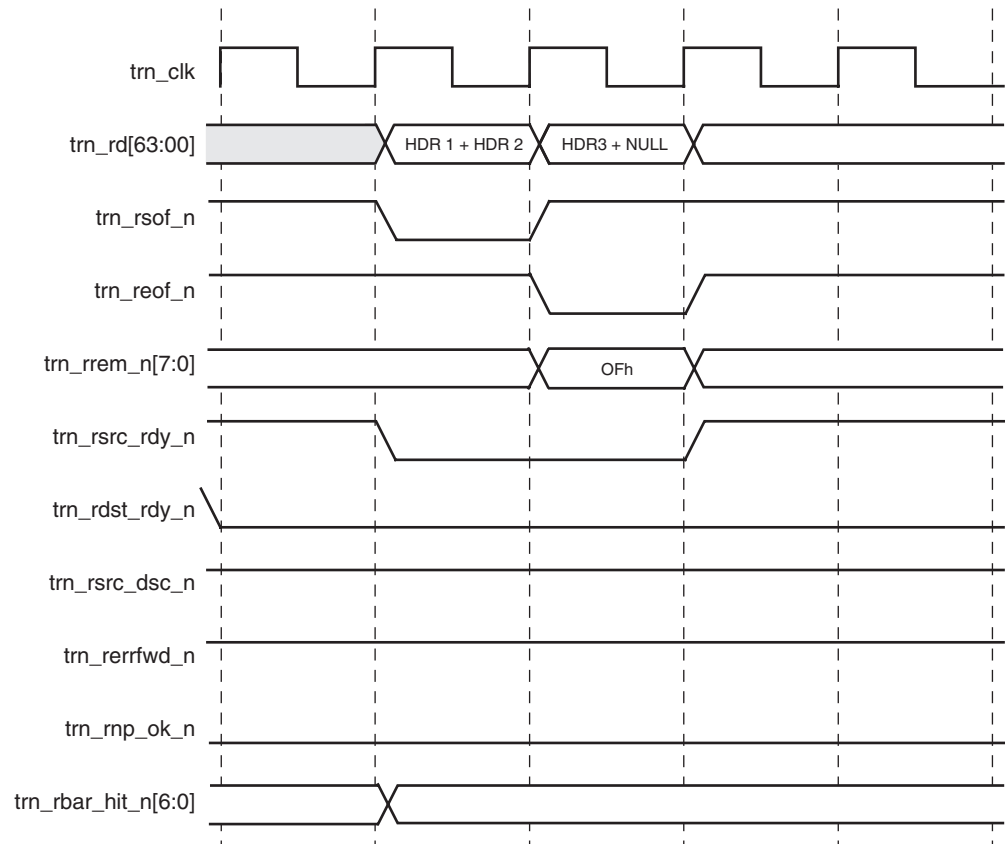


Figure 4-11: TLP 3-DW Header without Payload

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.

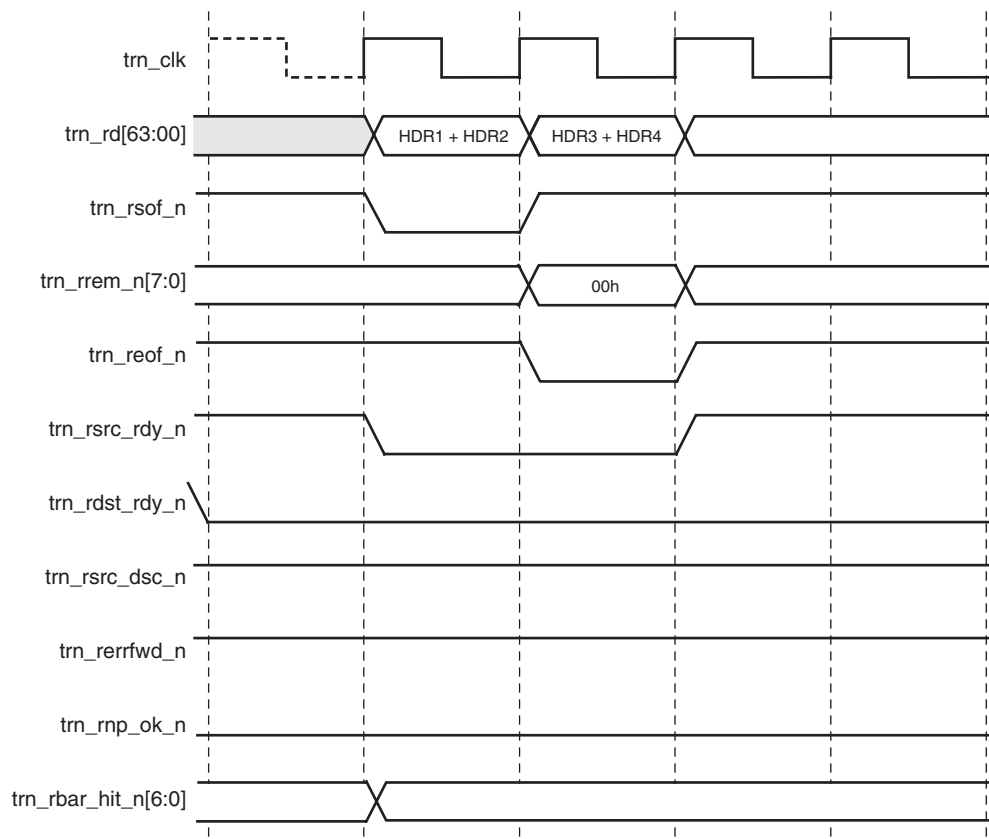


Figure 4-12: TLP 4-DW Header without Payload

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.

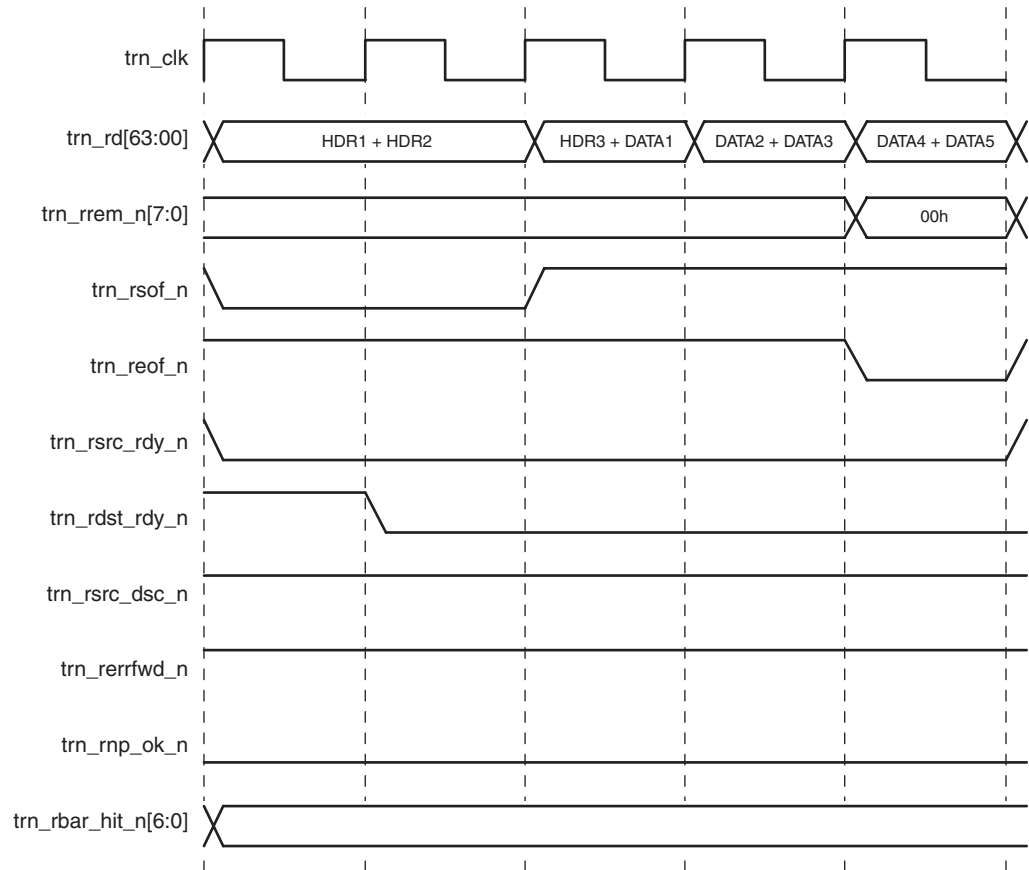


Figure 4-13: TLP 3-DW Header with Payload

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.

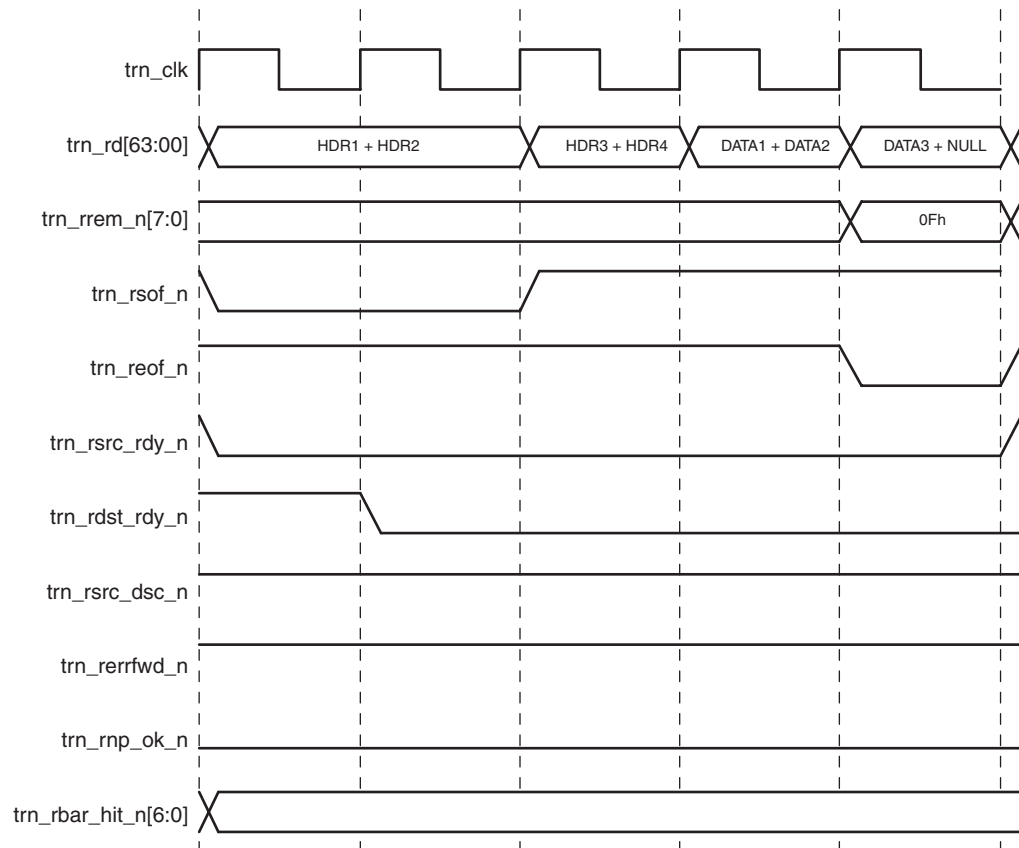


Figure 4-14: TLP 4-DW Header with Payload

Throttling the Data Path on the Receive Interface

The User Application can 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 Endpoint 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.

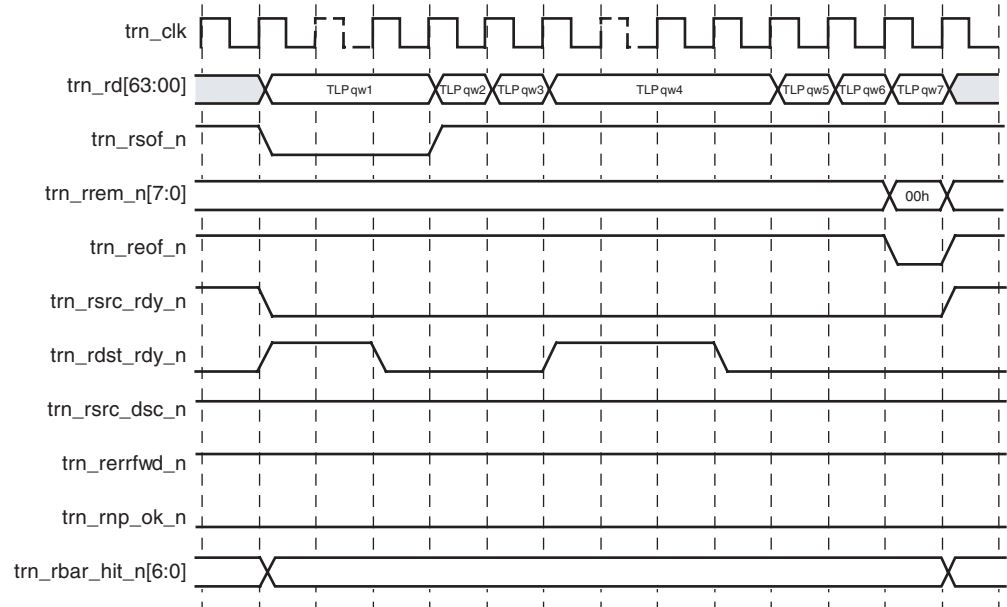


Figure 4-15: User Application Throttling Receive TLP

Receiving Back-To-Back Transactions on the Receive Interface

The User Application logic must be designed to handle presentation of back-to-back TLPs on the receive 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.

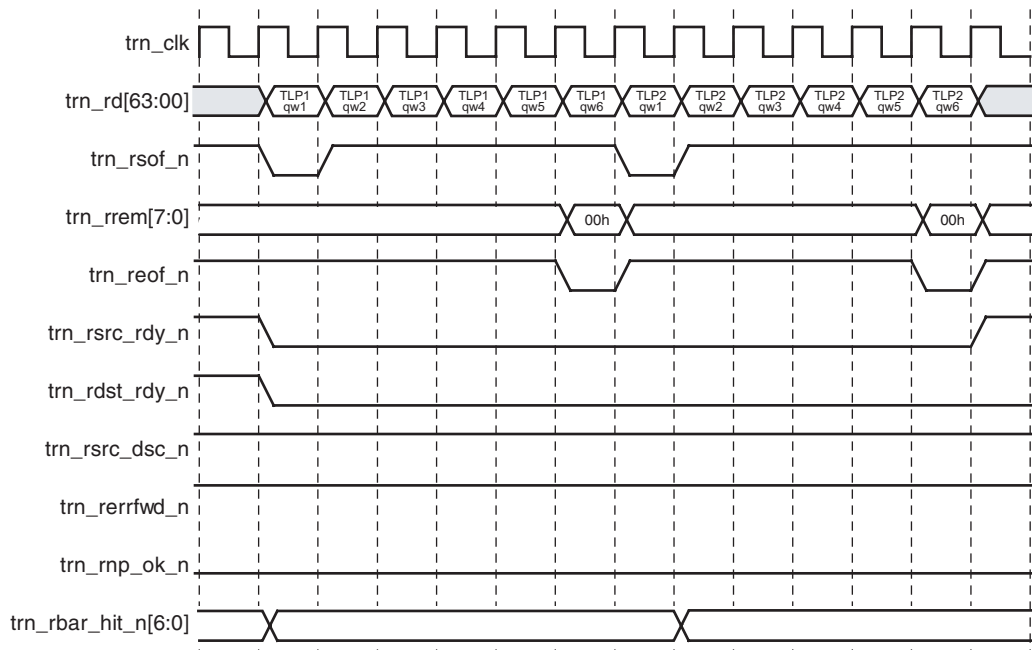


Figure 4-16: Receive Back-To-Back Transactions

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.

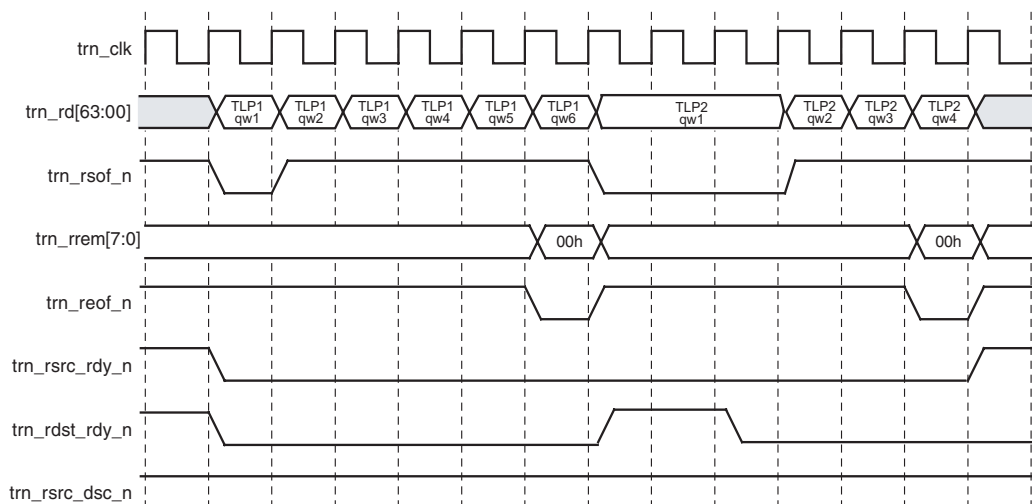


Figure 4-17: User Application Throttling Back-to-Back TLPs

Packet Re-ordering on Transaction Receive 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 [“Completion Streaming on Transaction Receive Interface”](#) for more information on `trn_rcpl_streaming_n`.

The receive User Application can 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 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 receive User Application before other received TLPs.

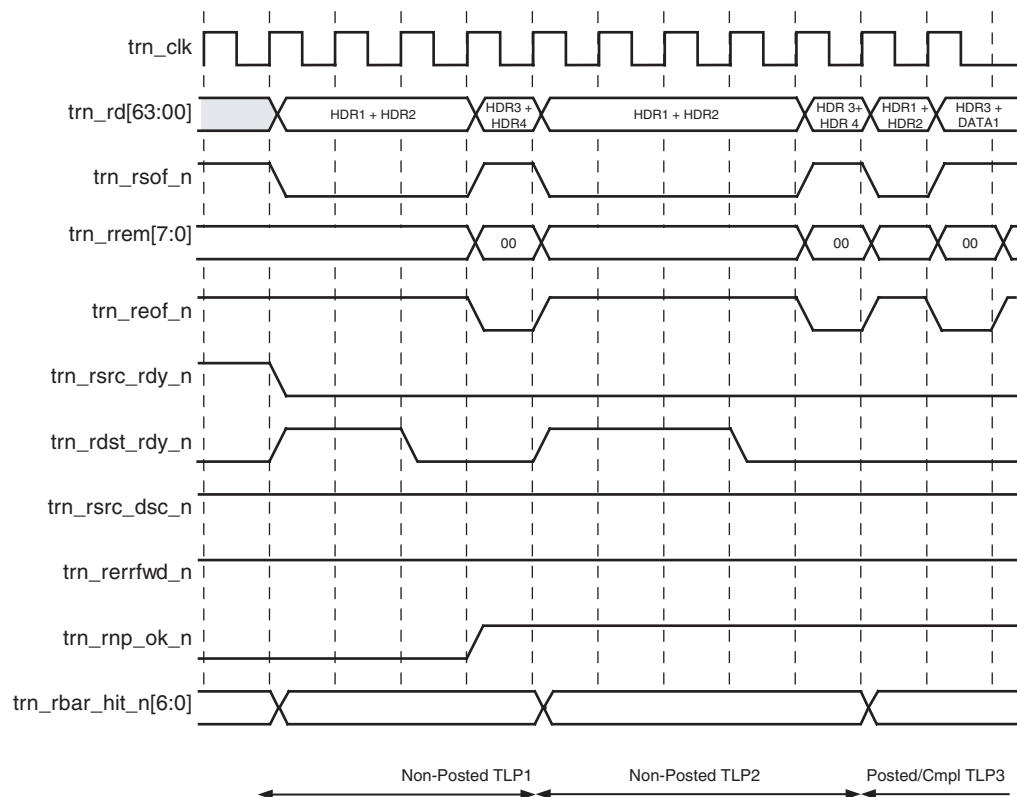


Figure 4-18: Packet Re-ordering on Transaction Receive Interface

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 Transaction Receive Interface

To simplify logic within the User Application, the Endpoint 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 Endpoint 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 Endpoint 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 Endpoint 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 Endpoint 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.

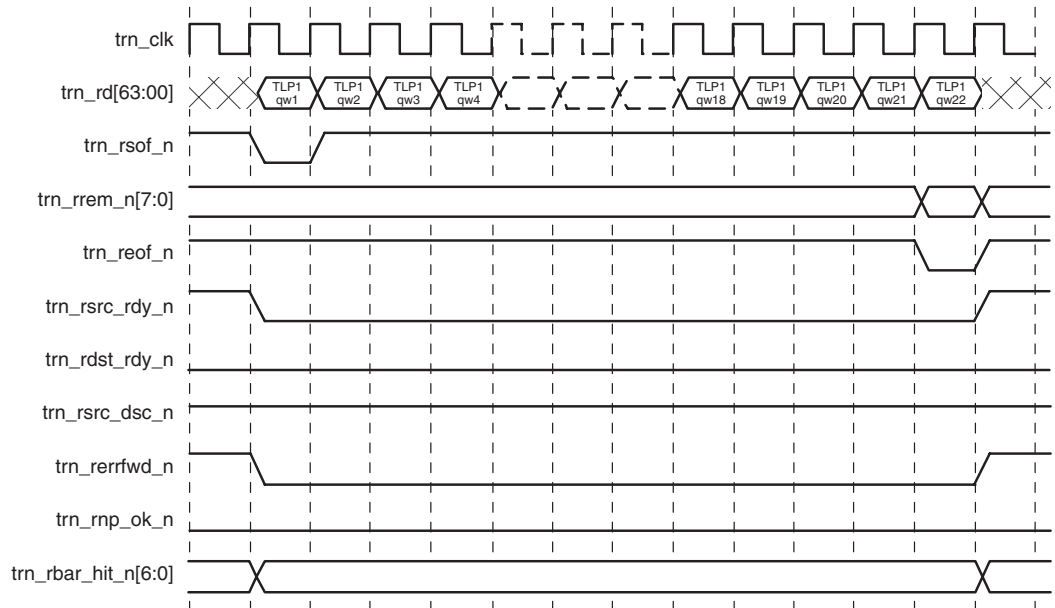


Figure 4-19: Receive Transaction Data Poisoning

Packet Base Address Register Hit on Transaction Receive Interface

The Endpoint core decodes incoming Memory and IO TLP request addresses to determine which Base Address Register (BAR) in the Endpoint'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.

[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 | 30h |

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.

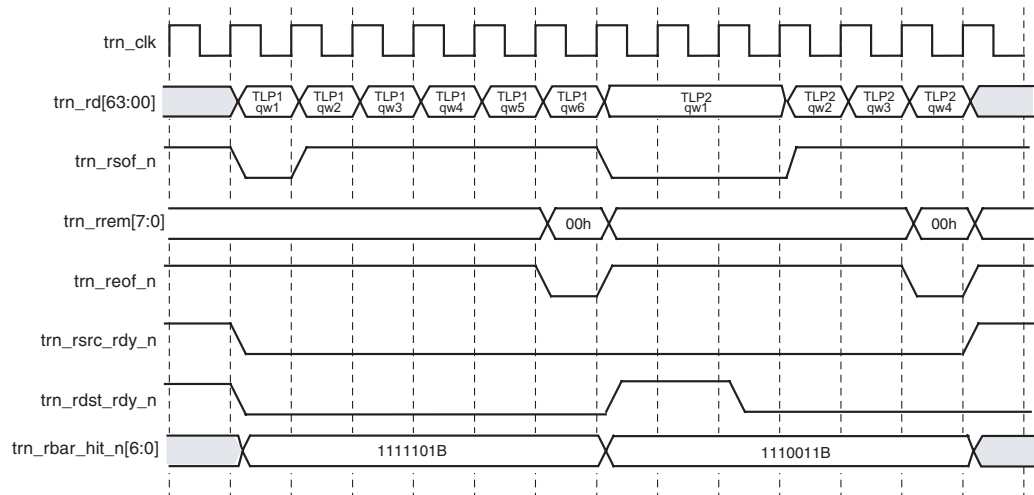


Figure 4-20: BAR Target Determination using `trn_rbar_hit`

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.

Packet Transfer Discontinue on Transaction Receive Interface

The PCIe Block Plus core asserts `trn_rsrc_dsc_n` if communication with the link partner is lost, which results in the termination of an *in-progress* TLP. The loss of communication with the link partner is signaled by deassertion of `trn_lnk_up_n`. When `trn_lnk_up_n` is deasserted, it effectively acts as a *Hot Reset* to the entire core. For this reason, all TLPs stored inside the core or being presented to the receive interface are irrecoverably lost. Figure 4-21 illustrates packet transfer discontinue scenario.

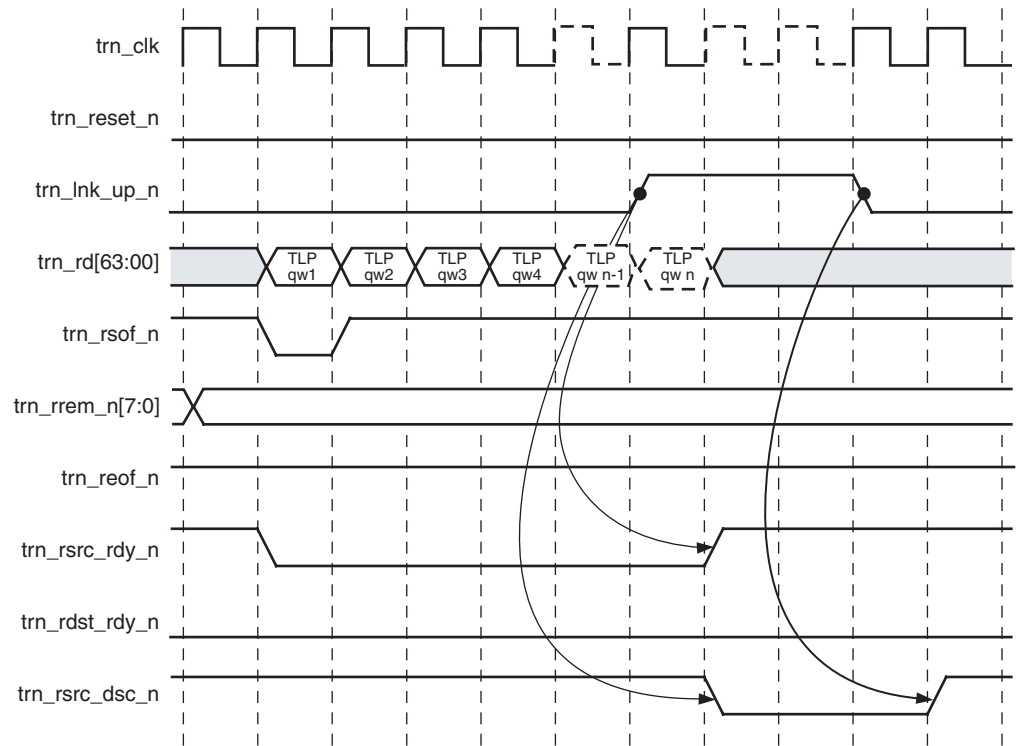


Figure 4-21: Receive Transaction Discontinue

Completion Streaming on Transaction Receive 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 PCIe Block Plus core receiver can store up to 8 Completion or Completion with Data TLPs to satisfy this requirement. When the User Application does not generate requests that could result in more than 8 outstanding Completion TLPs, the PCIe Block Plus can be used in non-streaming mode (`trn_rcpl_streaming_n` deasserted). See [Appendix A, “Managing Receive-Buffer Space for Inbound Completions”](#) for information on determining the number of outstanding Completions, and for instructions on managing Memory Read Requests when not using Completion Streaming.

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 PCIe Block Plus 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 PCIe Block Plus 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 PCIe Block Plus does not overflow. The 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 PCIe Block Plus 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.

Receiver Flow Control Credits Available

The PCIe Block Plus core provides the User Application information about the state of the receiver buffer pool queues. This information represents the current space available for the Posted, Non-Posted, and Completion queues.

One Header Credit is equal to either a 3 or 4 DWORD TLP Header and one Data Credit is equal to 16 bytes of payload data. [Table 4-3](#) provides values on credits available immediately after `trn_lnk_up_n` assertion but before the reception of any TLP. If space available for any of the above categories is exhausted, the corresponding credit available signals will indicate a value of zero. Credits available return to initial values after the receiver has drained all TLPs.

Table 4-3: Transaction Receiver Credits Available Initial Values

| Credit Category | Signal Name | Maximum Credits (Hex) | Max Size (Decimal) |
|-------------------|----------------------|-----------------------|------------------------|
| Non-Posted Header | trn_rfc_nph_av[7:0] | 08 | 8 Headers |
| Non-Posted Data | trn_rfc_npd_av[11:0] | 000 | Unlimited ¹ |
| Posted Header | trn_rfc_ph_av[7:0] | 08 | 8 Headers |
| Posted Data | trn_rfc_pd_av[11:0] | 074 | 1856 Bytes |
| Completion Header | N/A | 00 | Unlimited ² |
| Completion Data | N/A | 000 | Unlimited ² |

1. Non-posted data size is limited to 128 bytes by the header credit availability.
2. Although the core advertises infinite completion credit, the actual buffer space in the core is 8 headers and 2048 bytes of payload.

The User Application can use the `trn_rfc_ph_av[7:0]`, `trn_rfc_pd_av[11:0]`, and `trn_rfc_nph_av[7:0]` signals to efficiently utilize and manage receiver buffer space available in the core and the core application.

PCI Express Endpoints have a unique requirement where the User Application must use advanced methods to prevent buffer overflows while requesting Non-Posted Read Requests from an upstream component. According to the specification, a PCI Express endpoint is required to advertise infinite storage credits for Completion Transactions in its receivers. This means that endpoints must internally manage Memory Read Requests transmitted upstream and not overflow the receiver when the corresponding Completions are received. The Endpoint core transmit User Application must track Completions received for previously requested Non-Posted Transactions to modulate the rate and size of non-posted requests and stay within the instantaneous Completion space available in the Endpoint receiver. For additional information, see [Appendix A, “Managing Receive-Buffer Space for Inbound Completions.”](#)

Accessing Configuration Space Registers

Registers Mapped Directly onto the Configuration Interface

The PCIe 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 |
|--------------------------|-----------|---|
| cfg_bus_number[7:0] | Output | Bus Number: Default value after reset is 00h. Refreshed whenever a Type 0 Configuration packet is received. |
| cfg_device_number[4:0] | Output | Device Number: Default value after reset is 00000b. Refreshed whenever a Type 0 Configuration packet is received. |
| cfg_function_number[2:0] | Output | Function Number: Function number of the core, hard wired to 000b. |
| cfg_status[15:0] | Output | Status Register: Status register from the Configuration Space Header. |
| cfg_command[15:0] | Output | Command Register: Command register from the Configuration Space Header. |
| cfg_dstatus[15:0] | Output | Device Status Register: Device status register from the PCI Express Extended Capability Structure. |
| cfg_dcommand[15:0] | Output | Device Command Register: Device control register from the PCI Express Extended Capability Structure. |
| cfg_lstatus[15:0] | Output | Link Status Register: Link status register from the PCI Express Extended Capability Structure. |
| cfg_lcommand[15:0] | 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 PCIe Block Plus 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[1] | Memory Address Space Decoder Enable |
| cfg_command[0] | IO Address Space Decoder Enable |

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 |
| 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) |

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

| Bit | Name |
|----------------|-------------------------------------|
| 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 Extended Device Control 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 Status 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 |
| 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 |

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

| Bit | Name |
|-----------------|------------------------------------|
| 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 Device Control 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 Status 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 2-7, page 28](#).

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 PCIe Block Plus 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.

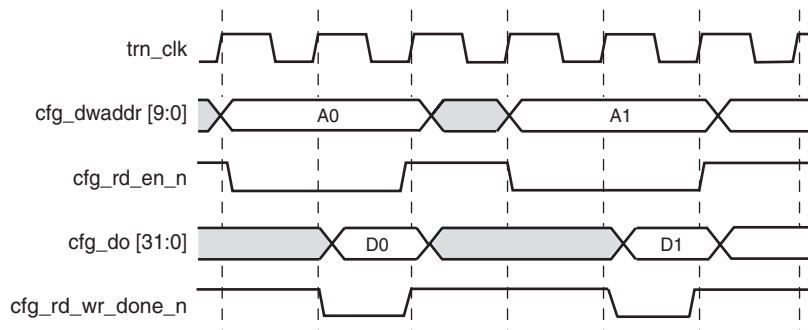


Figure 4-22: Example Configuration Space Access

Note that writing to the Configuration Space using `cfg_di[31:0]` and `cfg_wr_en_n` is not supported in this release. The PCIe Block Plus 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 PCIe Block Plus core 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 PCIe Block Plus core 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 PCIe 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 [Appendix A, “Managing Receive-Buffer Space for Inbound Completions,”](#) 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 Endpoint 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 2-8, page 30](#).

- 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 [Tables 4-11 and 4-12](#).

Table 4-11: User-indicated Error Signaling

| Reported Error | cfg_err_posted_n | Action |
|--------------------------|------------------|--|
| None | Don't care | No Action Taken |
| cfg_err_ur_n | 0 or 1 | 0: If enabled, a Non-Fatal Error Message is sent. 1: A Completion with a non-successful status is sent. |
| cfg_err_cpl_abort_n | 0 or 1 | 0: If enabled, a Non-Fatal Error message is sent. 1: A Completion with a non-successful status is sent. |
| cfg_err_cpl_timeout_n | Don't care | If enabled, a Non-Fatal Error Message is sent. |
| cfg_err_ecrc_n | Don't care | If enabled, a Non-Fatal Error Message is sent. |
| cfg_err_cor_n | Don't care | If enabled, a Correctable Error Message is sent. |
| cfg_err_cpl_unexpected_n | 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 (cfg_err_ur_n) | Completion Abort (cfg_err_cpl_abort_n) | Correctable Error (cfg_err_cor_n) | ECRC Error (cfg_err_ecrc_n) | Unexpected Completion (cfg_err_cpl_unexpect_n) | Value to Drive on (cfg_err_posted_n) | Drive Data on (cfg_err_tlp_cpl_header[47:0]) |
| | 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 Endpoint 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.

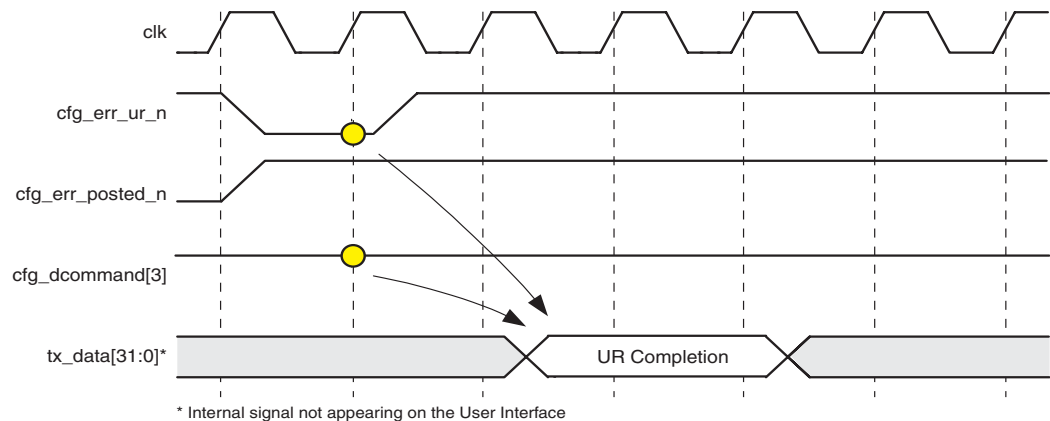


Figure 4-23: Signaling Unsupported Request for Non-Posted TLP

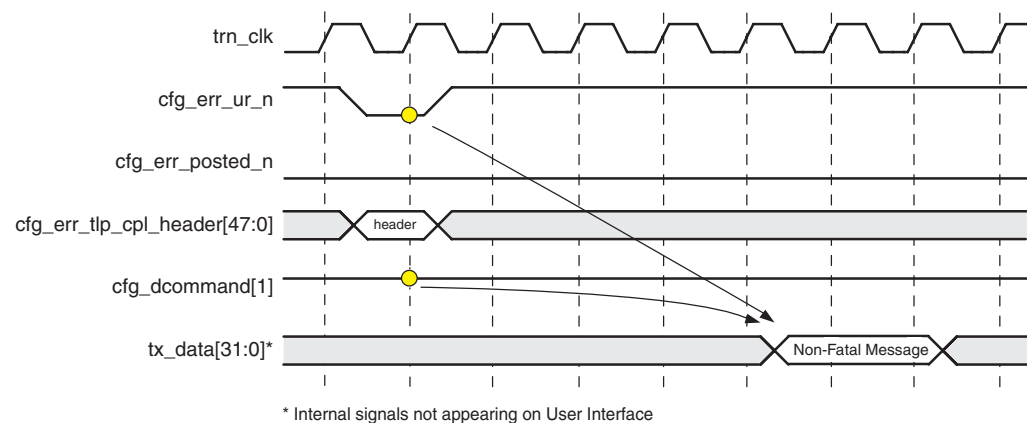


Figure 4-24: Signaling Unsupported Request for Posted TLP

Completion Timeouts

The PCIe 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 PCIe Block Plus core automatically reports Unexpected Completion in response to inbound Completions whose Completer 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. 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.

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 PCIe 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 PCIe Block Plus core 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 device 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 PCIe Block Plus core supports the conditions required for ASPM.

Programmed Power Management

To achieve considerable power savings on the PCI Express hierarchy tree, the PCIe Block Plus 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 PCIe Block Plus 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 PCIe Block Plus 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 PCIe Block Plus 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 PCIe Block Plus core to the PPM L3 state.

1. The PCIe Block Plus 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 Endpoint can 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 Endpoint 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 PCIe Block Plus core receives this TLP, it asserts `cfg_to_turnoff_n` to the User Application and starts polling the `cfg_turnoff_ok_n` input.
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 Endpoint core sends a PME_TO_Ack after approximately 250 nanoseconds.

Generating Interrupt Requests

The PCIe Block Plus core 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)

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. The User Application does not typically need to poll the MSI Enable bit unless it needs to know the method in which its interrupt requests are being conveyed to the Root Complex. Regardless of the type of interrupts being used, the user

initiates interrupt request 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 is ready to send the 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 4-25](#), the User Application asserts `cfg_interrupt_n` only if `cfg_interrupt_rdy_n` is asserted. If the MSI Enable bit in the MSI Control register is set to 1, the core sends a MSI Memory Write TLP.
- The User Application only needs to assert `cfg_interrupt_n` for one cycle and then deassert it. No further action is required in MSI mode. Note that if the user asserts `cfg_interrupt_n` for multiple cycles, the core continually sends MSI request s until the user deasserts `cfg_interrupt_n`.

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 the Root Complex through configuration writes to the MSI Capability structure. The type of TLP sent, whether it is 32-bit addressable or 64-bit addressable, depends on if a non-zero value has been written into the Upper Address field in the MSI capability structure. Coming out of reset this field is all zeroes, and if the Root Complex does not change it by writing a value to the register, MSI requests are sent as 32-bit addressable Memory Write TLPs as the specification forbids 64-bit addressable Memory Write TLPs with all zeroes in the upper 32 bits of the address.

Legacy Interrupt Mode

- As shown in [Figure 4-25](#), the User Application asserts `cfg_interrupt_n` only if `cfg_interrupt_rdy_n` is asserted. If the MSI Enable bit in the MSI Control register is set to 0 and the Interrupt Disable bit in the PCI Command register is set to 0, the core will send an assert interrupt A message (Assert_INTA).
- The User Application must not toggle `cfg_interrupt_n` while `cfg_interrupt_rdy_n` is deasserted by the core. It is up to the User Application to determine when the interrupt has been serviced. The core will send a deassert interrupt A message (Deassert_INTA).

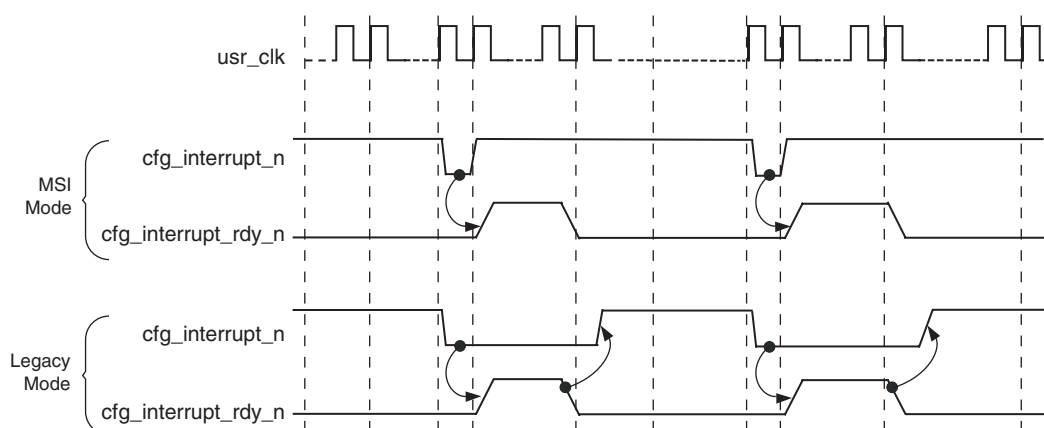


Figure 4-25: Requesting Interrupt Service: MSI and Legacy Mode

Link Training: 4-Lane and 8-Lane PCIe Block Plus Cores

The PCIe Block Plus 4-lane and 8-lane Endpoint cores are capable of operating at less than the maximum lane width as required by the *PCI Express Base Specification*. There are two cases that cause the core to operate at less than its specified maximum lane width, described below.

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 4-26. 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.

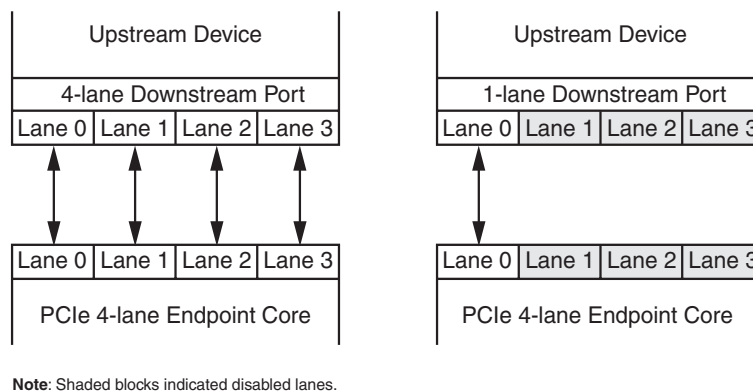


Figure 4-26: Scaling of 4-lane Endpoint Core from 4-lane to 1-lane Operation

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.

Clocking and Reset of the PCI Express Block Plus Core

Reset

The PCIe Block Plus core 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 GTP 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 PCIe:

- **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 PCIe Block Plus 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 PCIe Block Plus 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 PCIe Block:** Loses lock, indicating an issue with the stability of the clock input.
- **GTP Lock Lost:** 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—read the FPGA device data sheet carefully to understand the requirements for interfacing to such signals.

Clocking

The PCIe 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 PCIe Block Plus 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: Xilinx recommends that designers use synchronous clocking when using the PCIe Block Plus 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 4-27](#) and [4-29](#) 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 4-28](#) and [4-30](#). 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 4-27](#) and [4-28](#).

[Figures 4-28](#) and [4-30](#) 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.

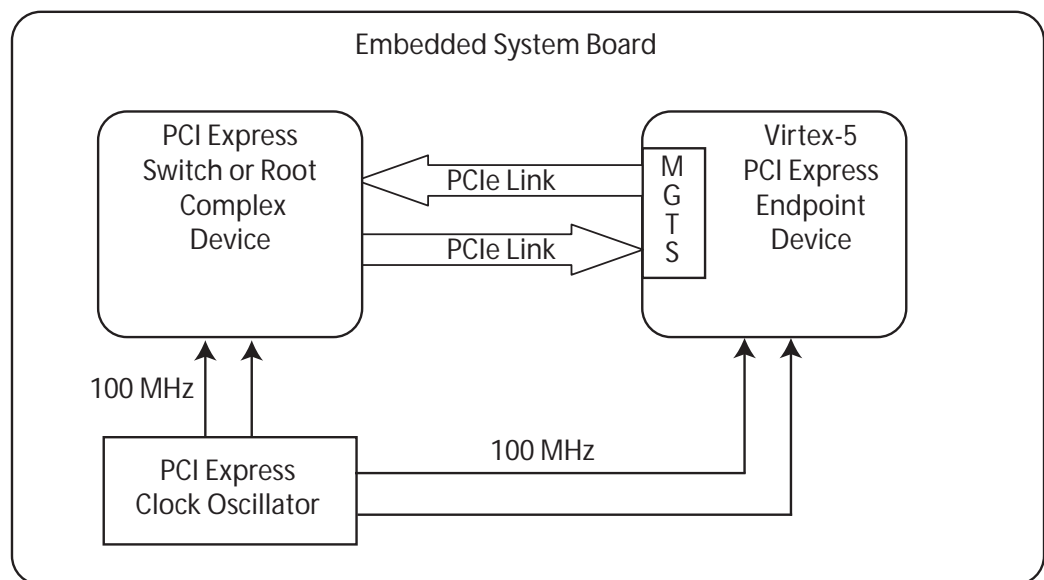


Figure 4-27: Embedded System Using 100 MHz Reference Clock

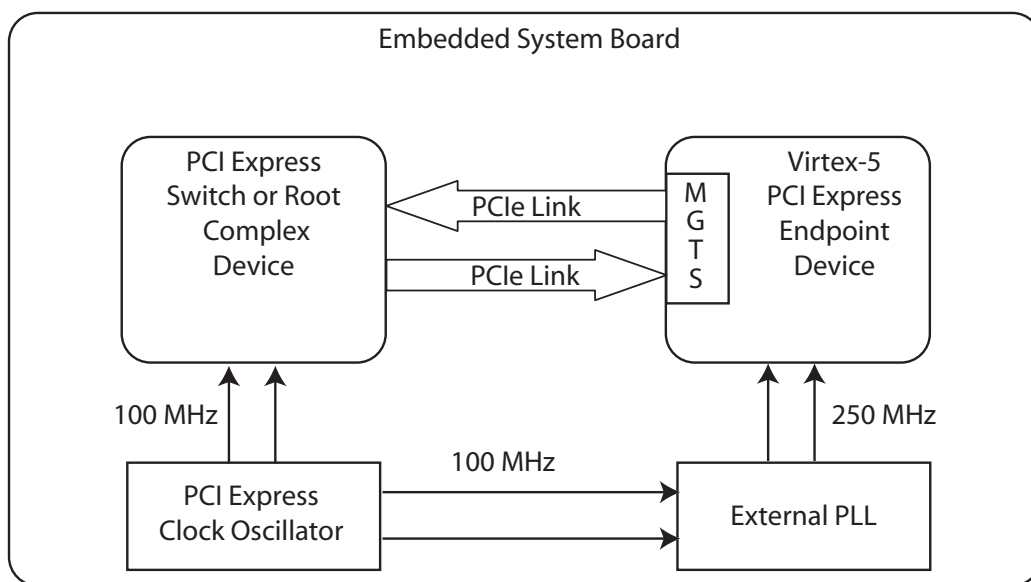


Figure 4-28: Embedded System Using 250 MHz Reference Clock

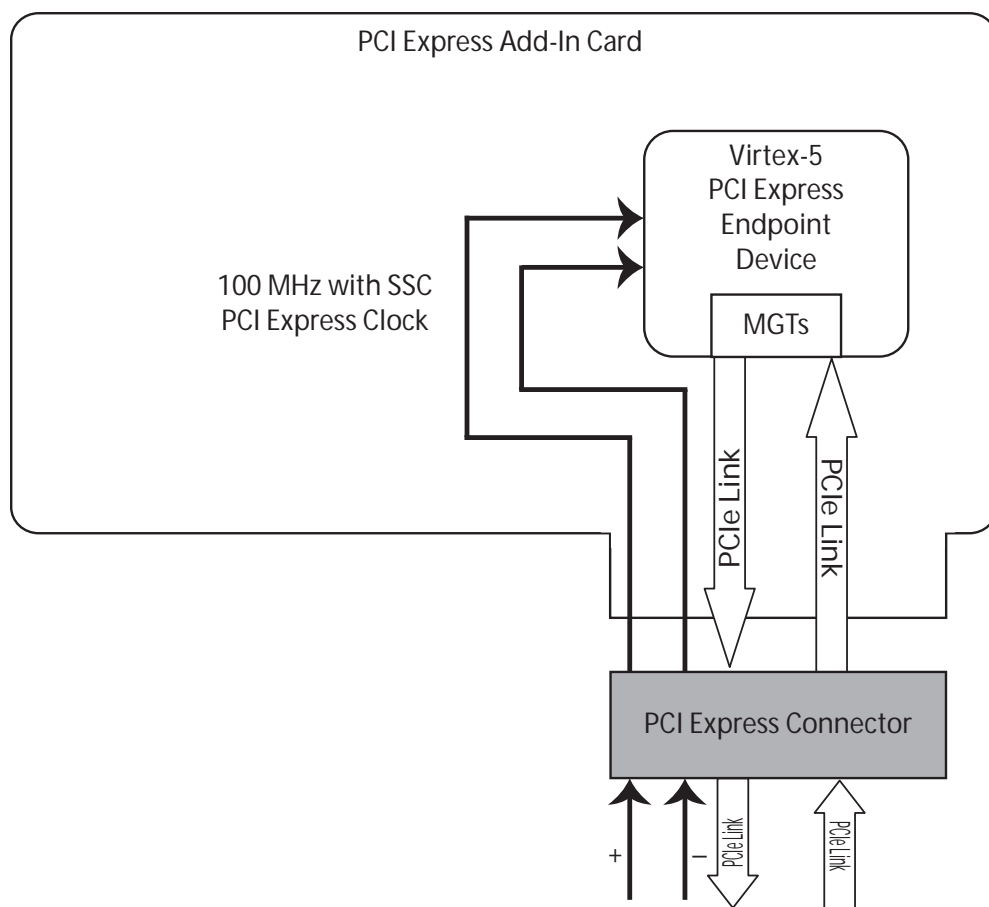


Figure 4-29: Open System Add-In Card Using 100 MHz Reference Clock

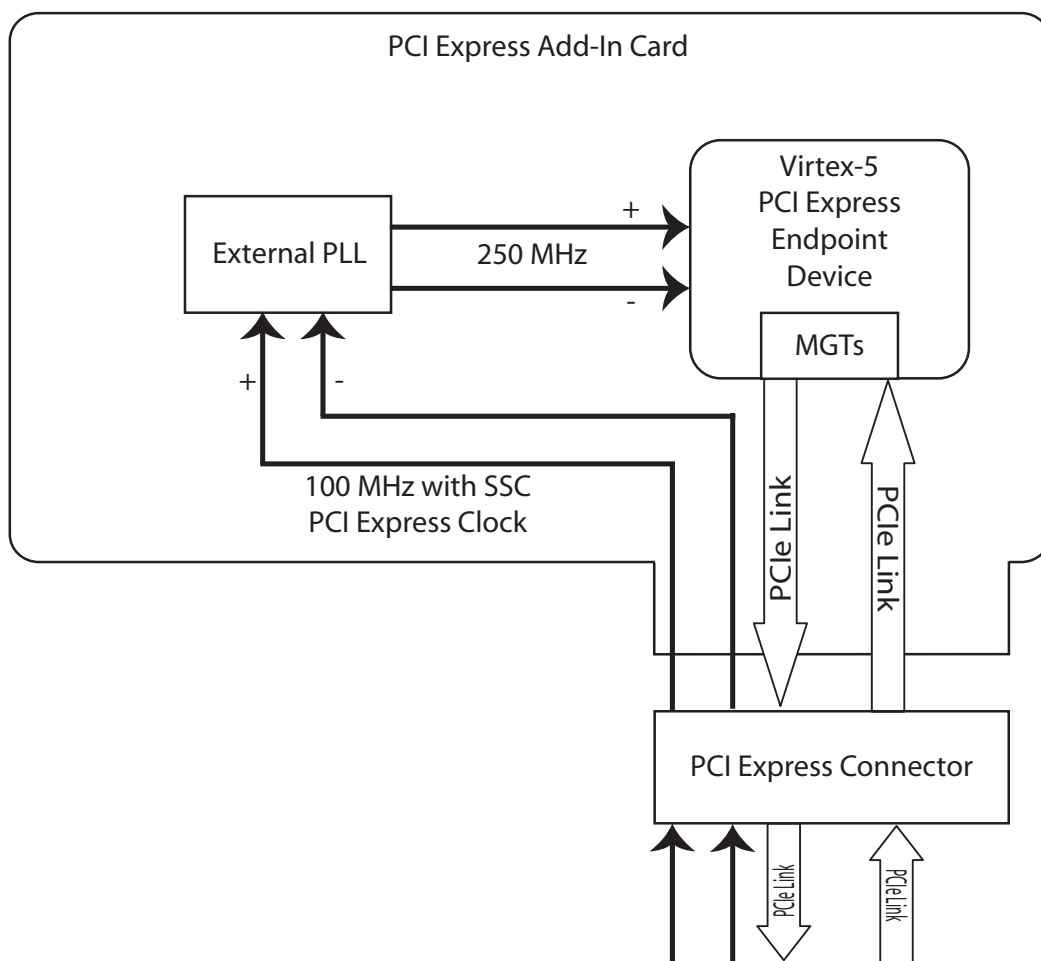


Figure 4-30: Open System Add-In Card Using 250 MHz Reference Clock

Core Constraints

The PCIe Block Plus solutions require the specification of timing and other physical implementation constraints to meet specified PCI Express performance requirements. These constraints are provided with the Endpoint Solutions in a 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 PCIe 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 PCIe Block Plus core share 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 PCIe 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 PCIe 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 PCIe 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" ;
```

See “[Clocking and Reset of the PCI Express Block Plus Core,](#)” page 87 for detailed information about reset and clock requirements.

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/pcie_gt_wrapper_i/GTPD[0].GTP_i"
LOC = GTP_DUAL_X0Y0;
```

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/pcie_gt_wrapper_i/gt_tx_data_k_reg_0" AREA_GROUP =
"GRP0" ;
```

Core Timing Constraints

Timing constraints are provided for all PCIe 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/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/pcie_gt_wrapper_i/gt_tx_elec_idle_reg*" MAXDELAY = 1.0 ns;
```

Relocating the PCI Express 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. If the core is moved, the relative location of all transceivers and clocking resources should be maintained to ensure timing closure.

Managing Receive-Buffer Space for Inbound Completions

The PCI Express Base Specification requires all Endpoints to advertise infinite Flow Control credits for received Completions to their link partners. This means that an Endpoint must only transmit Non-Posted Requests for which it has space to accept Completion responses. This section describes how a User Application can manage the receive-buffer space in the PCIe Block Plus core to fulfill this requirement.

General Considerations and Concepts

Completion Space

[Table A-1](#) defines the completion space reserved in the receive buffer by the core.

Table A-1: Receiver Buffer Completion Space

| Cpl. Header Credits (Total_CplH) | Cpl. Data Credits (Total_CplD) |
|-------------------------------------|-----------------------------------|
| 8 | 128 |

The receiver accommodates storage space for a maximum of eight Completion TLPs, where each TLP can include an ECRC field. Total payload associated with these Completions must not exceed 2048 bytes.

Maximum Request Size

A Memory Read cannot request more than the value stated in Max_Request_Size, which is given by Configuration bits `cfg_dcommand[14:12]` as defined in [Table A-2](#). If the User Application chooses not to read the Max_Request_Size value, it must use the default value of 128 bytes.

Table A-2: Max Request Size Settings

| cfg_dcommand[14:12] | Max_Request_Size | | | |
|---------------------|------------------|-----|----|---------|
| | Bytes | DW | QW | Credits |
| 000b | 128 | 32 | 16 | 8 |
| 001b | 256 | 64 | 32 | 16 |
| 010b | 512 | 128 | 64 | 32 |

Table A-2: Max Request Size Settings

| | | | | |
|-----------|----------|------|-----|-----|
| 011b | 1024 | 256 | 128 | 64 |
| 100b | 2048 | 512 | 256 | 128 |
| 101b | 4096 | 1024 | 512 | 256 |
| 110b–111b | Reserved | | | |

Read Completion Boundary

A Memory Read can be answered with multiple Completions, which when put together return all requested data. To make room for packet-header overhead, the User Application must allocate enough space for the maximum number of Completions that might be returned.

To make this process easier, the *Base Specification* quantizes the length of all Completion packets such that each must start and end on a naturally aligned Read Completion Boundary (RCB), unless it services the starting or ending address of the original request. The value of RCB is determined by Configuration bit `cfg_lcommand[3]` as defined in [Table A-3](#). If the User Application chooses not to read the RCB value, it must use the default value of 64 bytes.

Table A-3: Read Completion Boundary Settings

| cfg_lcommand[3] | Read Completion Boundary | | | |
|-----------------|--------------------------|----|----|---------|
| | Bytes | DW | QW | Credits |
| 0 | 64 | 16 | 8 | 4 |
| 1 | 128 | 32 | 16 | 8 |

When calculating the number of Completion credits a Non-Posted Request requires, the user must determine how many RCB-bounded blocks the Completion response can require; this is the same as the number of Completion Header credits required.

Methods of Managing Completion Space

A User Application can choose one of four methods to manage receive-buffer Completion space, as listed in Table B-3. For convenience, this discussion refers to these methods as LIMIT_FC, PACKET_FC, RCB_FC and DATA_FC. Each has advantages and disadvantages that the designer needs to consider when developing the User Application.

Table A-4: Managing Receive Completion Space Methods

| Method | Description | Advantage | Disadvantage |
|-----------|--|---|---|
| LIMIT_FC | Limit the total number of outstanding NP Requests | Simplest method to implement in user logic | Much Completion capacity goes unused |
| PACKET_FC | Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-packet basis | Relatively simple user logic; finer allocation granularity means less wasted capacity than LIMIT_FC | As with LIMIT_FC, credits for an NP are still tied up until the Request is completely satisfied |
| RCB_FC | Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-RCB basis | Ties up credits for less time than PACKET_FC | More complex user logic than LIMIT_FC or PACKET_FC |
| DATA_FC | Track the number of outstanding CplH and CplD credits; allocate and deallocate on a per-RCB basis | Lowest amount of wasted capacity | Most complex user logic |

The LIMIT_FC Method

The LIMIT_FC method is the simplest to implement. The User Application assesses the maximum number of outstanding Non-Posted Requests allowed at one time, MAX_NP. To calculate this value, perform the following steps:

- Determine the number of CplH credits required by a Max_Request_Size packet:

$$\text{Max_Header_Count} = \text{ceiling}(\text{Max_Request_Size} / \text{RCB})$$
- Determine the greatest number of maximum-sized Completions supported by the CplD credit pool:

$$\text{Max_Packet_Count_CplD} = \text{floor}(\text{CplD} / \text{Max_Request_Size})$$
- Determine the greatest number of maximum-sized Completions supported by the CplH credit pool:

$$\text{Max_Packet_Count_CplH} = \text{floor}(\text{CplH} / \text{Max_Header_Count})$$
- Use the *smaller* of the two quantities from steps 2 and 3 to obtain the maximum number of outstanding Non-Posted requests:

$$\text{MAX_NP} = \text{min}(\text{Max_Packet_Count_CplH}, \text{Max_Packet_Count_CplD})$$

With knowledge of MAX_NP, the User Application can load a register NP_PENDING with zero at reset and make sure it always stays with the range 0 to MAX_NP. When a Non-Posted Request is transmitted, NP_PENDING decrements by one. When *all* Completions for an outstanding NP Request are received, NP_PENDING increments by one.

Although this method is the simplest to implement, it potentially wastes the most receiver space because an entire Max_Request_Size block of Completion credit is allocated for each Non-Posted Request, regardless of actual request size. The amount of waste becomes greater when the User Application issues a larger proportion of short Memory Reads (on the order of a single DWORD), I/O Reads and I/O Writes.

The PACKET_FC Method

The PACKET_FC method allocates blocks of credit in finer granularities than LIMIT_FC, using the receive Completion space more efficiently with a small increase in user logic.

Start with two registers, CPLH_PENDING and CPLD_PENDING, (loaded with zero at reset), and then perform the following steps:

1. When the User Application needs to send an NP request determine the potential number of CplH and CplD credits it can require:

$$\text{NP_CplH} = \text{ceiling}[(\text{Start_Address mod RCB}) + \text{Request_Size}] / \text{RCB}]$$

$$\text{NP_CplD} = \text{ceiling}[(\text{Start_Address mod 16 bytes}) + \text{Request_Size}] / 16 \text{ bytes}]$$

(except I/O Write, which returns zero data)

The modulo and ceiling functions ensure that any fractional RCB or credit blocks are rounded up. For example, if a Memory Read requests 8 bytes of data from address 7Ch, the returned data can potentially be returned over two Completion packets (7Ch-7Fh, followed by 80h-83h). This would require two RCB blocks and two data credits.

2. Check the following:

$$\text{CPLH_PENDING} + \text{NP_CplH} \leq \text{Total_CplH (from Table B-1)}$$

$$\text{CPLD_PENDING} + \text{NP_CplD} \leq \text{Total_CplD (from Table B-1)}$$

3. If both inequalities are true, transmit the Non-Posted Request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD. For each NP Request transmitted, keep NP_CplH and NP_CplD for later use.
4. When all Completion data is returned for an NP Request, decrement CPLH_PENDING and CPLD_PENDING accordingly.

This method is less wasteful than LIMIT_FC but still ties up all of an NP Request's Completion space until the *entire* request is satisfied. RCB_FC and DATA_FC provide finer de-allocation granularity at the expense of more logic.

The RCB_FC Method

The RCB_FC method allocates and de-allocates blocks of credit in RCB granularity. Credit is freed on a per-RCB basis.

As with PACKET_FC, start with two registers, CPLH_PENDING and CPLD_PENDING (loaded with zero at reset).

1. Calculate the number of data credits per RCB:

$$\text{CplD_PER_RCB} = \text{RCB} / 16 \text{ bytes}$$

2. When the User Application needs to send an NP request, determine the potential number of CplH credits it may require. Use this to allocate CplD credits with RCB granularity:

$$\text{NP_CplH} = \text{ceiling}[(\text{Start_Address mod RCB}) + \text{Request_Size}) / \text{RCB}]$$

$$\text{NP_CplD} = \text{NP_CplH} \times \text{CplD_PER_RCB}$$
3. Check the following:

$$\text{CPLH_PENDING} + \text{NP_CplH} \leq \text{Total_CplH}$$

$$\text{CPLD_PENDING} + \text{NP_CplD} \leq \text{Total_CplD}$$
4. If both inequalities are true, transmit the Non-Posted Request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD.
5. At the start of each incoming Completion, or when that Completion begins at or crosses an RCB without ending at that RCB, decrement CPLH_PENDING by 1 and CPLD_PENDING by CplD_PER_RCB. Any Completion can cross more than one RCB. The number of RCB crossing can be calculated by:

$$\text{RCB_CROSSED} = \text{ceiling}[(\text{Lower_Address mod RCB}) + \text{Length}) / \text{RCB}]$$

Lower_Address and Length are fields that can be parsed from the Completion header. Alternatively, a designer can load a register CUR_ADDR with Lower_Address at the start of each incoming Completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over.

This method is less wasteful than PACKET_FC but still gives us an RCB granularity. If a User Application transmits I/O requests, the User Application could adopt a policy of only allocating one CplD credit for each I/O Read and zero CplD credits for each I/O Write. The User Application would have to match each incoming Completion's Tag with the Type (Memory Write, I/O Read, I/O Write) of the original NP Request.

The DATA_FC Method

The DATA_FC method provides the finest allocation granularity at the expense of logic.

As with PACKET_FC and RCB_FC, start with two registers, CPLH_PENDING and CPLD_PENDING (loaded with zero at reset).

1. When the User Application needs to send an NP request, determine the potential number of CplH and CplD credits it may require:

$$\text{NP_CplH} = \text{ceiling}[(\text{Start_Address mod RCB}) + \text{Request_Size}) / \text{RCB}]$$

$$\text{NP_CplD} = \text{ceiling}[(\text{Start_Address mod 16 bytes}) + \text{Request_Size}) / 16 \text{ bytes}]$$

(except I/O Write, which returns zero data)
2. Check the following:

$$\text{CPLH_PENDING} + \text{NP_CplH} \leq \text{Total_CplH}$$

$$\text{CPLD_PENDING} + \text{NP_CplD} \leq \text{Total_CplD}$$
3. If both inequalities are true, transmit the Non-Posted Request, increase CPLH_PENDING by NP_CplH and CPLD_PENDING by NP_CplD.
4. At the start of each incoming Completion, or when that Completion begins at or crosses an RCB without ending at that RCB, decrement CPLH_PENDING by 1. The number of RCB crossings can be calculated by:

$$\text{RCB_CROSSED} = \text{ceiling}[(\text{Lower_Address mod RCB}) + \text{Length}) / \text{RCB}]$$

Lower_Address and Length are fields that can be parsed from the Completion header. Alternatively, a designer can load a register CUR_ADDR with Lower_Address at the start of each incoming Completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over.

5. At the start of each incoming Completion, or when that Completion begins at or crosses at a naturally aligned credit boundary, decrement CPLD_PENDING by 1. The number of credit-boundary crossings is given by:

$$\text{DATA_CROSSED} = \text{ceiling}[(\text{Lower_Address mod 16 B}) + \text{Length}) / 16 \text{ B}]$$

Alternatively, a designer can load a register CUR_ADDR with Lower_Address at the start of each incoming Completion, increment per DW or QW as appropriate, then count an RCB whenever CUR_ADDR rolls over each 16-byte address boundary.

This method is the least wasteful but requires the greatest amount of user logic. If even finer granularity is desired, the user can scale the Total_CplD value by 2 or 4 to get the number of Completion QWORDS or DWORDs, respectively, and adjust the data calculations accordingly.

PCI Express Endpoint PIO 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. PCI Express endpoints 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 PCIe Block Plus core generated by the CORE Generator. This design 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 PCI Express Endpoint, PCI Express Endpoint Block Plus, and PCI Express PIPE Endpoint solutions. This appendix represents all the PCI Express solutions generically using the name PCI Express Endpoint.

System Overview

The PIO design is a simple target-only application that interfaces with the PCI Express Endpoint 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 Xilinx PCI Express 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 B-1 illustrates the PCI Express system architecture components, consisting of a Root Complex, a PCI Express switch device, and a PCI Express Endpoint. 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.

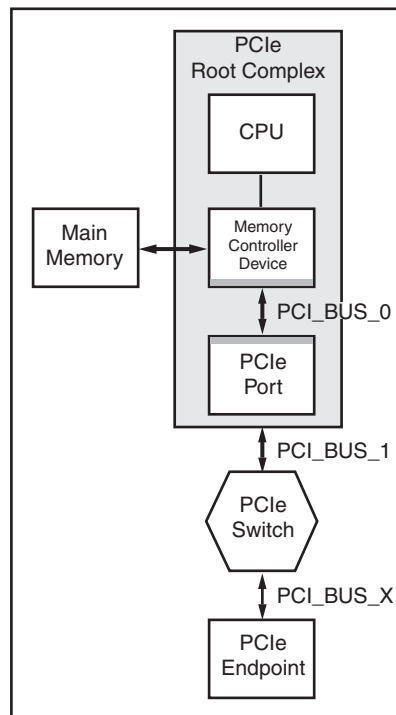


Figure B-1: PCI Express System Overview

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 PCI Express Endpoint core. 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 PCI Express Endpoint 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 IO Space BAR
- One 64-bit addressable Memory Space BAR
- One 32-bit Addressable Memory Space BAR
- One Expansion ROM BAR.

Users can 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 PCI Express Endpoint 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 B-1](#).

Table B-1: TLP Traffic Types

| Block RAM | TLP Transaction Type | Default BAR | <code>trn_rbar_hit_n[6:0]</code> |
|-----------|--|-------------|----------------------------------|
| ep_mem0 | IO TLP transactions | 0 | 111_1110b |
| ep_mem1 | 32-bit address Memory TLP transactions | 1 | 111_1101b |
| ep_mem2 | 64-bit address Memory TLP transactions | 2-3 | 111_0011b |
| ep_mem3 | 32-bit address Memory TLP transactions destined for EROM | Exp. ROM | 011_1111b |

Changing CORE Generator Default BAR Settings

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

- The example design supports one IO space BAR, two 32-bit Memory spaces (one of which must 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.

Although there are limitations to the PIO design, Verilog 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 107 and “Transmit Path,” page 109.

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 PCI Express 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 PCI Express core receives a Memory or IO Write TLP, the TLP destination address and transaction type are compared with the values in the PCI Express core BARs. If the TLP passes this comparison check, the PCI Express Endpoint 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 RX State Machine processes the incoming Write TLP and extracts the TLPs 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 PCI Express 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 PCI Express Endpoint core receives a Memory or IO Read TLP, the TLP destination address and transaction type are compared with the values programmed in the PCI Express core BARs. If the TLP passes this comparison check, the PCI Express Endpoint 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 prior to asserting the read enable request. For example, if a Memory Read 32 Request TLP is received by the PCI Express core targeting the Expansion ROM BAR, the core passes the TLP to the PIO design and asserts `trn_rbar_hit_n[6]`. 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[6]` instructs the PIO memory read controller to access the EROM 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 the `trn_rdst_rdy_n` signal, causing the Receive TRN interface to stall receiving any further TLPs until the internal Memory Read controller completes the read to the block RAM and generates the completion. 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.

PIO File Structure

Table B-2 defines the PIO design file structure. Note that based on the specific PCI Express 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 PCI Express cores 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 B-2: PIO Design File Structure

| File | Description |
|---------------------|--------------------------------|
| PIO.v | Top-level design wrapper |
| PIO_EP.v | PIO application module |
| PIO_TO_CTRL.v | PIO turn-off controller module |
| PIO_32.v | 32b interface macro define |
| PIO_64.v | 64b macro define |
| PIO_32_RX_ENGINE.v | 32b Receive engine |
| PIO_32_TX_ENGINE.v | 32b Transmit engine |
| PIO_64_RX_ENGINE.v | 64b Receive engine |
| PIO_64_TX_ENGINE.v | 64b Transmit engine |
| PIO_EP_MEM_ACCESS.v | Endpoint memory access module |
| EP_MEM.v | Endpoint memory |

Figure B-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.

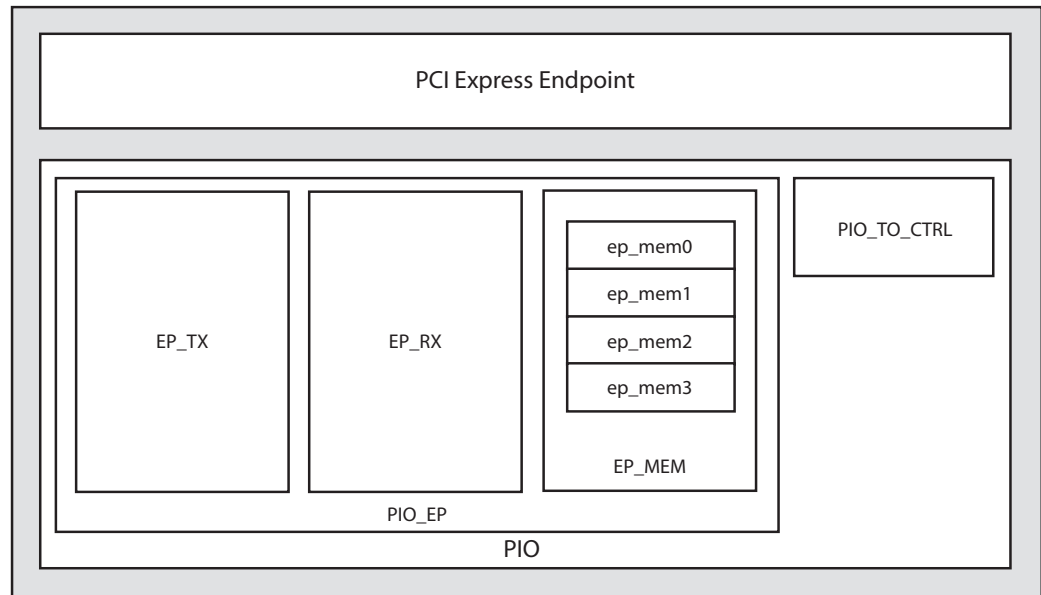


Figure B-2: PIO Design Components

PIO Application

Figures B-3 and B-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 PCI Express 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 PCI Express Endpoint LogiCore Transaction (trn) and Configuration (cfg) interfaces.

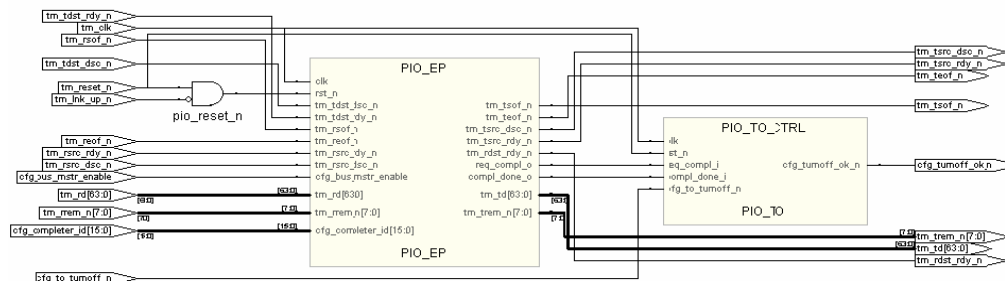


Figure B-3: PIO 64-bit Application

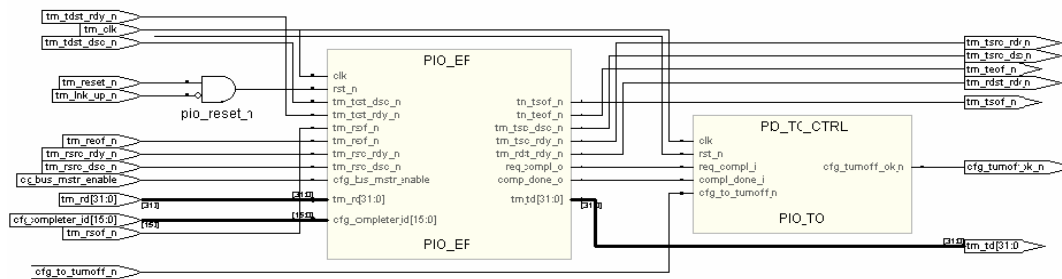


Figure B-4: PIO 32-bit Application

Receive Path

Figure B-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 PCI Express LogiCore Transaction Receive (trn_r*) interface.

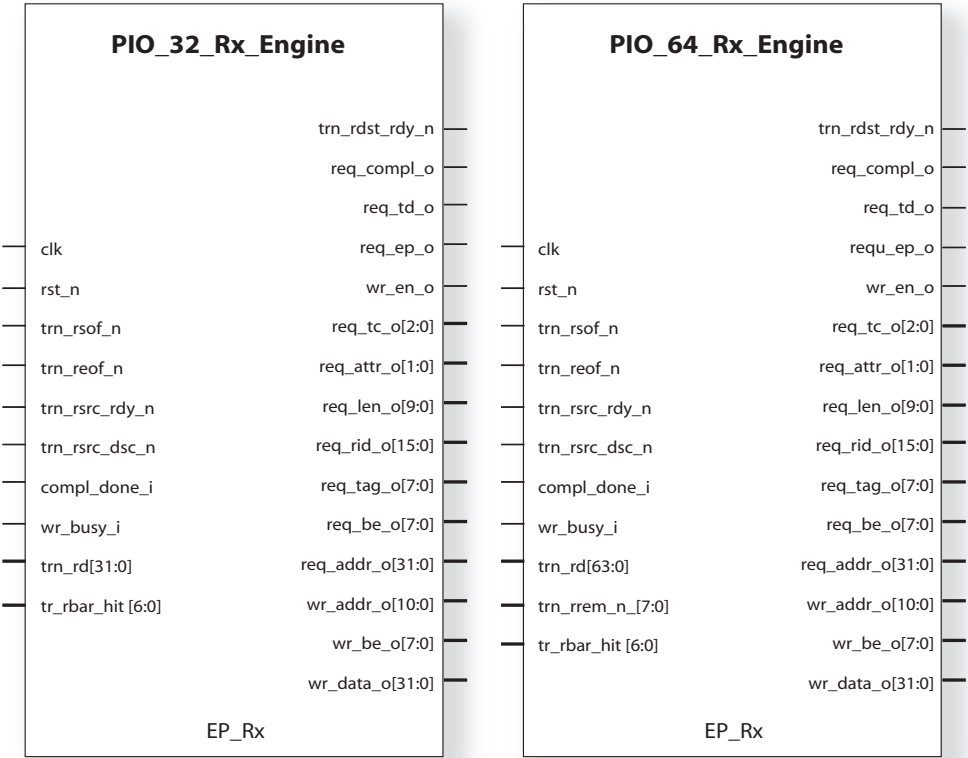


Figure B-5: Rx Engines

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 B-3](#).

Table B-3: 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 |
| 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 B-4](#).

Table B-4: 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 PCI Express Endpoint 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 B-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 PCI Express core Transaction Transmit (trn_r*) interface.

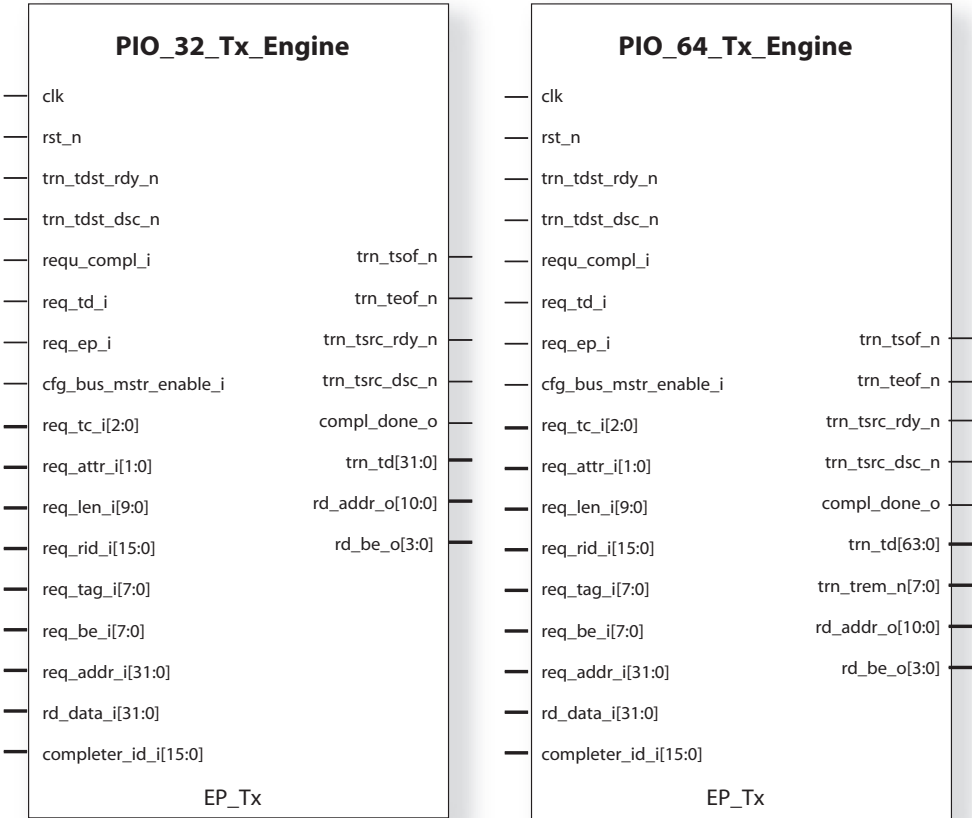


Figure B-6: Tx Engines

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 B-5.

Table B-5: 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 B-5: 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 B-7 displays the `PIO_EP_MEM_ACCESS` module. This module contains the Endpoint memory space.

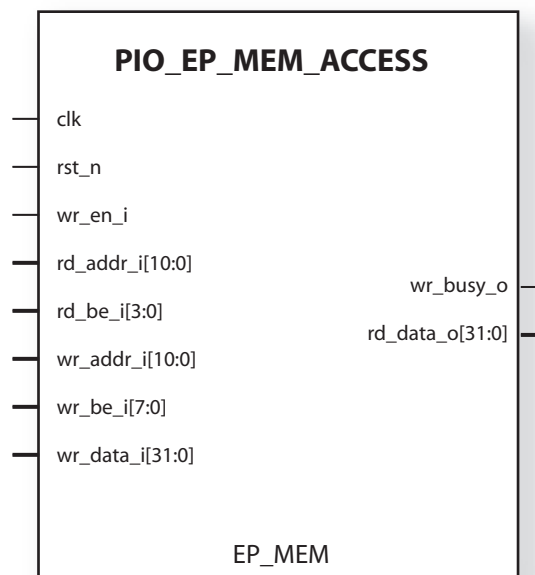


Figure B-7: EP Memory Access

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 B-6. While the memory controller is processing the write, it asserts the `wr_busy_o` output indicating it is busy.

Table B-6: 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 B-7**. After the read request is processed, the data is returned on `rd_data_o[31:0]`.

Table B-7: 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 B-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.

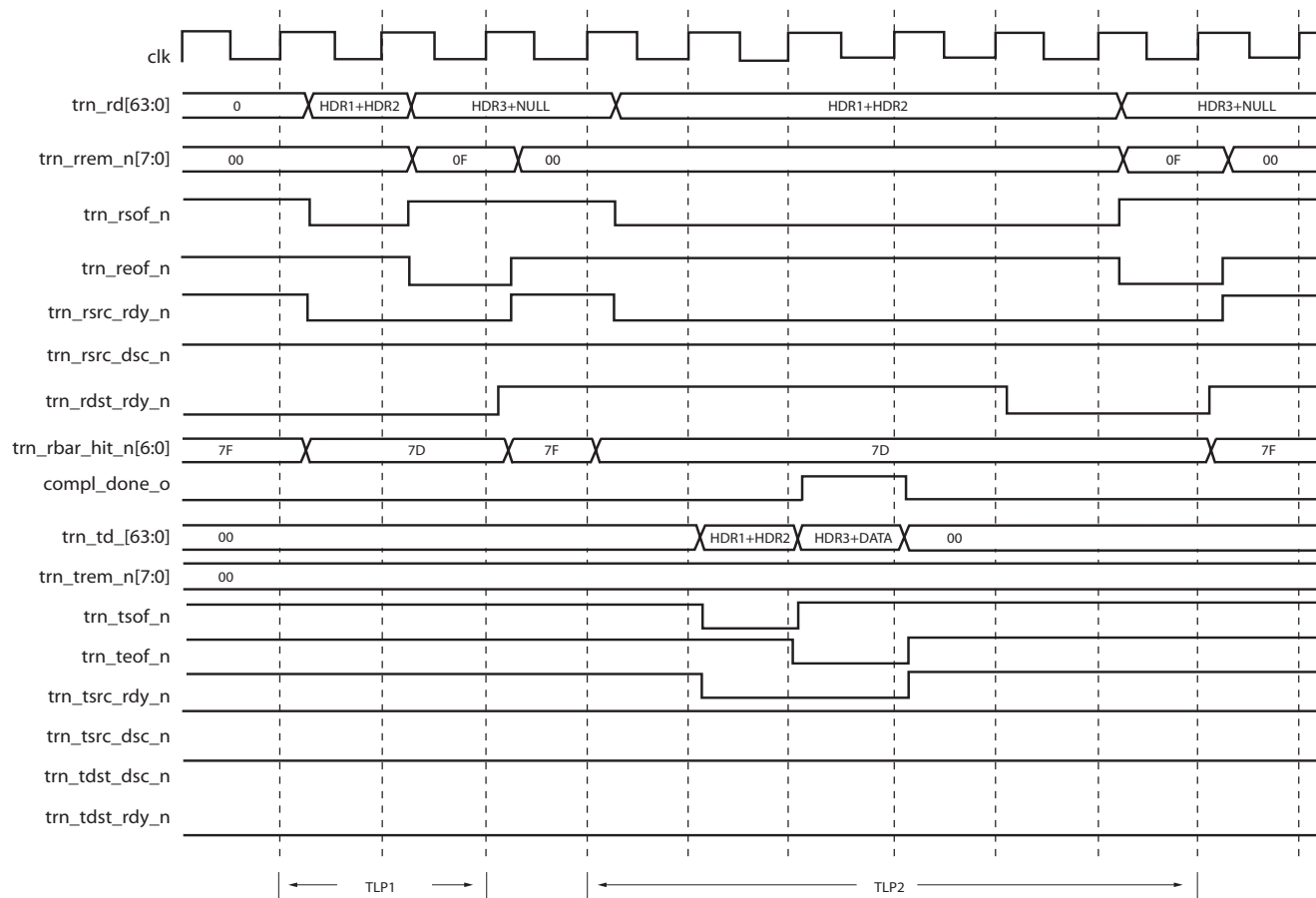


Figure B-8: Back-to-Back Read Transactions

PIO Write Transaction

Figure B-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.

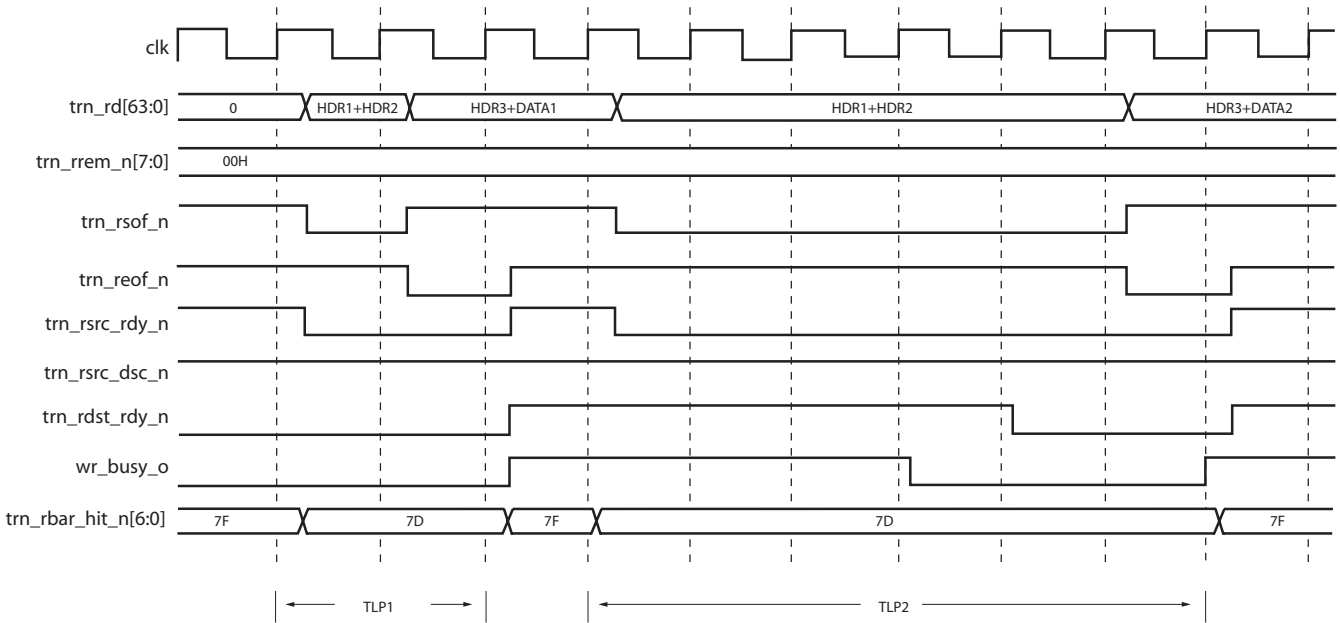


Figure B-9: Back-to-Back Write Transactions

Device Utilization

Table B-8 shows the PIO design FPGA resource utilization.

Table B-8: PIO Design FPGA Resources

| Resources | Utilization |
|------------|-------------|
| LUTs | 300 |
| Flip-Flops | 500 |
| block RAMs | 4 |

Summary

The PIO design demonstrates the PCI Express Endpoint core 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 can leverage standard operating system utilities that enable generation of read and write transactions to the target space in the reference design.

PCI Express Endpoint Downstream Model Test Bench

The PCI Express Endpoint 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 PCI Express 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 PCI Express Endpoint, PCI Express Endpoint Block Plus, and PCI Express PIPE Endpoint solutions. This appendix represents all the PCI Express solutions generically using the name PCI Express Endpoint.

Source code for the PCI Express Downstream Port Model is included to provide the model for a starting point for your test bench. All the significant work for initializing the PCI Express 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 a PCI Express test bench infrastructure.

The PCI Express Downstream Port Model consists of the following:

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

[Figure C-1](#) illustrates the illustrates the PCI Express Downstream Port Model coupled with the PIO design.

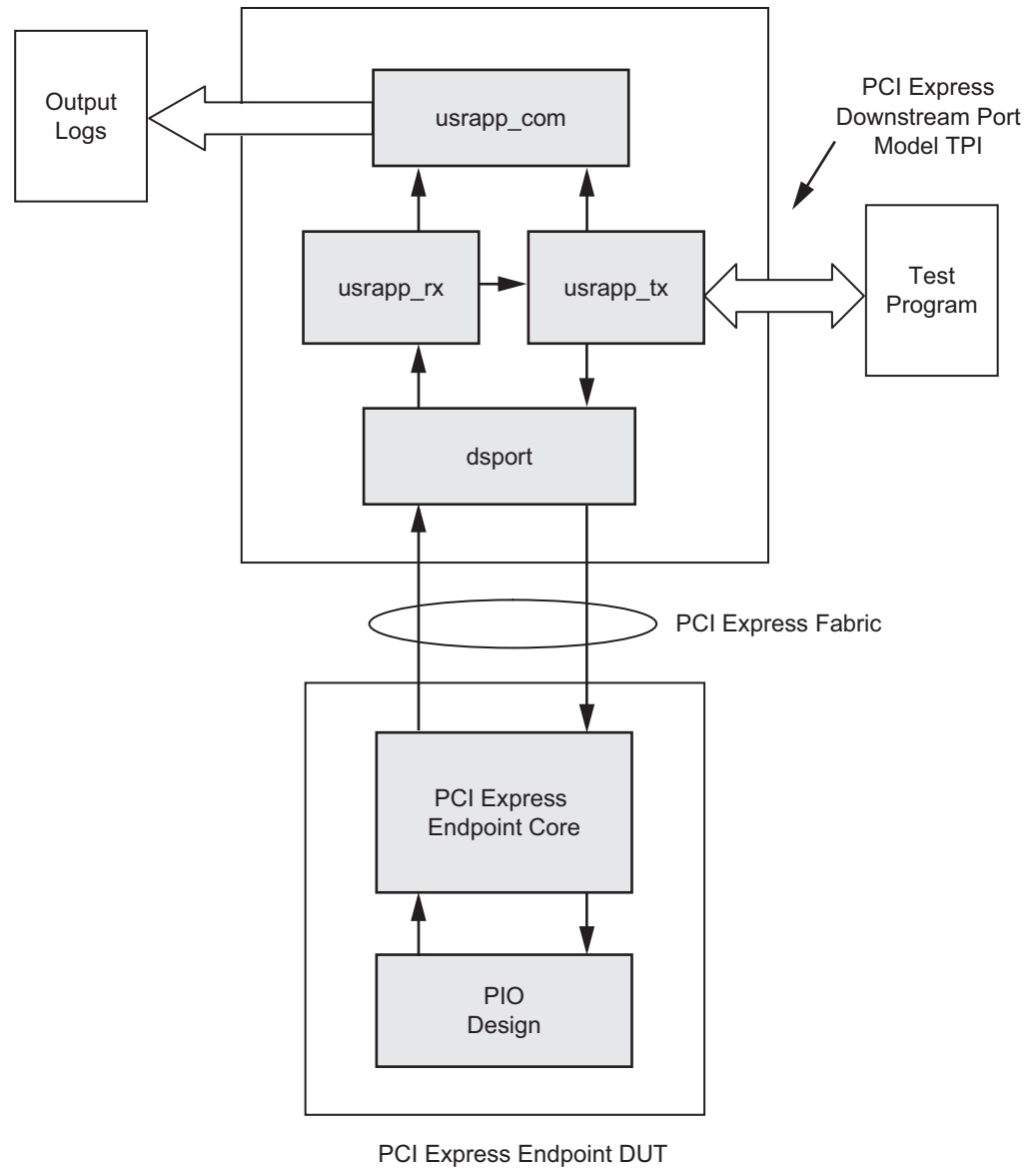


Figure C-1: PCI Express Downstream Port Model and Top-level Endpoint

Architecture

The PCI Express Downstream Port Model consists of the following blocks as shown in Figure C-1.

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

The `usrapp_tx` and `usrapp_rx` blocks interface with the `dsport` block for transmission and reception of TLPs to/from the PCI Express Endpoint. The PCI Express Endpoint consists of the PCI Express Endpoint 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 PCI Express endpoint device. In turn, the PCI Express endpoint device transmits TLPs across the PCI Express link to the `dsport` block, which are subsequently passed to the `usrapp_rx` block. The `dsport` and PCI Express 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.

Simulating the Design

Three simulation script files are provided with the model to facilitate simulation with VCS, NCVerilog, and ModelSim simulators:

- `simulate_vcs.sh`
- `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 PCI Express Downstream Port Model are provided in the *PCIe Endpoint Block Plus Getting Started Guide*.

Test Selection

The test model used for the PCI Express 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 NCVerilog 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. [Table C-1](#) defines all the tests provided with PCI Express Downstream Port Model.

Table C-1: Tests Provided with Downstream Port Model

| Test Name | Description |
|---------------------------------|---|
| <code>sample_smoke_test0</code> | 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. |

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

| | |
|---------------------------|---|
| sample_smoke_test1 | 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 | 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 | 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 | 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 | 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 | 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 C-1: Tests Provided with Downstream Port Model (Continued)

| | |
|--|--|
| pio_timeoutFailureExpected | 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) | 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. |

Waveform Dumping

The PCI Express 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`, is used to start the NCVerilog simulator can indicate to the PCI Express Downstream Port Model that the waveform should be saved to file by using the following command line:

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

This same mechanism is used for VCS and ModelSim. The dump file is in each simulator's native format, as defined in Table C-2.

Table C-2: Simulator Dump File Format

| Simulator | Dump File Format |
|-----------|------------------|
| VCS | .vpd |
| NCVerilog | .trn |
| Modelsim | .vcd |

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 PCI Express 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 can 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 different classes of tests that are supported by the PCI Express Downstream Port Model: sequential tests and parallel tests. Sequential tests are 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. Parallel tests are tests involving more than one process thread. The test `sample_smoke_test1` is an example of a parallel test with two process threads.

Sequential tests are very useful when verifying behavior that have events with a known order. 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 PCI Express 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 can include bus-mastering functionality).

Test Description

The PCI Express 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 PCI Express Downstream Port Model tests should follow the same six steps described as follows:

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 PCI Express Downstream Port Model and the PCI Express endpoint device
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 BAR1 and to the EROM space (BAR6). 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 PCI Express 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'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 PCI Express 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 PCI Express 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 PCI Express 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 PCI Express 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 PCI Express Downstream Port Model

The PCI Express 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 PCI Express 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 PCI Express Downstream Port Model by default makes a check during device configuration that verifies that the PCI Express 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.

PCI Express Downstream Port Model TPI Task List

The PCI Express Downstream Port Model TPI tasks include, which are defined in the tables that follow:

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

Table C-3: Test Setup Tasks

| Name | Input(s) | | Description |
|---------------------------|-------------|-------|---|
| TSK_SYSTEM_INITIALIZATION | None | | Waits for transaction interface reset and link-up between the downstream port and the end point DUT. This task must be invoked prior to PCI Express 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 C-4: TLP Tasks

| Name | Input(s) | | Description |
|---------------------------------|-----------------------------------|--------------------|---|
| TSK_TX_TYPE0_CONFIGURATION_READ | tag_ reg_addr_ first_dw_be_ | 7:0 11:0 3:0 | Waits for transaction interface reset and link-up between the downstream port and the end point DUT. This task must be invoked prior to PCI Express core initialization. |
| TSK_TX_TYPE1_CONFIGURATION_READ | tag_ reg_addr_ first_dw_be_ | 7:0 11:0 3:0 | Sends a Type 1 PCI Express Config Read TLP from downstream port to reg_addr_ of endpoint DUT with tag_ and first_dw_be_ inputs. CplID returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID. |

Table C-4: TLP Tasks (Continued)

| Name | Input(s) | Description |
|----------------------------------|---|---|
| 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 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 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>CplD 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 to 64 bit memory address addr_ of endpoint DUT.</p> <p>CplD returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p> |
| 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_ – | <p>Sends a PCI Express Memory Write TLP from downstream port to 32 bit memory address addr_ of endpoint DUT.</p> <p>CplD returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p> <p>The global DATA_STORE byte array is used to pass write data to task.</p> |
| 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_ – | <p>Sends a PCI Express Memory Write TLP from downstream port to 64 bit memory address addr_ of endpoint DUT.</p> <p>CplD returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p> <p>The global DATA_STORE byte array is used to pass write data to task.</p> |

Table C-4: TLP Tasks (Continued)

| Name | Input(s) | Description |
|------------------------|--|--|
| 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 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 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 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 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. |
| 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 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 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. |

Table C-4: TLP Tasks (Continued)

| Name | Input(s) | Description |
|------------------------|--|--|
| TSK_TX_BAR_READ | bar_index 2:0 byte_offset 31:0 tag_ 7:0 tc_ 2:0 | <p>Sends a PCI Express 1 DWORD Memory 32, Memory 64, or IO Read TLP from the downstream port 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.</p> <p>CplD returned from endpoint DUT will use contents of global COMPLETE_ID_CFG as completion ID.</p> |
| 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.</p> <p>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 C-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 PCI Express 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 can 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 PCI Express 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 PCI Express 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 PCI Express 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's PCI Express 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 C-6: Example PIO Design Tasks

| Name | Input(s) | | Description |
|------------------------|---------------------|-------------|--|
| TSK_TX_READBACK_CONFIG | None | | <p>Performs a sequence of PCI Type 0 Configuration Reads to the PCI Express Endpoint device's Base Address Registers, PCI Command Register, and PCIe Device Control Register using the PCI Express fabric.</p> <p>This task should only be called after TSK_SYSTEM_INITIALIZATION.</p> |
| TSK_MEM_TEST_DATA_BUS | bar_index | 2:0 | <p>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.</p> <p>For an exhaustive test, this task should be called four times, once for each block RAM used in the PIO design.</p> |
| TSK_MEM_TEST_ADDR_BUS | bar_index nBytes | 2:0 31:0 | <p>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.</p> <p>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.</p> |
| TSK_MEM_TEST_DEVICE | bar_index nBytes | 2:0 31:0 | <p>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.</p> <p>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.</p> |

Table C-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 C-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 PCIe Block Plus from the PCI Express core for Virtex-II Pro and Virtex-4 FX, the following list describes the differences in behaviors and options between the PCIe Block Plus core and the PCI Express core (versions 3.3 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 25](#) 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 25](#) 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 [Appendix A, “Managing Receive-Buffer Space for Inbound Completions.”](#)

Configuration Interface

The PCIe 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.
- 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 enabled. The aperture is fixed at 1 Megabyte.
- Accesses to non-supported configuration space result in successful completions with null data.
- There is no support for Phantom Functions or Extended Tag Fields.
- There is no support for user-implemented Extended PCI Configuration Space or PCIe 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.