LogiCORE™ IP Initiator/Target v3.167 for PCI™

User Guide

UG159 June 24, 2009

XILINX®

Xilinx is providing this product documentation, hereinafter “Information,” to you “AS IS” with no warranty of any kind, express or implied. Xilinx makes no representation that the Information, or any particular implementation thereof, is free from any claims of infringement. You are responsible for obtaining any rights you may require for any implementation based on the Information. All specifications are subject to change without notice. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE INFORMATION OR ANY IMPLEMENTATION BASED THEREON, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Except as stated herein, none of the Information 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.

The Design 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 Design in such High-Risk Applications is fully at your risk.

©2000-2009 Xilinx, Inc. XILINX, the Xilinx logo, Virtex, Spartan, ISE and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.

Initiator/Target v3.167 for PCI User Guide
UG159 June 24, 2009

The following table shows the revision history for this document.

	Version	Revision
06/01/00	1.0	Initial Xilinx release.
04/30/03	3.5.1	Updated Additional Resources table in Preface to give correct URL to data sheets index page instead of to obsolete *Programmable Logic Data Book* page.
11/11/04	3.5.2	Added installation and licensing chapter; updated to current template.
12/08/04	3.5.3	Updated for Virtex-4, added PCI 3.0 Compliance Checklist appendix. Build v3.0.140.
3/7/05	3.5.4	Updated to build 3.0.145 and ISE 7.1i
5/13/05	4.0	Updated to build 3.0.150 and Xilinx tools 7.1i SP2.
8/31/05	5.0	Updated to build 3.0.151 and Xilinx tools 7.1i SP3, reworded description of parity errors for PCI Bus to remove the phrase “an address parity error or other serious system error.”
9/12/05	6.0	Updated to build 3.0.152, updated Xilinx tools to 7.1i SP4, updated release date, removed verification of directory structure step from Core Licensing chapter.
1/18/06	7.0	Updated to build 3.0.155, Xilinx tools to 8.1i, release date, added missing figure 9-2.
2/14/06	8.0	Updated to build 158, added SP 2 support to ISE 8.1i, edited/replaced *Chapter 17, Design Constraints* text.
7/13/6	9.0	Updated to build 160, ISE to v8.2i, release date
2/15/07	9.5	Updated to build 161, core version 3.1
5/17/07	10.0	Updated to build 162, fixed all references to PCI to conform to PCI-SIG Guidelines. Advanced support for Cadence IUS to v5.7.

	Version	Revision
8/08/07	10.5	Updated build to 165, ISE to 10.1.
10/10/07	11.0	Updated for IP2 Jade Minor release, fixed trademarked name references
3/24/08	11.5	Updated tool support for ISE v10.1 release.
9/19/08	12.0	Updated to support ISE v10.1 Service Pack 3.
4/24/09	12.5	Updated to core to v3.167. Added support for ISE v11.1 and Spartan-6 FPGAs. Removed support for deprecated devices: Virtex-II, Virtex-II Pro, and Virtex-E.
6/24/09	13.0	Updated to support ISE v11.2. Removed support for Spartan-6 devices.

Table of Contents

Schedule of Figures 11

Schedule of Tables 15

Preface: About This Guide

Guide Contents 19
Conventions 20
Typographical 20
Online Document 21

Chapter 1: Getting Started

About the Core 23
Additional Documentation 23
Technical Support 23
Feedback 23
PCI Interface Core 24
Document 24

Chapter 2: Signal Descriptions

PCI Bus Interface Signals 27
User Interface Signals 33
Special Requirements 43

Chapter 3: General Design Guidelines

Design Steps 45
Target Designs 45
Initiator Designs 45
Burst Designs 45
Advanced Designs 46
Know the Level of Difficulty 46
Understanding Signal Pipelining 47
Keep it Registered 47
Recognize Timing-Critical Signals 47
Use Supported Design Flows 48
Make Only Allowed Modifications 48
Unsupported Devices 49

Chapter 4: Customizing the Interface

Device and Vendor ID 51
Class Code and Revision ID 51
Subsystem Vendor ID and Subsystem ID 52
CardBus CIS Pointer 52
Base Address Registers 52

Latency Timer Settings . . . 53
Interrupt Enable . . . 53
Capabilities List . . . 54
Interrupt Acknowledge . . . 54
Reserved Settings . . . 54

Chapter 5: Target Data Transfer and Control

Before you Begin . . . 55
Determine the Address Space . . . 55
Determine the Bandwidth . . . 55
Assemble the Design . . . 56
Target Register Overview . . . 56
Target Interface Signals . . . 57
Basic Target Interface Signals . . . 57
Additional Target Interface Signals . . . 57
Decoding Target Transactions . . . 58
Target Writes . . . 59
Target Reads . . . 60
Terminating Target Transactions . . . 61

Chapter 6: Target Data Phase Control

Control Modes . . . 63
Control Pipeline . . . 65
Deterministic Control . . . 65
Example 1: Always Ready, Single Transfers . . . 65
Example 2: Always Ready, Burst Transfers . . . 66
Example 3: Initial Latency . . . 66
Example 4: Retries . . . 67
Other Possibilities . . . 68
Non-Deterministic Control . . . 68
Example 5: Subdividing an Address Space . . . 69
Example 6: Multiple Base Address Registers . . . 69
Target Abort . . . 70

Chapter 7: Target Burst Transfers

Keeping Track of the Address Pointer . . . 71
Sinking Data in Burst Transfers . . . 72
Sourcing Data in Burst Transfers . . . 72
Design Examples . . . 74
Example 1: Prefetchable Data Source . . . 74
Example 2: Non-Prefetchable Data Source . . . 75

Chapter 8: Target 64-bit Extension

Target Extension Signals . . . 79
Handling 64-bit Transfers . . . 79

Chapter 9: Target Only Designs

Logic Design Considerations . . . 83
System Level Considerations . . . 84

Chapter 10: Initiator Data Transfer and Control

Before you Begin . . . 85
Determine the Required Transfers . . . 85
Determine Termination Behavior . . . 85
Determine Transaction Ordering Rules . . . 85
Assemble the Design . . . 86
Typical Initiator Data Interface . . . 86
Initiator Interface Signals . . . 87
Initiator Control . . . 88
Inputs to the State Machine . . . 89
The State Machine . . . 91
Outputs to the Core Interface . . . 93
Data Register Control Signals . . . 94
Sample Transactions . . . 95

Chapter 11: Initiator Data Phase Control

Control Modes . . . 101
Control Pipeline . . . 103
Transaction Termination Rules . . . 103
Implementation . . . 104
Sample Transactions . . . 105

Chapter 12: Initiator Burst Transfers

Keeping Track of the Address Pointer . . . 111
Sinking Data in Burst Transfers . . . 112
Sourcing Data in Burst Transfers . . . 112
Design Example . . . 114
Inputs to the State Machine . . . 115
The State Machine . . . 117
Outputs to the Core Interface . . . 119
Control Signals . . . 122
Sample Transactions . . . 122

Chapter 13: Initiator 64-bit Extension

Initiator Extension Signals . . . 129
Controlling 64-bit Transfers . . . 129
Additional Considerations . . . 131
Only Perform Burst Transfers . . . 131
Only Use Aligned Addresses . . . 131
Only Use Allowed Commands . . . 132
Monitor the Target Response . . . 132

Chapter 14: Other Bus Cycles

Supported Commands 137
Target Commands 138
Interrupt Acknowledge 138
Configuration Cycles 139
Initiator Commands 147
Interrupt Acknowledge 147
Special Cycle Commands 148
Configuration Cycles 149
IDSEL Signal Generation 150

Chapter 15: Error Detection and Reporting

Address Parity Errors 155
Data Parity Errors 156

Chapter 16: Design Constraints

Supplied User Constraints 159
Pinout Definition Constraints 159
Absolute Placement Constraints 159
Timing Constraints 160
Directed Routing Constraints 160
Additional User Constraints 160
Guide Files 160

Appendix: Protocol Compliance Checklist

Configuration Organization 164
Configuration Device Control 166
Configuration Device Status 167
Configuration Base Addresses 169
VGA Devices 169
General Component Protocol Checklist (Master) 170
General Component Protocol Checklist (Target) 172
Component Protocol Checklist for a Master Device 174
Test Scenario: 1.1. PCI Device Speed (as indicated by DEVSEL#) Tests 174
Test Scenario: 1.2. PCI Bus Target Abort Cycles 177
Test Scenario: 1.3. PCI Bus Target Retry Cycles 179
Test Scenario: 1.4. PCI Bus Single Data Phase Disconnect Cycles 181
Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles 183
Test Scenario: 1.6. PCI Bus Multi-Data Phase Retry Cycles 186
Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles 189
Test Scenario: 1.8. Multi-data Phase & TRDY# Cycles 191
Test Scenario: 1.9. Bus Data Parity Error Single Cycles 195
Test Scenario: 1.10. Bus Data Parity Error Multi-data Phase Cycles 196
Test Scenario: 1.11. Bus Master Timeout 198
Test Scenario: 1.12. PCI Bus Master Parking 199
Test Scenario: 1.13. PCI Bus Master Arbitration 200
Component Protocol Checklist for a Target Device 201

Test Scenario: 2.1. Target Reception of an Interrupt Cycle 201
Test Scenario: 2.2. Target Reception of Special Cycle 201
Test Scenario: 2.3. Target Detection of Address and Data Parity Error for Special Cycle 201
Test Scenario: 2.4. Target Reception of I/O Cycles With Legal and Illegal Byte Enables 201
Test Scenario: 2.5. Target Ignores Reserved Commands 202
Test Scenario: 2.6. Target Receives Configuration Cycles 202
Test Scenario: 2.7. Target Receives I/O Cycles With Address and Data Parity Errors 202
Test Scenario: 2.8. Target Gets Config Cycles With Address and Data Parity Errors 203
Test Scenario: 2.9. Target Receives Memory Cycles 203
Test Scenario: 2.10. Target Gets Memory Cycles With Address and Data Parity Errors 203
Test Scenario: 2.11. Target Gets Fast Back to Back Cycles 204

Schedule of Figures

Chapter 1: Getting Started

Chapter 2: Signal Descriptions

Figure 2-1: **Top-Level Block Diagram** 26
Figure 2-2: **Bus Interface Symbols for PCI** 27

Chapter 3: General Design Guidelines

Figure 3-3: **Signal Pipeline Delay** 47

Chapter 4: Customizing the Interface

Chapter 5: Target Data Transfer and Control

Figure 5-1: **Example Target Register** 56
Figure 5-2: **Target Write Transaction** 60
Figure 5-3: **Target Read Transaction** 61

Chapter 6: Target Data Phase Control

Figure 6-1: **Permitted Data Phase Control Sequences** 64
Figure 6-2: **Control Signal Pipeline Delay** 65

Chapter 7: Target Burst Transfers

Figure 7-1: **Target Burst Write Transaction** 72
Figure 7-2: **Target Burst Read Transaction** 73
Figure 7-3: **Prefetchable Data Source** 74
Figure 7-4: **Non-prefetchable Data Source** 76
Figure 7-5: **Burst Read of Target FIFO** 78

Chapter 8: Target 64-bit Extension

Figure 8-1: **Unaligned 64-bit Target Read Transaction** 80
Figure 8-2: **Aligned 64-bit Target Read Transaction** 81
Figure 8-3: **Aligned 32-bit Target Read Transaction** 82

Chapter 9: Target Only Designs

Chapter 10: Initiator Data Transfer and Control

Figure 10-1: **Example Initiator Register** 86
Figure 10-2: **Initiator Control Block Diagram** 89

Figure 10-3: **Sample User Application State Machine** 91
Figure 10-4: **Initiator Read Transaction** 95
Figure 10-5: **Initiator Write Retried by Target** 96
Figure 10-6: **Initiator Write Retried and Completed** 97
Figure 10-7: **Target Abort** 98
Figure 10-8: **Master Abort** 99

Chapter 11: Initiator Data Phase Control

Figure 11-1: **Permitted Data Phase Control Sequences** 102
Figure 11-2: **Two DWORD Initiator Read Transfer** 106
Figure 11-3: **Two DWORD Initiator Write Transfer** 107
Figure 11-4: **Three DWORD Initiator Read Transfer** 108
Figure 11-5: **Three DWORD Initiator Write Transfer** 109

Chapter 12: Initiator Burst Transfers

Figure 12-1: **Initiator Burst Read Transaction** 112
Figure 12-2: **Initiator Burst Write Transaction** 113
Figure 12-3: **Non-Prefetchable Data Source** 114
Figure 12-4: **User Application State Machine** 117
Figure 12-5: **Burst Read Transaction with Normal Termination** 123
Figure 12-6: **Burst Write Transaction with Normal Termination** 124
Figure 12-7: **Burst Read with Target Disconnect** 125
Figure 12-8: **Burst Read Continues after Target Disconnect** 126
Figure 12-9: **Burst Write with Target Disconnect** 127
Figure 12-10: **Burst Write Continues after Target Disconnect** 128

Chapter 13: Initiator 64-bit Extension

Figure 13-1: **Two QWORD Initiator Read Transfer** 130
Figure 13-2: **Two QWORD Initiator Write Transfer** 131
Figure 13-3: **32-bit Target Responds to 64-bit Read** 133
Figure 13-4: **32-bit Target Disconnects with Data on Read** 134
Figure 13-5: **32-bit Target Responds to 64-bit Write** 135
Figure 13-6: **32-bit Target Disconnects with Data on Write** 136

Chapter 14: Other Bus Cycles

Figure 14-1: **Target Accepts Interrupt Acknowledge Cycle** 139
Figure 14-2: **Example Configuration Register** 141
Figure 14-3: **Configuration Read with Wait States** 145
Figure 14-4: **Configuration Write with Wait States** 146
Figure 14-5: **Initiator Issues Interrupt Acknowledge Cycle** 148
Figure 14-6: **Initiator Issues Special Cycle** 149
Figure 14-7: **Self-Configuration Read** 151
Figure 14-8: **Self-Configuration Write** 152

Chapter 15: Error Detection and Reporting

Figure 15-1: **Target Write with no Parity Errors** 155
Figure 15-2: **Target Write with Address Parity Error** 156
Figure 15-3: **Target Write with Data Parity Error** 157

Chapter 16: Design Constraints

Appendix: Protocol Compliance Checklist

Schedule of Tables

Chapter 1: Getting Started

Chapter 2: Signal Descriptions

Table 2-1: **Bus Interface Signals for PCI** 28
Table 2-2: **User Interface Signals** 34

Chapter 3: General Design Guidelines

Table 3-3: **Level of Difficulty with Implementations for PCI** 46

Chapter 4: Customizing the Interface

Chapter 5: Target Data Transfer and Control

Chapter 6: Target Data Phase Control

Table 6-1: **Data Phase Control Signals for Targets** 63

Chapter 7: Target Burst Transfers

Table 7-1: **Example Target Address Pointer** 71

Chapter 8: Target 64-bit Extension

Chapter 9: Target Only Designs

Table 9-1: **Initiator Control Signals** 83

Chapter 10: Initiator Data Transfer and Control

Chapter 11: Initiator Data Phase Control

Table 11-1: **Data Phase Control Signals for Initiators** 102

Chapter 12: Initiator Burst Transfers

Table 12-1: **Example Initiator Address Pointer** 111

Chapter 13: Initiator 64-bit Extension

Chapter 14: Other Bus Cycles

Table 14-1: **Supported PCI Bus Commands** 137

Table 14-2: **PCI Configuration Space Header** 140
Table 14-3: **Configuration Data Phase Control** 143

Chapter 15: Error Detection and Reporting

Chapter 16: Design Constraints

Appendix: Protocol Compliance Checklist

Table A-1: **Configuration Organization Checklist** 164
Table A-2: **Function Implementation Checklist** 165
Table A-3: **Configuration Device Control Checklist** 166
Table A-4: **Command and Status Registers Checklist** 166
Table A-5: **Configuration Device Status Checklist** 167
Table A-6: **Device Status Options Checklist** 167
Table A-7: **Configuration Base Address Checklist** 169
Table A-8: **VGA Device Checklist** 169
Table A-9: **General Component Protocol Checklist (Master)** 170
Table A-10: **General Component Protocol Checklist (Target)** 172
Table A-11: **PCI Device Speed Checklist #1** 174
Table A-12: **PCI Device Speed Checklist #2** 175
Table A-13: **PCI Device Speed Checklist #3** 175
Table A-14: **PCI Device Speed Checklist #4** 176
Table A-15: **PCI Device Speed Checklist #5** 176
Table A-16: **Explanations for PCI Device Speed** 176
Table A-17: **PCI Bus Target Abort Cycles Checklist #1** 177
Table A-18: **PCI Bus Target Abort Cycles Checklist #2** 177
Table A-19: **PCI Bus Target Abort Cycles Checklist #3** 178
Table A-20: **PCI Bus Target Abort Cycles Checklist #4** 179
Table A-21: **Explanations for PCI Target Abort Cycles** 179
Table A-22: **PCI Bus Target Retry Cycles Checklist #1** 179
Table A-23: **PCI Bus Target Retry Cycles Checklist #2** 180
Table A-24: **PCI Bus Target Retry Cycles Checklist #3** 180
Table A-25: **PCI Bus Target Retry Cycles Checklist #4** 181
Table A-26: **Explanations for PCI Bus Target Retry Cycles** 181
Table A-27: **PCI Bus Single Data Phase Disconnect Cycles Checklist #1** 181
Table A-28: **PCI Bus Single Data Phase Disconnect Cycles Checklist #2** 182
Table A-29: **PCI Bus Single Data Phase Disconnect Cycles Checklist #3** 182
Table A-30: **PCI Bus Single Data Phase Disconnect Cycles Checklist #4** 182
Table A-31: **Explanation for PCI Bus Single Data Phase Disconnect Cycles** 183
Table A-32: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #1** 183
Table A-33: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #2** 184
Table A-34: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #3** 184
Table A-35: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #4** 185

Table A-36: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #5** 185
Table A-37: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #6** 186
Table A-38: **Explanations for PCI Bus Multi-Data Phase Target Abort Cycles** 186
Table A-39: **PCI Bus Multi-Data Phase Retry Cycles Checklist #1** 186
Table A-40: **PCI Bus Multi-Data Phase Retry Cycles Checklist #2** 187
Table A-41: **PCI Bus Multi-Data Phase Retry Cycles Checklist #3** 187
Table A-42: **PCI Bus Multi-Data Phase Retry Cycles Checklist #4** 188
Table A-43: **PCI Bus Multi-Data Phase Retry Cycles Checklist #5** 188
Table A-44: **PCI Bus Multi-Data Phase Retry Cycles Checklist #6** 188
Table A-45: **Explanations for PCI Bus Multi-Data Phase Retry Cycles** 188
Table A-46: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #1** 189
Table A-47: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #2** 189
Table A-48: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #3** 190
Table A-49: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #4** 190
Table A-50: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #4** 190
Table A-51: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #5** 191
Table A-52: **Explanation for PCI Bus Multi-Data Phase Disconnect Cycles** 191
Table A-53: **Multi-data Phase & TRDY# Cycles Checklist #1** 191
Table A-54: **Multi-Data Phase & TRDY# Cycles Checklist #2** 192
Table A-55: **Multi-Data Phase & TRDY# Cycles Checklist #3** 193
Table A-56: **Multi-Data Phase & TRDY# Cycles Checklist #4** 194
Table A-57: **Multi-Data Phase & TRDY# Cycles Checklist #5** 194
Table A-58: **Explanations for Multi-Data Phase & TRDY# Cycles** 195
Table A-59: **Bus Data Parity Error Single Cycles Checklist #1** 195
Table A-60: **Bus Data Parity Error Single Cycles Checklist #2** 195
Table A-61: **Bus Data Parity Error Single Cycles Checklist #3** 196
Table A-62: **Explanations for Bus Data Parity Error Single Cycles** 196
Table A-63: **Bus Data Parity Error Multi-data Phase Cycles Checklist #1** 196
Table A-64: **Bus Data Parity Error Multi-data Phase Cycles Checklist #2** 197
Table A-65: **Bus Data Parity Error Multi-data Phase Cycles Checklist #3** 197
Table A-66: **Bus Data Parity Error Multi-data Phase Cycles Checklist #4** 197
Table A-67: **Bus Data Parity Error Multi-data Phase Cycles Checklist #5** 197
Table A-68: **Bus Data Parity Error Multi-data Phase Cycles Checklist #6** 198
Table A-69: **Explanations for Bus Data Parity Error Multi-data Phase Cycles** 198
Table A-70: **Bus Master Timeout Checklist #1** 198
Table A-71: **Bus Master Timeout Checklist #2** 199
Table A-72: **Explanations for Bus Master Timeout Checklist** 199
Table A-73: **PCI Bus Master Parking Checklist #1** 199
Table A-74: **PCI Bus Master Parking Checklist #2** 199
Table A-75: **Explanations for PCI Bus Master Parking** 200
Table A-76: **PCI Bus Master Arbitration Checklist** 200
Table A-77: **Explanations for PCI Bus Master Arbitration** 200
Table A-78: **Target Reception of an Interrupt Cycle Checklist** 201

Table A-79: **Target Reception of Special Cycle Checklist** 201
Table A-80: **Target Detection of Address and Data Parity Error for Special Cycle Checklist** 201
Table A-81: **Target Reception of I/O Cycles With Legal and Illegal Byte Enables Checklist** 201
Table A-82: **Target Ignores Reserved Commands Checklist** 202
Table A-83: **Target Receives Configuration Cycles** 202
Table A-84: **Target Receives I/O Cycles With Address and Data Parity Errors Checklist** 202
Table A-85: **Target Gets Config Cycles With Address and Data Parity Errors Checklist** 203
Table A-86: **Target Receives Memory Cycles Checklist** 203
Table A-87: **Target Gets Memory Cycles With Address and Data Parity Errors Checklist** 203
Table A-88: **Target Gets Fast Back to Back Cycles Checklist** 204
Table A-89: **Explanations for Test Scenarios 2.1 - 2.11** 204

About This Guide

The L*ogiCORE™ IP Initiator/Target v3.167 for PCI™* guide provides information about the Peripheral Component Interconnect (PCI) interface, which provides a fully verified, pre-implemented PCI bus interface available in both 32-bit and 64-bit versions, based on the Virtex® and Spartan® architectures. The guide serves as a comprehensive reference for use during the design phase of a project.

Guide Contents

This guide contains the following chapters:

- Chapter 1, "Getting Started" provides information about getting technical support, and providing feedback to Xilinx about the Initiator/Target v3.166 for PCI core and accompanying documentation.
- Chapter 2, "Signal Descriptions" defines the PCI bus interface signals and the user interface signals.
- Chapter 3, "General Design Guidelines" provides guidelines for turning anInitiator/Target v3.166 for PCI core interface into a fully functioning design integrated with user application logic.
- Chapter 4, "Customizing the Interface"contains detailed information about the allowable modifications to the Initiator/Target v3.166 for PCI core.
- Chapter 5, "Target Data Transfer and Control" provides basic design guidelines and defines the required user application logic to generate the load and output enable signals for a typical target register.
- Chapter 6, "Target Data Phase Control" discusses the mechanism by which the user application can control various aspects of target transactions to accommodate its own ability to source or sink data.
- Chapter 7, "Target Burst Transfers" discusses the advantages of building user applications that support target burst transfers.
- Chapter 8, "Target 64-bit Extension" provides detailed information about the target 64-bit extension.
- Chapter 9, "Target Only Designs" provides instructions for disabling the initiator functions present in the Initiator/Target v3.166 for PCI core interface.
- Chapter 10, "Initiator Data Transfer and Control" defines the fundamental aspects of controlling initiator transactions.
- Chapter 11, "Initiator Data Phase Control" discusses the mechanism by which the user application can control aspects of initiator transactions to accommodate its own ability to source or sink data.

- Chapter 12, "Initiator Burst Transfers" discusses the performance advantages derived from building applications that support initiator burst transfers.
- Chapter 13, "Initiator 64-bit Extension" provides details about the initiator 64-bit extension, including provisions that must be made for exceptional conditions that may occur during 64-bit transfers.
- Chapter 14, "Other Bus Cycles" demonstrates the use of the more esoteric commands available in the Initiator/Target v3.166 for PCI core.
- Chapter 15, "Error Detection and Reporting" defines how the Initiator/Target v3.166 for PCI interface generates and checks parity, and how errors are reported, as outlined the *PCI Local Bus Specification*.
- Chapter 16, "Design Constraints" defines how the Initiator/Target v3.166 for PCI uses timing constraints during processing to ensure that the timing requirements are met when the user design is added.
- Appendix, "Protocol Compliance Checklist" provides the PCI-SIG protocol compliance checklist to assist in the design review process.

Conventions

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`
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *`design_name`*
	References to other manuals	See the *`Development System Reference Guide`* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *`not`* connected.
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}`**

Convention	Meaning or Use	Example

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;`

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See "Additional Resources" for details. See "Title Formats" in Chapter 1 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

XILINX®

Chapter 1

Getting Started

The Initiator/Target v3.166 for PCI is a fully verified, pre-implemented PCI bus interface available in 32-bit and 64-bit versions, with support for multiple Xilinx FPGA device families. All code samples in this guide are provided in Verilog™-HDL. In addition, an example design is provided in both Verilog-HDL and VHDL in the *Initiator/Target v3.2 for PCI Getting Started Guide*.

About the Core

This chapter assumes that you have installed the core using either the CORE Generator™ IP Software Update installer, or by performing a manual installation after downloading the core from the web.

For detailed information about the core, see the product page for PCI. For information about licensing options, see Chapter 2, "Licensing the Core" in the *Initiator/Target v3.2 for PCI Getting Started Guide.*

Additional Documentation

For more information about the core, see the following documents, located in the CORE Generator software zip file:

- *Initiator/Target for PCI Release Notes*
- *Initiator/Target v3.2 for PCI Getting Started Guide*

Additional information is available in the Mindshare PCI System Architecture text, and the *PCI Local Bus Specification*, available from the PCI Special Interest Group site.

Technical Support

For technical support, visit www.xilinx.com/support. Questions are routed to a team of engineers with expertise using the Initiator/Target v3.166 for PCI core.

Xilinx provides technical support for use of this product as described in the *Initiator/Target v3.2 for PCI User Guide* and the *Initiator/Target v3.2 for PCI 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 Initiator/Target v3.166 for PCI and the documentation supplied with the core.

PCI Interface Core

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

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

XILINX®

Chapter 2

Signal Descriptions

This chapter defines the interface signals for PCI. The standard PCI Bus interface signals are defined on the left side of the interface symbol and all user interface signals are defined

on the right side of the symbol. Figure 2-1 displays a typical user application interfaced to the Initiator/Target v3.166 for PCI core.

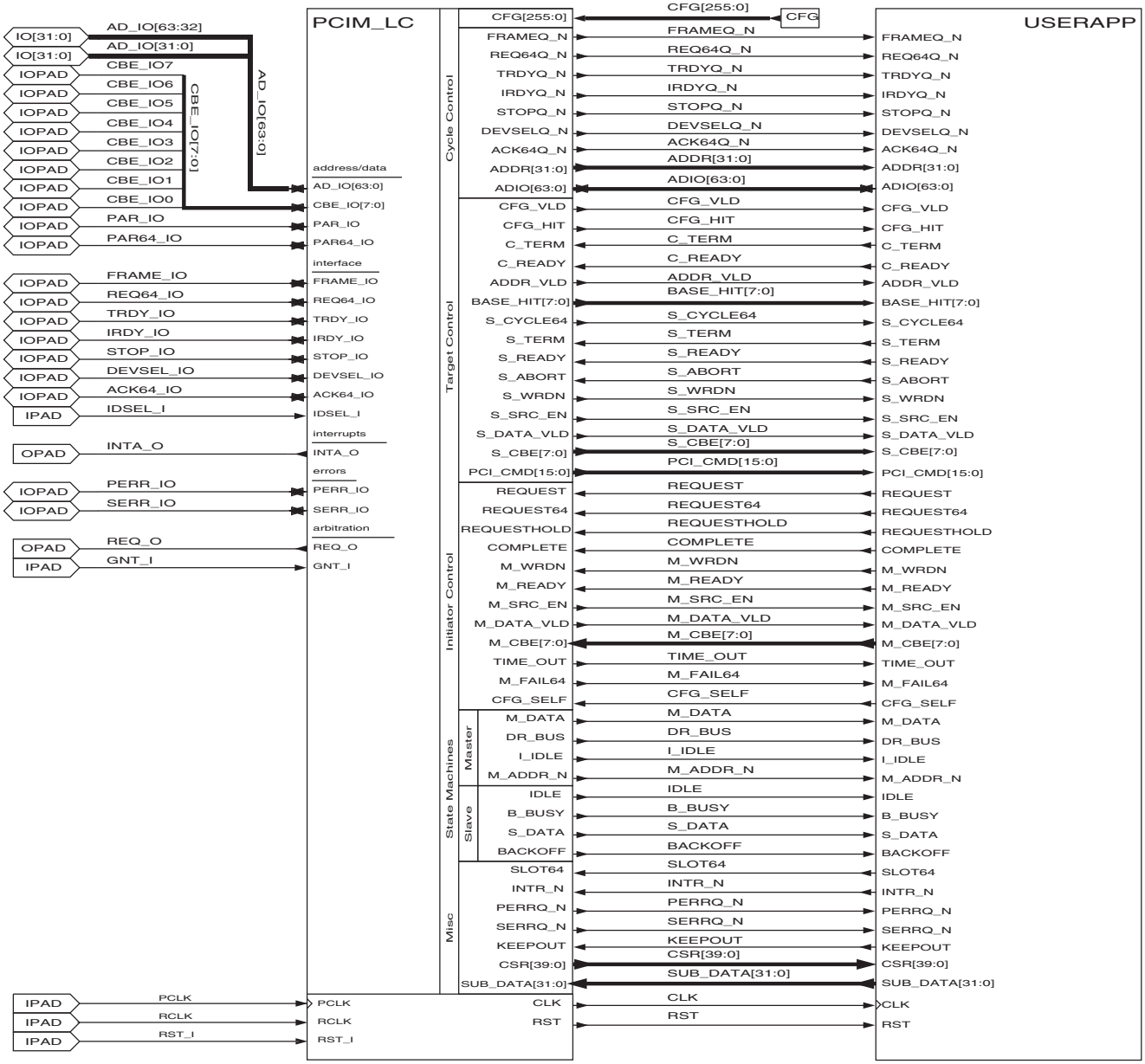


Figure 2-1: **Top-Level Block Diagram**

For HDL users, the file `pcim_top.v` or `pcim_top.vhd` represents this structure.

PCI Bus Interface Signals

Table 2-1 defines the interface signals that comprise the PCI Local Bus. These signals appear on the left side of Figure 2-2.

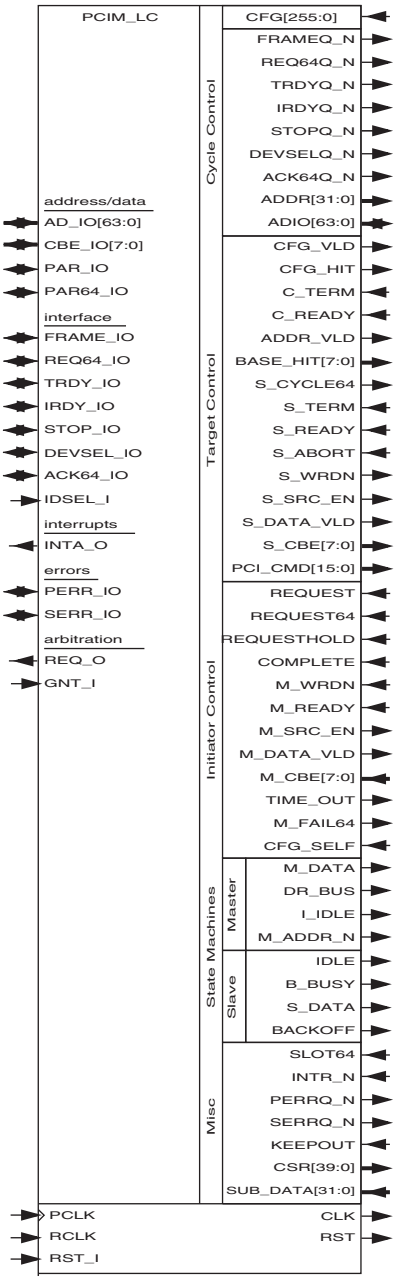


Figure 2-2: **Bus Interface Symbols for PCI**

Note: Pin locations are device and package dependent. See the appropriate user constraints file for specific device configurations

Table 2-1: **Bus Interface Signals for PCI**

Signal Name	Type	Functional Description
Address and Data Path		
AD_IO[31:0]	t/s	AD_IO[31:0] is a time-multiplexed address and data bus. Each bus transaction consists of an address phase followed by one or more data phases.
CBE_IO[3:0]	t/s	CBE_IO[3:0] is a time-multiplexed bus command and byte enable bus. Bus commands are asserted during an address phase on the bus. Byte enables are asserted during data phases.
PAR_IO	t/s	PAR_IO generates and checks even parity across AD_IO[31:0] and CBE_IO[3:0]. When the core interface is the source of an address or data, the interface generates even parity across AD_IO[31:0] and CBE_IO[3:0] and presents the result on PAR_IO one cycle after the values were presented on AD_IO[31:0] and CBE_IO[3:0]. When the interface receives an address or data, the interface checks for even parity across AD_IO[31:0] and CBE_IO[3:0] and compares it to PAR_IO one cycle later. Parity errors are reported via PERR_IO.

Table 2-1: **Bus Interface Signals for PCI *(Continued)***

Signal Name	Type	Functional Description
Transaction Control		
FRAME_IO	s/t/s active low	FRAME_IO is driven by an initiator to indicate a bus transaction. FRAME_IO is asserted for the duration of the operation and is deasserted during the last data phase to identify the end of the transaction. When operating as an initiator, the core interface will only assert FRAME_IO when all of the following conditions are met: • GNT_I has been asserted for more than one cycle • IRDY_IO and FRAME_IO are deasserted, meaning the bus is idle • The bus master enable bit (CSR2) is set in the command register • The user application has asserted REQUEST or REQUEST64 The core interface will deassert FRAME_IO upon any of the following conditions: • The user application asserts COMPLETE • The interface receives a termination from the addressed target (retry, disconnect, or abort) • *Not* receiving a DEVSEL_IO assertion from the addressed target (master abort) • The internal latency timer has expired, if enabled, and the system arbiter is no longer asserting GNT_I • A 32-bit target responds to a 64-bit transfer request
DEVSEL_IO	s/t/s active low	DEVSEL_IO indicates that a target has decoded the address presented during the address phase and is claiming the transaction. This occurs when the address matches one of the Base Address Registers in the target.
TRDY_IO	s/t/s active low	TRDY_IO indicates that the target is ready to complete the current data phase. When TRDY_IO is asserted, the target is ready to transfer data. Data transfer occurs when both TRDY_IO and IRDY_IO are asserted on the bus.

Table 2-1: **Bus Interface Signals for PCI *(Continued)***

Signal Name	Type	Functional Description
IRDY_IO	s/t/s active low	IRDY_IO indicates that the initiator is ready to complete the current data phase. When IRDY_IO is asserted, the initiator is ready to transfer data. Data transfer occurs when both TRDY_IO and IRDY_IO are asserted on the bus.
STOP_IO	s/t/s active low	STOP_IO indicates that the target has requested to stop the current transaction. The target uses STOP_IO to signal a disconnect, retry, or target abort. The interface asserts STOP_IO under the control of the user application. However, the interface automatically asserts it during non-linear memory transactions, performing disconnect with data.
IDSEL_I	in	IDSEL_I indicates that the interface is the target of a configuration cycle.
Interrupts		
INTA_O	o/d active low	INTA_O indicates the core interface requests an interrupt. This may be disabled by setting the interrupt disable bit in the command register.
Error Signals		
PERR_IO	s/t/s active low	PERR_IO indicates that a parity error was detected while the core interface was the target of a write transfer or the initiator of a read transfer. Parity errors are reported two clock cycles after the data transaction appeared on the AD_IO and CBE_IO lines. Parity error reporting on PERR_IO is enabled by setting the report parity errors bit (CSR6) in the command register. Parity errors, except those during special cycles, are always reported in the status register (CSR31). Additionally, the initiator reports parity errors during a transaction when it was the bus master. The error is reported via the data parity error detected bit (CSR24) in the status register if the report parity errors bit (CSR6) is set in the command register.

Table 2-1: **Bus Interface Signals for PCI *(Continued)***

Signal Name	Type	Functional Description
SERR_IO	o/d active low	SERR_IO indicates that a parity error was detected during an address cycle, except during special cycles. SERR_IO is asserted on the third clock after FRAME_IO is first asserted. System errors are reported on the signaled system error bit (CSR30) in the status register if the SERR_IO enable bit (CSR8) and the report parity errors bit (CSR6) are set in the command register. SERR_IO is an open-drain output. Per the *PCI Local Bus Specification*, SERR_IO is not actively driven high after assertion.
Arbitration		
REQ_O	t/s active low	REQ_O indicates to the arbiter that the PCI initiator requests access to the bus. The initiator may only request the bus when it has been enabled by setting the bus master enable bit (CSR2) in the command register.
GNT_I	t/s active low	GNT_I indicates that the arbiter has granted the bus to the PCI initiator. If GNT_I is asserted and there is *not* a pending request, or the bus master enable bit is not set, then the interface performs bus parking.
System Signals		
RST_I	in active low	RST_I is the PCI Bus reset signal. This signal is used to bring PCI-specific registers, sequencers, and signals to a consistent state. Any time RST_I is asserted, all PCI output signals are three-stated.
PCLK	in	PCLK is the PCI Bus clock signal. This signal provides timing for all transactions on the PCI Bus and is an input to every PCI device. The frequency of PCLK may vary as allowed in the *PCI Local Bus Specification*.
RCLK	in	RCLK is a reference clock input present only in specific Virtex-4 implementations of this interface. It is a reference clock used to calibrate input delay lines.
64-bit Extension		
AD_IO[63:32]	t/s	AD_IO[63:32] is a time-multiplexed address and data bus. Each bus transaction consists of an address phase followed by one or more data phases. During address phases presented by 64-bit initiators, AD_IO[31:0] is driven with valid (reserved) values.

Table 2-1: **Bus Interface Signals for PCI *(Continued)***

Signal Name	Type	Functional Description
CBE_IO[7:4]	t/s	CBE_IO[7:4] is a time-multiplexed bus command and byte enable bus. During address phases presented by 64-bit initiators, CBE_IO[7:4] is driven with valid (reserved) values. Byte enables for the 64-bit extension are asserted during data phases.
PAR64_IO	t/s	PAR64_IO generates and checks even parity across AD_IO[63:32] and CBE_IO[7:4]. When the core interface is the source of an address or data, the interface generates even parity across AD_IO[63:32] and CBE_IO[7:4] and presents the result on PAR64_IO one cycle after the values were presented on AD_IO[63:32] and CBE_IO[7:4]. When the interface receives an address or data, the interface checks for even parity across AD_IO[63:32] and CBE_IO[7:4] and compares it to PAR64_IO one cycle later. Parity errors are reported via PERR_IO.
ACK64_IO	s/t/s active low	ACK64_IO indicates that a target has decoded the address presented during the address phase and is claiming the transaction as a 64-bit target. This occurs when the initiator makes a 64-bit transfer request using REQ64_IO, the address matches one of the Base Address Registers in the target, and the target is 64-bit enabled.

Table 2-1: **Bus Interface Signals for PCI *(Continued)***

Signal Name	Type	Functional Description
REQ64_IO	s/t/s active low	REQ64_IO is driven by the initiator to indicate a 64-bit bus transaction. REQ64_IO is asserted for the duration of the operation and is deasserted during the last data phase to identify the end of the transaction. Its behavior is similar to FRAME_IO. When operating as an initiator, the core interface will only assert REQ64_IO when all of the following conditions are met: • GNT_I has been asserted for more than one cycle • IRDY_IO and FRAME_IO are deasserted, meaning the bus is idle • The bus master enable bit (CSR2) is set in the command register • The user application has asserted REQUEST64 The core interface will deassert REQ64_IO upon any of the following conditions: • The user application asserts COMPLETE • The interface receives a termination from the addressed target (retry, disconnect, or abort) • *Not* receiving a DEVSEL_IO assertion from the addressed target (master abort) • The internal latency timer has expired, if enabled, and the system arbiter is no longer asserting GNT_I • A 32-bit target responds to a 64-bit transfer request

in = input only signal

out = output only signal

t/s = bidirectional, three-state signal

s/t/s = bidirectional, sustained three-state signal

o/d = open drain

User Interface Signals

The user interface to the core interface provides a superset of the necessary data paths and control signals required for typical applications. This provides ultimate flexibility for specialized user applications.

Table 2-2 describes the interface signals available to the user application. These signals appear on the right side of Figure 2-1.

Table 2-2: **User Interface Signals**

Signal Name	Type	Functional Description
Configuration		
CFG[255:0]	in	Driven by a cfg module, configures the core interface
Cycle Control		
FRAMEQ_N	out active low	A registered version of the PCI Bus FRAME_IO signal
DEVSELQ_N	out active low	A registered version of the PCI Bus DEVSEL_IO signal
IRDYQ_N	out active low	A registered version of the PCI Bus IRDY_IO signal
TRDYQ_N	out active low	A registered version of the PCI Bus TRDY_IO signal
STOPQ_N	out active low	A registered version of the PCI Bus STOP_IO signal
Address and Data Path		
ADDR[31:0]	out	Holds PCI Bus addresses latched during address phases. It may be used for address decoding or for loading user address counters. The address is available in the cycle following the assertion of ADDR_VLD and remains stable until ADDR_VLD is asserted again.
ADIO[31:0]	t/s	A time multiplexed address and data bus. This bus must be driven using internal three-state buffers.
Target Control		
ADDR_VLD	out	Indicates the beginning of a *potential* address phase on the PCI Bus, which is available on the ADIO[31:0] internal bus. The latched address is presented on ADDR[31:0] one cycle later. Likewise, the PCI Bus command is presented on S_CBE[3:0] and latched, decoded, and presented on PCI_CMD[15:0]. ADDR_VLD is active only during potential target operations. It is not asserted during address phases that result from core initiator activity.

Table 2-2: **User Interface Signals *(Continued)***

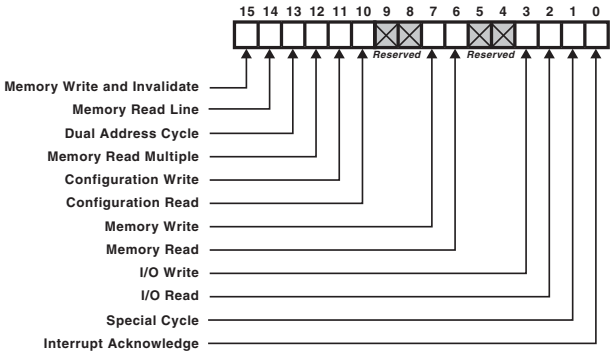
Signal Name	Type	Functional Description
CFG_VLD	out	Indicates the beginning of a *potential* configuration cycle. This signal is similar in nature to ADDR_VLD but is further qualified by IDSEL_I.
S_DATA_VLD	out	Indicates that a data transaction has occurred on the PCI Bus while the interface is a target. S_DATA_VLD is asserted on the clock cycle after data transfer occurs on the PCI Bus and the target state machine is in the S_DATA state. When receiving data, S_DATA_VLD also indicates that the data is available on the ADIO bus. When sourcing data, S_DATA_VLD indicates successful data transfer.
S_SRC_EN	out	An enable signal used to increment a data pointer when the interface is the source of data in a target burst read.
S_WRDN	out	Indicates the data transfer direction during target transactions. During target writes, S_WRDN is asserted; during target reads, it is deasserted.
PCI_CMD[15:0]	out	Indicates the current decoded and latched PCI Bus operation. This bus is a fully decoded (one hot) version of the current PCI Bus command. The command is captured during the address phase and remains stable until the next address phase. 
S_CBE[3:0]	out	Indicates the current PCI Bus command or byte enables for a target access. Note that byte enables are active low.

Table 2-2: **User Interface Signals *(Continued)***

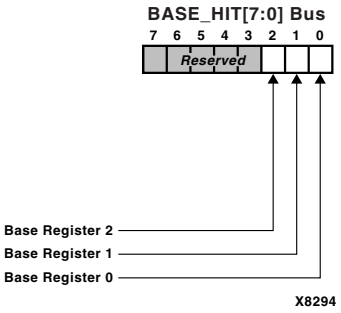
Signal Name	Type	Functional Description
BASE_HIT[7:0]	out	Indicates that one of the base address registers has decoded and matched an address. The bus is one-hot encoded as indicated below. The BASE_HIT signals are active for one clock cycle, the cycle preceding the S_DATA state. 
CFG_HIT	out	Indicates the start of a valid configuration cycle. The CFG_HIT signal is active for one clock cycle, the cycle preceding the S_DATA state. It is similar in nature to the BASE_HIT signals.
C_READY	in	Signals that the user application is ready to transfer configuration data. This is one of the signals that controls TRDY_IO. For most applications, C_READY should always be asserted. The exceptions are applications that require access to user configuration space.
C_TERM	in	Signals that the user application is terminating the transfer of configuration data. This is one of the signals that controls STOP_IO. For most applications C_TERM should always be asserted. The exceptions are applications that require wait states when accessing user configuration space.
S_READY	in	Signals that the user application is ready to transfer data. This is one of the signals that controls TRDY_IO. If the user application is not ready to transfer data, S_READY should be delayed until it is ready to support a sustained burst transfer. Do not deassert S_READY during a transfer.
S_TERM	in	Signals that the user application is terminating the transfer of data. This is one of the signals that controls STOP_IO.

Table 2-2: **User Interface Signals *(Continued)***

Signal Name	Type	Functional Description
S_ABORT	in	Signals a serious error condition that requires the current transaction to stop. S_ABORT should be used to signal an address overrun during a burst transfer or an unaligned 64-bit access.
Initiator Control		
REQUEST	in	Used to request a initiator transaction. Assertion of REQUEST causes the core interface to assert REQ_O if the bus master enable bit (CSR2) is set in the command register. This bit is cleared at reset.
REQUESTHOLD	in	Used to force an extended bus request. Assertion of REQUESTHOLD causes the core interface to assert REQ_O if the bus master enable bit (CSR2) is set in the command register. Unlike the REQUEST signal, REQUESTHOLD is not an input to the core initiator state machine. REQUESTHOLD is intended to allow applications with very demanding bandwidth requirements to keep REQ_O asserted as long as possible. Do not assert REQUESTHOLD unless a transfer has been requested and is in progress. Otherwise, the arbiter may identify the core interface as a broken master.
M_CBE[3:0]	in	Used by the user application to drive command and byte enables during initiator transactions. Bus commands should be presented during the assertion of M_ADDR_N, and byte enables should be presented during the M_DATA state. Byte enables are active low.
M_WRDN	in	Indicates the data transfer direction during initiator transactions. During initiator writes, assert M_WRDN; during initiator reads, deassert it.
COMPLETE	in	Signals the initiator state machine to finish the current transaction. Once asserted, COMPLETE must remain asserted until the state machine exits M_DATA state. This is one of the signals that control FRAME_IO and IRDY_IO.

Table 2-2: **User Interface Signals *(Continued)***

Signal Name	Type	Functional Description
M_READY	in	Signals that the user application is ready to transfer data. If deasserted, wait states are inserted. This is one of the signals which controls IRDY_IO. If the user application is not ready to transfer data, M_READY should be delayed until the application is ready to support a sustained burst transfer. Do not deassert M_READY during a transfer.
M_DATA_VLD	out	Indicates that a data transaction has occurred on the PCI Bus while the core interface is an initiator. M_DATA_VLD is asserted on the clock cycle after data transfer occurs on the PCI Bus and the initiator state machine is in the M_DATA state. When receiving data, M_DATA_VLD also indicates that the data is available on the ADIO bus. When sourcing data, M_DATA_VLD indicates successful data transfer.
M_SRC_EN	out	An enable signal used to increment a data pointer when the interface is the source of data in an initiator burst write.
CFG_SELF	in	Indicates to the core interface that it is allowed to issue a configuration cycle to itself. The assertion of this signal overrides the bus master enable bit (CSR2) and modifies the internal data path. It is intended for use *only* in host bridge applications.
TIME_OUT	out	Indicates the internal latency timer has expired and the user application has exceeded the maximum number of clock cycles allowed by the system configuration software. If the latency timer expires while the system arbiter is still asserting GNT_I, the operation continues until either the operation completes, or the arbiter deasserts GNT_I. If the latency timer expires and the system arbiter has already deasserted GNT_I, the operation terminates. The user application should handle this termination like any other target termination. **Note**: The default latency timer value is 0, indicating immediate time-out. Ensure that the system configuration software writes a sufficiently large value in the latency timer register to allow the desired transfer size.

Table 2-2: **User Interface Signals *(Continued)***

Signal Name	Type	Functional Description
State Machine - Initiator		
M_DATA	out	Indicates that the initiator is in the data transfer state. The M_DATA state occurs after the assertion of M_ADDR_N unless a single cycle assertion of GNT_I occurs.
DR_BUS	out	Indicates that the bus is parked on the core interface. The core initiator is then responsible for driving the AD_IO[31:0] bus, the CBE_IO[3:0] bus, and the PAR_IO signal to prevent these three-state bus signals from floating. The actual values driven on these lines are not important.
M_ADDR_N	out active low	Indicates that the initiator is in the address state. During this time, the user application must drive a valid address on ADIO and a valid bus command on M_CBE. M_ADDR_N is asserted with a one clock cycle overlap with either the I_IDLE or DR_BUS states.
I_IDLE	out	Indicates that the initiator is in the idle state. The initiator is either not enabled, does not have an active request pending, or has not received GNT_I from the system arbiter. The state machine will always remain in either the I_IDLE or DR_BUS state when the bus master enable bit (CSR2) in the command register is reset.
State Machine - Target		
IDLE	out	Indicates that the target is in the idle state.
B_BUSY	out	Indicates that the PCI Bus is busy. An agent has started a transaction (FRAME_IO has been asserted) but the target state machine either has not yet finished decoding the address or has determined that it is not the target of the current operation.
S_DATA	out	Indicates that the target is in a data transfer state. The target has decoded the address and matched it to one of its Base Address Registers or a configuration operation is in progress. The target has accepted the request and will respond.
BACKOFF	out	Indicates that the user application asserted S_TERM or C_TERM and the target state machine is waiting for the transaction to complete.

Table 2-2: **User Interface Signals *(Continued)***

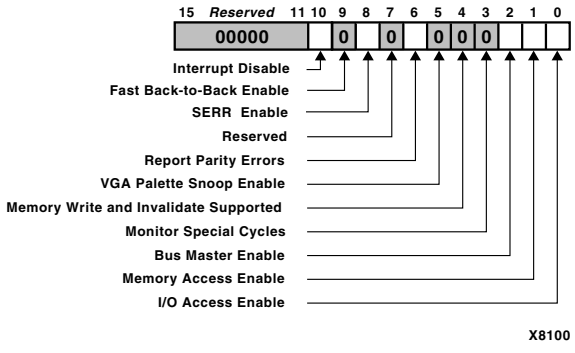
Signal Name	Type	Functional Description
Miscellaneous Signals		
PERRQ_N	out active low	A registered version of the PCI Bus PERR_IO signal.
SERRQ_N	out active low	A registered version of the PCI Bus SERR_IO signal.
INTR_N	in active low	Signals an interrupt request from the user application. The assertion of this signal generates an interrupt request on the PCI Bus unless the interrupt disable bit of the command register is set. Once INTR_N is asserted, the user application must keep it asserted until the device driver clears the interrupt. This mechanism is implementation dependent.
KEEPOUT	in	Isolates the internal ADIO bus from the core interface. This allows the user application to perform data transfer over ADIO without interference. When using KEEPOUT, assert S_TERM and C_TERM, and deassert S_READY and C_READY to terminate all incoming transfer attempts with retry. Tie to logic 0 when not used.
CSR[15:0]	out	Provides access to the command register state bits. These bits are directly set or reset through the system configuration software. All values in the command register are either registered or read-only. **Note**: The bus master enable bit must be set in the command register before the initiator can access the PCI Bus. The I/O access enable bit and/or the memory access enable bit must be set in the command register before the target will respond. 

Table 2-2: **User Interface Signals *(Continued)***

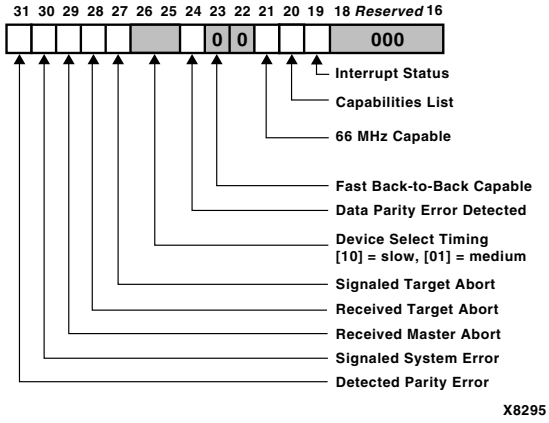
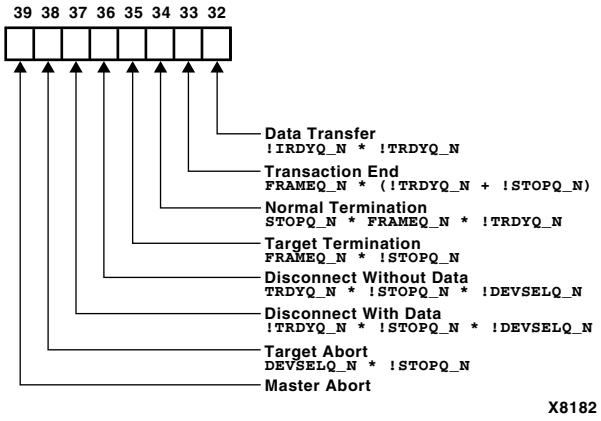
Signal Name	Type	Functional Description
CSR[31:16]	out	Provides access to the status register state bits. These are set automatically by the core interface. Individual status bits are reset by the system software by writing a '1' to the bit location to be reset. All values in the status register are either registered or read-only. Fast back to back transactions are not supported. 
CSR[39:32]	out	Provides access to the transaction status signals. These are an extension of the standard command and status register bits and reflect the status of a PCI transaction. With the exception of "master abort", these status bits reflect any bus activity, as they are derived from registered copies of PCI Bus signals. It is important to note that CSR[38:32] are combinational outputs generated by the equations shown below. 
SUB_DATA[31:0]	in	Performs one of two functions, depending on interface configuration. If the interface is not configured to use external Subsystem ID, this input provides the CardBus CIS Pointer data. If the interface is configured to use external Subsystem ID, this input provides the Subsystem ID data.

Table 2-2: **User Interface Signals *(Continued)***

Signal Name	Type	Functional Description
System Signals		
CLK	out	The PCI Bus clock driven by a clock buffer. Use this clock for all flip-flops that are synchronized to the PCI Bus clock.
RST	out	An inverted copy of the PCI Bus reset signal. This signal should be used as an asynchronous reset signal for the user application.
64-bit Extension		
REQ64Q_N	out active low	A registered version of the PCI Bus REQ64_IO signal.
ACK64Q_N	out active low	A registered version of the PCI Bus ACK64_IO signal.
ADIO[63:32]	t/s	A time multiplexed address and data bus. This bus must be driven using internal three-state buffers.
S_CYCLE64	out	Indicates that the core interface is engaged in a 64-bit target transaction. This signal is asserted at the same time as BASE_HIT and remains asserted until the transaction is complete.
S_CBE[7:4]	out	Indicates the current PCI Bus command or byte enables for a target access. Note that byte enables are active low.
REQUEST64	in	Used to request a 64-bit PCI initiator transaction. Assertion of REQUEST64 causes the PCI interface to assert REQ_O if the bus master enable bit (CSR2) is set in the command register. This bit is cleared at reset.
M_FAIL64	out	Indicates that a 64-bit initiator transfer attempt has encountered a 32-bit target. In this case, the initiator transfers two 32-bit words (at most) before terminating the transfer. This signal should be used to adjust the increment value (step size) of initiator address pointers.
SLOT64	in	Used to enable the 64-bit extension. See the appropriate implementation guide for details.
M_CBE[7:4]	in	Used by the initiator to drive byte enables during initiator transactions. Note that byte enables are active low.

Special Requirements

In general, the signals that interface to the user application may be connected as required by the application. However, certain signals have special connectivity requirements.

To ensure a successful design, use the following design rules.

1. Any output signals from the core interface may remain unconnected if they are unused.
2. The user application must connect to the full width of the internal ADIO address and data bus. Do not leave ADIO unconnected.
3. All bits of the PCI interface configuration bus, CFG, must be driven by power or ground.
4. Most input signals to the core interface that are unused or disabled may be connected to power or ground, as appropriate. However, the following signals must never be tied to power or ground.

   Caution! Note that this is the most important rule; the first three design rules are trivial and do not require much attention during the design of a user application.

   - SLOT64
   - S_TERM
   - S_READY
   - S_ABORT
   - COMPLETE
   - M_READY
   - M_WRDN

Signals are often explicitly tied to power or ground through assignment statements, but can be *accidentally* tied to power or ground through optimization of the driving logic.

In cases where the user application would normally assign one of these signals to power or ground, drive the signal from the output of a flip-flop.

For example, in place of:

```
assign WHATEVER = 1'b0;
```

Use the following:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized
   if (RST) WHATEVER = 1'b1;
   else WHATEVER = 1'b0;
end
```

The above requirements stem from the fact that certain pieces of logic inside the core interface, which are connected to these signals, must not be optimized or reduced. Following the rules above will prevent such undesired optimization.

XILINX®

Chapter 3

General Design Guidelines

This chapter provides guidelines for turning a Initiator/Target v3.166 for PCI interface into a fully functioning design integrated with user application logic. A target-only design does not require any of the initiator steps; however, an initiator always requires the target interface. The burst support steps may require four separate sub-steps—read and write operations for both target and initiator.

Design Steps

- Configure the Base Address Register(s)
- Configure the contents of the Configuration Space Header ROM
- Configure the interface options

See Chapter 4, "Customizing the Interface" for information about each design step in the process.

Target Designs

- Build an interface to read and write locations in the user application. See Chapter 5, "Target Data Transfer and Control."
- Create logic to signal various target termination conditions if required by the user application. See Chapter 6, "Target Data Phase Control."
- If the target uses the 64-bit extension, see Chapter 8, "Target 64-bit Extension." If the design is target only, see Chapter 9, "Target Only Designs."

Initiator Designs

- Build an initiator control state machine and the required support logic, and an interface to read and write locations in the user application. See Chapter 10, "Initiator Data Transfer and Control."
- Create logic to control initiator data phases. See Chapter 11, "Initiator Data Phase Control." If the initiator uses the 64-bit extension see Chapter 13, "Initiator 64-bit Extension."

Burst Designs

- Provide pipelined data sources that correctly respond to various target and initiator termination conditions, and build an address counter. See Chapter 7, "Target Burst Transfers," and Chapter 12, "Initiator Burst Transfers."
- Build FIFOs for the specific application, if required.

Advanced Designs

- Implement configuration space registers to support additional features and capabilities, if required. See Chapter 14, "Other Bus Cycles."
- Modify the initiator or target logic to support special bus commands, if desired. See Chapter 14, "Other Bus Cycles."
- Modify design to support operation as a host bridge, if required. See Chapter 14, "Other Bus Cycles."

Know the Level of Difficulty

A fully compliant PCI interface is challenging to implement in any technology and especially so in FPGA devices.

Table 3-3 provides information about the degree of difficulty in implementing various PCI designs, which is sharply influenced by:

- Maximum system clock frequency
- Targeted device architecture
- Nature of the user application

All implementations for PCI need careful attention to system performance requirements. Pipelining, logic mapping, placement constraints, and logic duplication are all methods that help boost system performance.

Table 3-3: **Level of Difficulty with Implementations for PCI**

Device Family	System Clock Frequency	Level of Difficulty
Spartan-3	33 MHz	Easy
	66 MHz	Difficult
Spartan-3E[a] Spartan-3A	33 MHz	Easy
	66 MHz	Moderate
Virtex-4	33 MHz	Easy
	66 MHz	Moderate

a. The XC3S1200E device requires special post-route processing to allow it to meet hold-time requirements in 66 MHz PCI. See the *Initiator/Transfer v3.1 for PCI Getting Started Guide*.

Understanding Signal Pipelining

To meet the stringent PCI performance requirements, theInitiator/Target v3.166 for PCI interface pipelines all of the bus control signals and the data path. Consequently, some signals must be presented up to two clock cycles before they appear on the PCI Bus. Likewise, arriving signals are captured and available to the user application one cycle after they appear on the PCI Bus. Figure 3-3 provides basic guidelines about how the interface for PCI is pipelined.

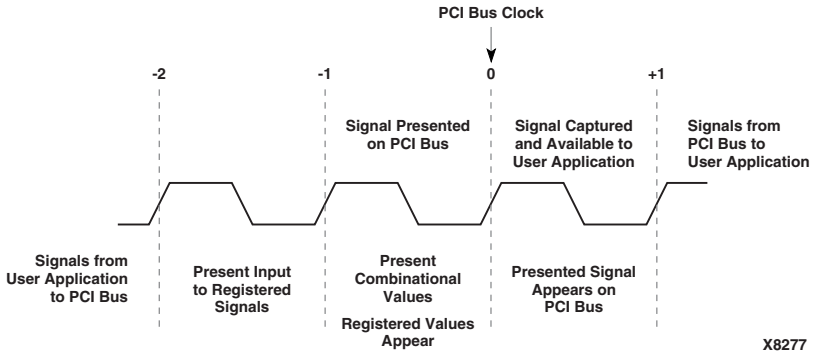


Figure 3-3: **Signal Pipeline Delay**

When the PCI interface receives a signal, it is captured in an input flip-flop to guarantee the setup time as required by the *PCI Local Bus Specification*. The signal is available to the user application one clock cycle after it appears on the PCI Bus. For example, data is captured in input flip-flops and becomes available on the `ADIO` internal bus one cycle after it appeared on the PCI Bus. Signals like `M_DATA_VLD` and `S_DATA_VLD` signal the user application to grab the value from the `ADIO` bus.

When the user application is sending a combinational signal, the signal must be presented one cycle before it is to appear on the PCI Bus. Most of the outputs connected to the PCI Bus originate from an output flip-flop to meet the clock-to-output specification. If the signal first originates from a register in the user application, then the register inputs must be presented two cycles before the signal is to appear on the PCI Bus.

Keep it Registered

To simplify timing and increase system performance in an FPGA design, keep everything registered; all inputs and outputs from the user application should come from, or connect to, a flip-flop. While registering signals may not be possible for all paths, it simplifies timing analysis.

Recognize Timing-Critical Signals

Watch the timing and loading on the signals listed below. Some of these signals are part of the critical timing path and may require special attention when used in the user application.

- **`M_SRC_EN`** and **`S_SRC_EN`**: These signals are combinational outputs of the interface and are the two most time critical signals passed to the user application.
- **`ADDR_VLD`**: A heavily loaded signal.

- **M_DATA_VLD and S_DATA_VLD**: These signals can become heavily loaded in most user applications.
- **C_READY**, **M_READY** and **S_READY**: Connect to the target and initiator state machine control logic through multiple layers of logic. Drive C_READY, M_READY, and S_READY from a flip-flop, if possible.
- **C_TERM and S_TERM**: Connect to the target state machine control logic through multiple layers of logic. Drive C_TERM and S_TERM from a flip-flop, if possible.
- **COMPLETE**: Drives critical logic in the initiator state machine. Pay special attention to reducing delay for this signal.
- **M_ADDR_N**: Connects to the output enables driving the initiator start address onto the ADIO internal bus.

Use Supported Design Flows

The core uses a variety of advanced software features to enhance system performance. These features may include using a guide file to direct the placement and routing of timing-critical logic.

A *guide file* is a fragment of a full design. This fragment guarantees the performance of timing-critical control signals in the target and initiator state machine logic. Guide files are carefully hand-crafted to achieve maximum performance.

The place-and-route (PAR) program matches logic and routing in the final design to logic and routing in the hand-crafted guide file. Individual logic blocks and nets are matched by their instance name and connectivity. For this reason, it is very important that the names used in the final design match those used in the guide file. If the names do not match, the PAR will not be able to guide the full design, and that may adversely affect performance.

To guarantee that critical instance names and net names match those used in the guide file, the design must be processed using a supported design flow. See to the *Initator/Target v3.1 for PCI Getting Started Guide* for a list of supported design flows.

Make Only Allowed Modifications

The Initiator/Target v3.166 for PCI core interface is not user-modifiable. Do not make modifications as they may have adverse effects on system timing and PCI protocol compliance. All modifications must be completed using the web-based Configuration and Download Tool or by hand-editing the configuration file.

Unsupported Devices

If you wish to target a device/package combination that is not officially supported (not listed in the *Initiator/Target for PCI Data Sheet*), you may use the UCF Generator for PCI/PCI-X to create a user constraints file that implements a suitable pinout for your target device.

- For Spartan-3 devices, the UCF Generator for PCI is available on the web from the Xilinx PCI Lounge: www.xilinx.com/products/logicore/lounge/lounge.htm
- For Spartan-3A, Spartan-3ADSP, Spartan-3E and Virtex-4 devices, this tool is available in the Xilinx CORE Generator under **UCF Generator for PCI/PCI-X**. For more information on this tool, consult the *UCF Generator for PCI/PCI-X Data Sheet*.

Note: It is important to verify the UCF files generated by this tool to confirm that the timing requirements of your application are met. Xilinx cannot guarantee that every UCF file generated by the UCF Generator tool will work for every application.

XILINX®

Chapter 4

Customizing the Interface

The Initiator/Target v3.166 for PCI core is customized by using the settings in the configuration text file. Settings in the configuration file are communicated to the core through a 256-bit bus and affect both the design and the functional simulation model. There are two ways to modify the values in the configuration file:

- Using the CORE Generator software graphical user interface
- Using a text editor

For starting new designs, the CORE Generator software GUI is the recommended method; for subsequent changes, you may modify the configuration file using a text editor. The following sections define the customization options.

Note: : Modifications to the `pcim_lc` module are not allowed and may cause designs to fail timing specifications. Modifying any options that are not described in this chapter may result in non-compliant behavior.

Device and Vendor ID

Device ID is a unique identifier for the application. This field can be any value. Change this value for the application.

```
// Device ID and Vendor ID
assign CFG[151:120] = 32'h4062_10ee ;
```

Vendor ID identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI-SIG to guarantee that each identifier is unique. The value `10EEh`, provided in the default configuration, is the Vendor ID for Xilinx. Enter a vendor identification number here. The value `FFFFh` is reserved.

Class Code and Revision ID

Class Code identifies the general function of a device. The value, as provided in the default configuration, identifies the device as a generic co-processor function.

```
// Class Code and Revision ID
assign CFG[183:152] = 32'h0B40_0000 ;
```

Class Code is divided into three byte-size fields, as described in the *PCI Local Bus Specification*. The upper byte broadly identifies the type of function performed by the device.

The middle byte defines a sub-class that more specifically identifies device function. The sub-classes are defined in "Appendix D" of the *PCI Local Bus Specification*.

The lower byte defines a specific register-level programming interface (if any). This allows device-independent software to interact with the device.

The Revision ID indicates the revision of the device or application. It is an extension of the Device ID. Enter the values appropriate for the application.

Subsystem Vendor ID and Subsystem ID

Subsystem Vendor ID further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; typically, it is the same as the Vendor ID. Setting it to 0000h may cause issues with compliance testing. The Subsystem ID can be set as desired to identify the device, revision, or other manufacturing data.

```
// Subsystem ID and SubVendor ID

assign CFG[215:184] = 32'h4062_10ee ;
```

By default, the Subsystem Vendor ID and Subsystem ID are set at design time and part of the resulting interface netlist.

```
// External Subsystem ID and Subvendor ID

assign CFG[114] = `DISABLE ;
```

Enabling the External Subsystem ID allows these fields to be dynamic and supplied by the user application through the SUB_DATA bus. This may be used to load unique values which are determined by the user application at run-time. When this feature is enabled, the CardBus CIS Pointer is disabled and always set to zero.

CardBus CIS Pointer

The CardBus CIS Pointer is used in CardBus applications. By default, this field is supplied through the SUB_DATA bus. If the CardBus CIS Pointer is not used, the SUB_DATA bus should be set to zero. If the interface is configured to use an external Subsystem Vendor ID and Subsystem ID via SUB_DATA, the CardBus CIS Pointer is disabled and always set to zero.

Base Address Registers

The PCI interface supports up to three Base Address Registers (BAR). The designer is free to use any BAR. Each BAR has several attributes, which define:

- Whether the BAR is enabled. Disabling the BAR allows the optimization tools to delete the entire circuit.
- The size of the address space required. In the core interface, the address space can be as small as 16 bytes, or as large as two gigabytes. For 80x86 systems, the maximum allowed I/O space is 256 bytes.
- The ability of memory space to be prefetched. The *PCI Local Bus Specification* defines memory as prefetchable if:
  - there are no side-effects on reads (for example, 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.
- Whether the address space is defined as memory or I/O. The BAR only responds to commands that access the specified address space.

- The address space location "preference" of the device. The core interface supports 32-bit address spaces for both memory and I/O.
- Whether the BAR address space is defined as 64-bit capable. This only applies to memory spaces.

Generally, memory spaces less than 4K in size should use a 4K block size, as recommended in the *PCI Local Bus Specification*. The maximum I/O space allowed is 256 bytes. Some machines may disable a card if it requests more than 256 bytes of contiguous I/O space.

```
// BAR0 -- 64-Bit Capable, 1 Mb Memory Space
assign CFG[0]       = `ENABLE ;
assign CFG[32:1]    = `SIZE1M ;
assign CFG[33]      = `PREFETCH ;
assign CFG[35:34]   = `TYPE00 ;
assign CFG[36]      = `MEMORY ;
assign CFG[241]     = `SPACE64 ;

// BAR1 -- 1 Mb Non-Prefetchable Memory Space
assign CFG[37]      = `ENABLE ;
assign CFG[69:38]   = `SIZE1M ;
assign CFG[70]      = `NOFETCH ;
assign CFG[72:71]   = `TYPE00 ;
assign CFG[73]      = `MEMORY ;
assign CFG[242]     = `SPACE32 ;

// BAR2 -- 16 Byte I/O Space
assign CFG[74]      = `ENABLE ;
assign CFG[106:75]  = `SIZE16 ;
assign CFG[107]     = `IO_PREFETCH ;
assign CFG[109:108] = `IO_TYPE ;
assign CFG[110]     = `IO ;
assign CFG[243]     = `SPACE32 ;
```

Latency Timer Settings

The MAX_LAT and MIN_GNT registers are used to specify the desired latency timer settings. MAX_LAT specifies how often the device needs to gain access to the PCI Bus. MIN_GNT specifies how long the minimum burst period should be, assuming a 33 MHz bus speed. The values for both registers are periods of time in units of a quarter microsecond.

```
// Max_Lat
assign CFG[231:224] = 8'h0f ;
// Min_Gnt
assign CFG[223:216] = 8'h0f ;
```

These registers are intended for use by the system software and do not directly affect the operation of the core.

For applications that do not support bursting or that burst only two words, the Latency Timer function can be disabled. This saves additional logic and routing resources.

```
// Latency Timer Enable
assign CFG[112] = `ENABLE ;
```

Interrupt Enable

This turns on the interrupt registers in the Initiator/Target v3.166 for PCI interface. The interface only supports INTA_O.

```
// Interrupt Enable
assign CFG[113] = `ENABLE ;
```

Capabilities List

The core has the ability to implement a Capabilities List in configuration space. If the design requires a Capabilities List, enable this option and set the pointer to the desired address in configuration space. Otherwise set it to zero. Also enable the User Config Space option.

```
// Capabilities List Enable
assign CFG[116] = `DISABLE ;
// Capabilities List Pointer
assign CFG[239:232] = 8'h00 ;
// User Config Space Enable
assign CFG[118] = `DISABLE ;
```

Note: If multiple clock domains are used, the USER_FFS flip-flops should be limited to those that connect to the coe in the PCI clock domain.

Interrupt Acknowledge

This bit enables the core to respond to interrupt acknowledge cycles as a target. Enabling this feature disables the last available Base Address Register.

```
// Interrupt Acknowledge
assign CFG[240] = `DISABLE ;
```

Reserved Settings

Several option settings are reserved for implementation specific features or for backwards compatibility. Do not modify these option settings. The following are reserved and should not be modified.

XILINX®

Chapter 5

Target Data Transfer and Control

The chapter provides basic design guidelines for building the target portion of an application and demonstrates the logic required in the user application to generate the load and output enable signals for a typical target register.

Before you Begin

The following guidelines are provided to assist you in determining the best methods for building the target portion of a user application. Before you begin, carefully consider what it must accomplish.

Determine the Address Space

In a PCI system, no central address decoding is performed. Instead, the address decoding is distributed across the various agents present in the system.

A target design may request up to three separate address spaces of different types and sizes. These selections are made when the base address registers in the core interface are configured for the user application. How should the user application allocate address space?

Where I/O space is required for PC legacy support, the use of an I/O base address register is unavoidable. In general, it is best to use base address registers located in memory space, for several reasons. They support target burst, larger address spaces, prefetchable reads, and 64-bit data transfers.

User application design complexity increases with multiple base address registers, especially if each address space responds to transactions differently.

Determine the Bandwidth

The core supports both single and burst transfers. What type of bandwidth is required when other bus agents access the user application?

Currently, support for bursting to targets is poor in desktop PC chip sets. Additionally, the host bridge is involved in most, if not all, transactions (as either the initiator or the target). If the user application is intended for use in PC systems, higher bandwidth may be achieved by using single target transfers to set up an initiator burst transfer.

In embedded systems, it may be the case that the host bridge is used for configuration transactions only, and after configuration, agents on the bus burst data to each other in peer-to-peer transactions. In this case, it is critical to support target burst to achieve high bandwidth.

Assemble the Design

This chapter defines the fundamentals of a target design and provides information sufficient for implementing non-burst transfers to registers and peripherals in the user application. Additional information about the target portion of a design is provided in the following chapters:

- Chapter 6, "Target Data Phase Control," provides information about inserting target wait states and generating different types of target termination for more elaborate designs. (This information is useful for both non-burst and burst designs.)
- Chapter 7, "Target Burst Transfers," demonstrates the additional logic required to support target burst transfers for burst designs.
- Chapter 8, "Target 64-bit Extension," discusses the use of the 64-bit extension for increased performance.
- Note that the core interface is both a target and an initiator. In designs where the initiator functions are unused, the initiator control signals must be tied off to benign values.
- Chapter 9, "Target Only Designs," provides information about setting initiator controls to benign values when the initiator functions are unused. (The core interface is both a target and an initiator.)

Where high bandwidth is desirable in a target design, posted writes and prefetched reads are two methods to increase bandwidth. These techniques are discussed in the *PCI Local Bus Specification*. Posted write buffers are generally easy to implement with FIFOs in a user application. Prefetched reads are more complicated in nature and require additional design considerations when the user application data source is not prefetchable.

Target Register Overview

In applications that do not use target burst transactions, data is generally transferred to and from registers in the user application. These registers are connected to control signals required for target data transfer and to any additional control and data path logic provided by the user, and may also connect to internal FIFOs or I/O pins on the user application. Figure 5-1 illustrates a typical target register interface.

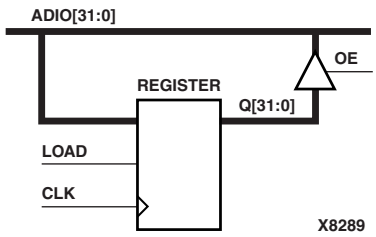


Figure 5-1: **Example Target Register**

Target Interface Signals

The target interface signals control target data transfer to and from the core interface.

Basic Target Interface Signals

For basic transfers, a subset of the target interface signals is used. An example of a basic target design using the reduced subset is presented later in this chapter.

- **BASE_HIT[7:0]**: Input indicates that one of the base address registers recognizes that it is the target of a current transaction. This is the first indicator to the user application that a target transaction is about to begin.
- **ADIO[31:0]**: Bidirectional bus provides the means for data and address transfer to and from the core interface.
- **ADDR[31:0]**: Input is a registered version of the PCI address provided by the core interface. It becomes valid in the cycle after ADDR_VLD is asserted, and remains valid through the entire transaction.
- **S_WRDN**: Input indicates the direction of data transfer for the current target transaction. Logic high indicates that the user application is sinking data (such as target write). It is valid during the cycle BASE_HIT is asserted and is held through the entire transaction.
- **S_CBE[3:0]**: A registered version of the CBE_IO lines, and is delayed by one cycle. It indicates the PCI command and byte enables during a target transaction. This signal is used primarily for byte enable information, as the command is decoded and latched in PCI_CMD during the address phase of the transaction.
- **PCI_CMD[15:0]**: Input is a decoded and latched version of the PCI command for the current bus transaction.
- **S_DATA_VLD**: Input has two interpretations depending on the direction of data transfer. When the user application is sinking data (target writes), S_DATA_VLD indicates that the user application should capture valid data from the ADIO bus. When the user application is sourcing data (target reads), S_DATA_VLD indicates that a data phase has completed on the PCI Bus.
- **S_DATA**: Input indicates that the target state machine is in the data transfer state.

Additional Target Interface Signals

More elaborate designs involving non-deterministic target termination, target wait state insertion, or target burst require additional target interface signals. Examples of more complex target designs are presented in later chapters. (Note that basic target interface signals are also used in more complex designs.) References to inputs and outputs are made with respect to the user application.

- **ADDR_VLD**: Input indicates that a valid PCI address is available on the ADIO bus, and may be used as a clock enable by the user application to capture a copy of this address. This is particularly useful in target burst applications where a loadable counter must track the target address. In non-burst applications, the latched address present on the ADDR bus may suffice. Note, however, that the assertion of ADDR_VLD does not mean that the user application will be the selected target. The ADDR_VLD signal is asserted for a single cycle.
- **S_SRC_EN**: Input is only used during target burst reads. It indicates to the user application that the data source which drives output data onto the ADIO bus must

provide the next piece of data. In most applications, this signals the user application to advance the data pointer for the data source providing the data.

- **CSR[39:0]**: Input provides general status information about the core interface. The high eight bits provide status information about the current transfer. This status information is used primarily in target burst applications with non-prefetchable sources to determine if any associated address pointers must be "backed up."
- **S_READY**: Output from the user application indicates that it is ready to transfer data, and can be used to insert wait states during the first data phase of a transaction. Together with S_TERM, it is also used to signal different types of target termination. It is important to note that the user application is prohibited from using S_READY to insert wait states after the first data phase.
- **S_TERM**: Output from the user application indicates that data transfer should cease. It is also used with S_READY to signal different types of target termination.
- **S_ABORT**: Output from the user application indicates that a serious (fatal) error condition has occurred and that the current transaction must stop.

The following signals are output by the target state machine in the core interface. These states are defined in "Appendix B" of the *PCI Local Bus Specification*.

- **IDLE**: Input indicates that the target state machine is in the idle state and that there is no activity on the PCI Bus.
- **B_BUSY**: Input indicates that the target state machine has recognized the beginning of a PCI Bus transaction. The target state machine will change to the S_DATA state if it determines that it is the target of the transaction.
- **S_DATA**: Input indicates that the target state machine is in the data transfer state.
- **BACKOFF**: Input indicates that the target state machine is waiting for a transaction to complete because the user application has asserted S_TERM.

Decoding Target Transactions

The user application is responsible for monitoring outputs from the core interface to respond to target transactions. The signals used in target transactions are active and available at different times. The most important signal is BASE_HIT[x], which indicates that the core interface has claimed the current PCI transaction for base address register **x**. It is asserted for a single cycle. The following logic decodes target reads and writes directed at base address register x.

```
always @(posedge CLK or posedge RST)
begin : decode
    if (RST)
    begin
        BAR_x_RD = 1'b0;
        BAR_x_WR = 1'b0;
    end
    else
    begin
        if (BASE_HIT[x])
        begin
            BAR_x_RD = !S_WRDN & OPTIONAL;
```

```
            BAR_x_WR = S_WRDN & OPTIONAL;
        end
        else if (!S_DATA)
        begin
            BAR_x_RD = 1'b0;
            BAR_x_WR = 1'b0;
        end
    end
end

assign OPTIONAL = fn(PCI_CMD[15:0]) & fn(ADDR[31:0]);
```

The optional term is for sub-decode and allows the address space allocated by a base address register to be mapped into multiple regions. If the user application performs sub-decoding of the address space, the above logic needs to be replicated to cover each region. Do not generate *holes* in the address space. If desired, the user application can also distinguish between different types of PCI read and write commands.

The last clause in the decoding block holds the decode asserted throughout the entire data transfer state. Again, BASE_HIT[x] is only active for a single clock cycle at the start of the transaction. Effectively, the code above describes a synchronous set/reset flip-flop with set dominant.

Target Writes

During a target write operation, data is captured from the ADIO bus to a data register in the user application by asserting the load input. The first step is to generate the BAR_x_WR signal as described in the previous section on decoding target transactions.

The critical gating signal is S_DATA_VLD. It is the final signal required to qualify the write operation. Consequently, the other signals can be decoded earlier and gated with S_DATA_VLD. The assignment for the load input of the register would then be:

```
assign LOAD = BAR_x_WR & S_DATA_VLD;
```

Decoding BAR_x_WR and registering it before the assertion of S_DATA_VLD allows more time for routing and reduces the number of logic levels from the critical input, S_DATA_VLD. If the user application supports byte-addressable registers, separate load signals should be generated for each byte in the register by further gating the expression shown above. The S_CBE signals are available for this purpose.

The time relationship is shown in the waveform of Figure 5-2. This waveform includes both PCI Bus signals and internal user application signals.

Figure 5-2: **Target Write Transaction**

Target Reads

During a target read operation, data from the user application is driven onto the ADIO bus. To do this, the user application must assert the output enable for the desired register. The first step is to generate the BAR_x_RD signal on decoding target transactions. The assignment for the output enable would then be:

```
assign OE = BAR_x_RD & S_DATA;
```

Note: Ensure that all output enables are deasserted at system reset to avoid possible bus contention.

Do not qualify the output enable signal with the S_CBE signals even if the user application supports byte-addressable registers; drive the entire ADIO bus with valid data.

Although S_DATA_VLD is not used in the output enable logic, the user application may monitor this signal during target reads. The assertion of S_DATA_VLD during a target read indicates that the initiator has acknowledged the data transfer. This is useful in designs where a data source must change state after it is read.

The time relationship is shown in the waveform of Figure 5-3. This waveform includes both PCI Bus signals and internal user application signals.

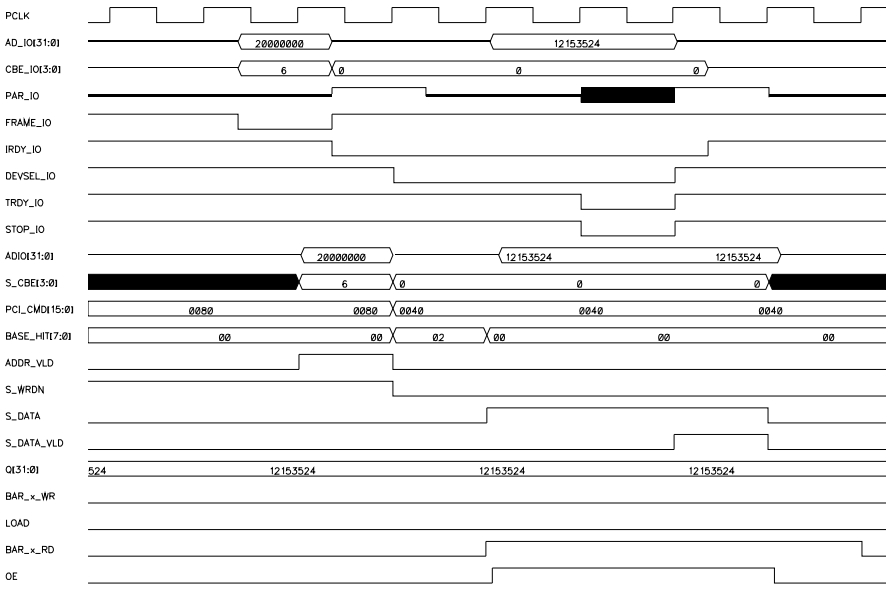


Figure 5-3: **Target Read Transaction**

Terminating Target Transactions

In general, PCI transactions may be terminated by the initiator or the target. In the specific case of non-burst target transactions, the core interface must terminate the transaction after the first data phase even if the initiator wishes to continue. Target terminations are controlled using the S_READY, S_TERM, and S_ABORT signals.

The S_READY, S_TERM, and S_ABORT signals must not be assigned static values. For timing reasons, these signals should be driven from the output of a flip-flop, although it is permitted to drive these signals from combinational logic.

Note: Never tie these signals to logic one or to logic zero.

The best way to achieve termination after the first data phase is to instruct the core interface to always disconnect with data:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
   if (RST) S_ABORT = 1'b1;
   else S_ABORT = 1'b0;
end

always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
   if (RST) S_READY = 1'b0;
   else S_READY = 1'b1;
end

always @(posedge CLK or posedge RST)
```

```
begin : cannot_be_optimized_c
   if (RST) S_TERM = 1'b0;
   else S_TERM = 1'b1;
end
```

This causes all target transactions to terminate after a single data phase. This termination behavior is exhibited in the target write and read transactions, as shown in Figure 5-2 and Figure 5-3.

This type of deterministic termination is sufficient for simple user application designs that do not support non-deterministic target termination, target wait state insertion, or target burst.

Chapter 6

Target Data Phase Control

This chapter discusses the mechanism by which the user application can control various aspects of target transactions to accommodate its own ability to source or sink data.

Target initiated wait states allow a user application additional time before the first data transfer, and target initiated terminations allow the user application to limit the number of data phases in a transaction. These features are useful in both burst and non-burst target designs.

Control Modes

Data phase control is achieved using the S_TERM and S_READY signals. Combinations of the two control signals yield the following four modes:

- **Wait**. Inserts wait states at the beginning of a PCI Bus transaction (holds off the first data phase) by delaying the assertion of TRDY_IO by the core interface.
- **Normal**. Allows PCI Bus data phase(s) to complete without the insertion of extra wait states or termination by the target.
- **Disconnect without data**. Terminates the current PCI Bus transaction without data transfer on the final data phase. A disconnect without data on the first data phase is equivalent to a retry.
- **Disconnect with data**. Terminates the current PCI Bus transaction with data transfer on the final data phase.

The exact disconnect sequence is affected by whether or not the initiator also terminates the transaction. The core interface will automatically generate the correct behavior. Note that the core interface will immediately disconnect with data if it receives a transaction using the non-linear addressing mode.

Table 6-1 shows the four modes of operation and the corresponding S_TERM and S_READY values.

Table 6-1: **Data Phase Control Signals for Targets**

Condition	Bus Signals	From User Application	
		S_TERM	S_READY
Wait	TRDY_IO = 1 DEVSEL_IO = 0 STOP_IO = 1	Low	Low

Table 6-1: **Data Phase Control Signals for Targets *(Continued)***

Condition	Bus Signals	From User Application	
		S_TERM	S_READY
Normal	TRDY_IO = 0 DEVSEL_IO = 0 STOP_IO = 1	Low	High
Disconnect Without Data (Retry)	TRDY_IO = 1 DEVSEL_IO = 0 STOP_IO = 0	High	Low
Disconnect With Data	TRDY_IO = 0 DEVSEL_IO = 0 STOP_IO = 0	High	High

Changing from one mode to another must not be done in an arbitrary sequence. In addition, the timing of mode transitions is critical to ensure precise control over the number of data phases that occur in a transaction, particularly when changing to a disconnect mode.

The permitted data phase control sequences for target designs using the core interface are shown in Figure 6-1. After the target has entered into the data phase control sequence indicated in Figure 6-1, it must follow this sequence until the transaction ends. You can monitor the transaction status via the S_DATA and BACKOFF signals. By logically ORing these signals together, you will know that a transaction is currently in progress. After S_DATA and BACKOFF are both deasserted, the transaction has ended.

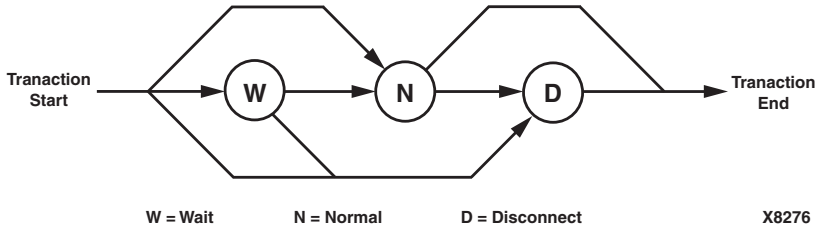


Figure 6-1: **Permitted Data Phase Control Sequences**

The Wait mode cannot be used to insert wait states during arbitrary data phases in a transaction—it may only be used to delay the completion of the first data phase of a transaction. This is called initial latency. The target is required to complete the first data phase of a transaction within 16 clocks from the assertion of FRAME_IO. The user application is responsible for observing this requirement.

Note: During target state machine activity, do not violate the mode sequencing shown above. When the target state machine is inactive, S_TERM and S_READY are ignored by the core interface.

Control Pipeline

In order to meet the stringent PCI Bus performance requirements, the core interface pipelines all of the bus control signals and the data path. Consequently, the S_TERM and S_READY signals must be presented one cycle in advance of the desired effect. See Figure 6-2.

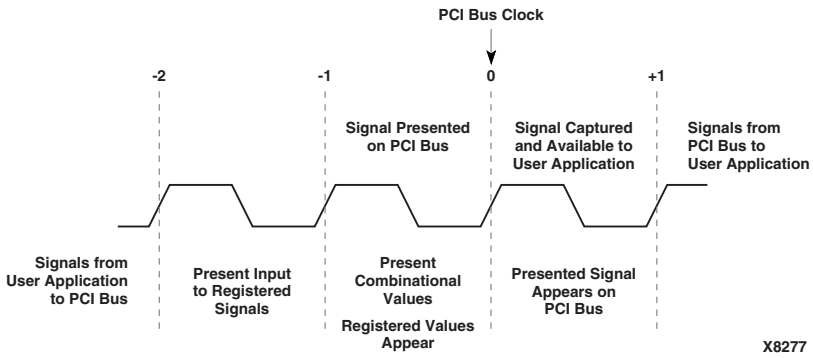


Figure 6-2: **Control Signal Pipeline Delay**

The signals S_TERM and S_READY connect to the target state machine through multiple levels of logic. For this reason, it is highly recommended that these signals are driven directly from the output of a flip-flop. This adds an extra cycle of latency.

Deterministic Control

Deterministic control refers to cases where the initial response of a user application to a target transaction does not depend on the parameters of the transaction. That is, the logic that drives S_TERM and S_READY does not use any information about the incoming target transaction to control the initial data phase. Such information includes, but is not limited to:

- Base address register number
- Target transaction address
- Target transaction command

In these cases, the user application anticipates a target transaction before the transaction occurs or selects control modes based on internal state information.

Deterministic control is easy to implement and simplifies timing considerations. While it is possible to feed transaction specific information back to the S_TERM and S_READY outputs in time for the first data phase, doing so results in driving S_TERM and S_READY with several levels of combinational logic. This is not recommended.

For deterministic control, the S_TERM and S_READY signals must be set properly before any transaction begins (at the time BASE_HIT is asserted). The following examples demonstrate S_TERM and S_READY generation for several common cases.

Example 1: Always Ready, Single Transfers

A simple user application that does not perform target burst transactions and is always ready to transfer data, as discussed in earlier chapters, can instruct the core interface to automatically disconnect with data on the first data phase.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
   if (RST) S_READY = 1'b0;
   else S_READY = 1'b1;
end

always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
   if (RST) S_TERM = 1'b0;
   else S_TERM = 1'b1;
end
```

An example of such an application is a bank of registers mapped in memory or I/O space. The correct control is achieved by assigning S_READY and S_TERM to values that result in disconnect with data, as shown in Table 6-1, page 63.

Example 2: Always Ready, Burst Transfers

Another very simple method of control may be used where the user application does perform target burst transactions and is always ready. For this case, the user application can instruct the core interface to proceed normally.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
   if (RST) S_READY = 1'b0;
   else S_READY = 1'b1;
end

always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
   if (RST) S_TERM = 1'b1;
   else S_TERM = 1'b0;
end
```

This allows the initiator of a transaction to burst data at full speed until it is done or its latency timer expires. The user application must be capable of sourcing or sinking data at full speed. The target may disconnect by adding logic to assert S_TERM. This behavior is commonly found in target designs that perform posted writes.

Example 3: Initial Latency

This is similar to Example 1, with the addition of initial latency. The initial latency may be fixed by a counter or may be variable and determined by other internal state. In cases where the initial latency is variable, the user application must not cause the core interface to violate the *PCI Local Bus Specification* limit on initial latency. The target is required to complete the first data phase of a transaction within 16 clocks from the assertion of FRAME_IO.

The following Verilog pseudocode demonstrates how to insert initial wait states:

```
always @(posedge CLK or posedge RST)
begin : bl_timer
   if (RST) TIMER = 4'h0;
   else if (ADDR_VLD) TIMER = BL_WAIT[3:0];
   else if (TIMER != 4'h0) TIMER = TIMER - 4'h1;
end

assign S_READY = (TIMER <= 4'h3);
assign S_TERM = (TIMER <= 4'h3);
```

The start of any transaction on the PCI Bus is marked by the falling edge of FRAME_IO. This information is needed to reload the initial wait timer. A registered version of FRAME_IO is available to the user application as FRAMEQ_N. However, this signal is heavily loaded within the core interface. Fortunately, ADDR_VLD is asserted with the falling edge of FRAMEQ_N during target transactions, so ADDR_VLD may be used instead.

At the beginning of all transactions, the logic sets S_TERM and S_READY to the proper values in advance of target state machine signalling a pending target transaction. In non-target transactions, S_TERM and S_READY are simply ignored.

The latency is determined by the value present on BL_WAIT, and is expressed in cycles after FRAME_IO is asserted. The timeout does not occur when the timer reaches zero, due to the latency in detecting the start of a transaction and the latency involved in propagating the S_TERM and S_READY signals through the core interface to the PCI Bus.

This example can easily be extended to allow multiple transfers, as shown in "Example 2: Always Ready, Burst Transfers" by removing the assignment for S_TERM and adding the following:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
   if (RST) S_TERM = 1'b1;
   else S_TERM = 1'b0;
end
```

Bounded initial latency is useful in user application designs that access off chip registers or peripherals.

Example 4: Retries

In some user application designs, it may be necessary to retry target transactions if the data source or sink cannot be ready within the initial latency bounds given in the *PCI Local Bus Specification*. This situation may arise with very slow peripherals or if the user application implements delayed reads. In such cases, the initiator is bound by PCI protocol to retry the original transaction at a later time.

When the user application is ready to transfer data, S_TERM and S_READY may be generated as required. The following Verilog pseudocode provides an example, using the BLAT_RDY signal generated from "Example 3: Initial Latency."

```
always @(posedge CLK or posedge RST)
begin : keep_it_registered
   if (RST)
```

```
        begin
            S_READY = 1'b1;
            S_TERM = 1'b0;
        end
        else
        begin
            S_READY = RETRY ? 1'b0 : BLAT_RDY;
            S_TERM  = RETRY ? 1'b1 : BLAT_RDY;
        end
    end

    assign RETRY = fn(PERIPHERAL_STATE);
```

This code generates retries if the peripheral is not ready. If the peripheral is ready, the user application disconnects with data after several cycles of initial latency.

Care must be taken to ensure that RETRY is valid before the initial data phase of a target transaction and that it does not change during the transaction. One method to produce this result is to sample the peripheral state at the beginning of each PCI Bus transaction, in the same way that the initial latency timer is loaded in "Example 3: Initial Latency."

Other Possibilities

These examples are only a subset of the possible data phase control schemes. The ideas presented above may be combined in many ways to control the flow of data across the target interface so that the data rate is acceptable to the user application.

Note: Do not forget to account for the latency in S_TERM and S_READY. This latency is two cycles if S_TERM and S_READY are registered inside the user application, as is recommended. Otherwise, the latency is one cycle.

Non-Deterministic Control

Non-deterministic control refers to cases where the initial response of a user application to a target transaction depends on the parameters of the transaction. Such information includes, but is not limited to:

- Base address register number.
- Target transaction address.
- Target transaction command.

In these cases, the user application does not know how to respond to a target transaction until the transaction has already started. The user application must feed transaction specific information back to the S_TERM and S_READY outputs in time for the first data phase. The difficulty in this is created by the latency in S_TERM and S_READY.

To feed this information back in time to control the first data phase, S_TERM and S_READY must be driven with combinational logic and not from flip-flops. This is because the S_TERM and S_READY must be presented in the same cycle that a target transaction is detected (when a BASE_HIT signal is asserted). Although this is not recommended, it is possible if the logic is very fast.

For non-deterministic control, the S_TERM and S_READY signals must be presented as soon as one of the BASE_HIT signals is asserted. That is, S_TERM and S_READY will be combinational functions of BASE_HIT and other signals from the user application. The restrictions shown in Figure 6-1 still apply.

Driving S_READY and S_TERM with combinational logic is highly discouraged for timing reasons. In cases where transaction specific information must be fed back to the core interface to control the initial data phase, an alternative method exists at the expense of performance.

The solution is to insert a single wait state at the beginning of all target transactions. The extra cycle in latency allows S_TERM and S_READY to be registered. Although performance is reduced, potential timing problems are avoided.

If, for performance reasons, the suggested method is not feasible, the following examples illustrate how to drive S_TERM and S_READY with combinational logic.

Example 5: Subdividing an Address Space

Consider a user application design where the address space assigned to a base address register is partitioned into two regions. The first region maps to a peripheral device which supports bursts but may require retries, and the second region maps to registers that require immediate disconnect with data. The data phase control signals for the peripheral are P_READY and P_TERM, while the data phase control signals for the registers are R_READY and R_TERM.

```
assign PERIPH_ADDR = fn(ADDR[31:0]);
assign RETRY = fn(PERIPHERAL_STATE);

assign P_READY = RETRY ? 1'b0 : 1'b1;
assign P_TERM = RETRY ? 1'b1 : 1'b0;
assign R_READY = 1'b1;
assign R_TERM = 1'b1;

assign S_READY = PERIPH_ADDR ? P_READY : R_READY;
assign S_TERM = PERIPH_ADDR ? P_TERM : R_TERM;
```

This logic can be greatly reduced. It is presented this form for clarity. As in Example 4, care must be taken to ensure that RETRY is valid during the cycle a BASE_HIT signal is asserted, and that it does not change during the transaction.

As an alternative approach, use an additional base address register (if available) instead of subdividing the address space of a single base address register.

Example 6: Multiple Base Address Registers

Consider a user application design with two base address registers, "x" and "y". The first maps to a peripheral device which supports bursts but may require retries, and the second maps to registers that require immediate disconnect with data. In concept, this is similar to Example 5, but is complicated by the fact that the BASE_HIT signals are only valid for a single cycle, unlike the ADDR bus which is valid throughout a transaction.

```
always @(posedge CLK or posedge RST)
begin : similar_to_decode
```

```
    if (RST) BAR_x_IN_USE = 1'b0;
    else if (BASE_HIT[x]) BAR_x_IN_USE = 1'b1;
    else if (!S_DATA) BAR_x_IN_USE = 1'b0;
end

assign PERIPH_ADDR = BASE_HIT[x] | BAR_x_IN_USE;
```

The final multiplexing logic shown in Example 5 would use this version of PERIPH_ADDR as the select signal.

Target Abort

There is a special type of target termination called target abort. It informs the initiator that the target cannot perform the requested transaction. A target abort is a serious error and signals that data may have been lost or corrupted.

When the S_ABORT signal is asserted from the user application, the PCI interface automatically signals the target abort condition on the PCI Bus and sets the signaled target abort bit (CSR27) in the status register. The user application must continue to assert S_ABORT until the transaction is complete.

The S_ABORT signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic. When target aborts are not used, drive the S_ABORT signal as shown below:

```
always @(posedge CLK or posedge RST)
begin : not_using_abort
    if (RST) S_ABORT = 1'b1;
    else S_ABORT = 1'b0;
end
```

Chapter 7

Target Burst Transfers

Performing a single data transfer across the PCI Bus is the simplest type of transaction. However, because of the overhead of distributed address decoding, this wastes valuable bus bandwidth. The performance advantage in PCI is derived from burst transactions, where two or more data words are transferred during the transaction.

Building a user application that supports single target transfers is the easiest to design. Building a user application that supports target burst transfers is significantly more complex, but worth the effort, if maximum bandwidth is the goal.

Keeping Track of the Address Pointer

In a PCI transaction, only the starting address is broadcast over the bus. For single transfers, this is sufficient. For burst transfers, however, the user application must keep track of the current address.

If the user application performs target burst transfers, then it must keep a local copy of the current address pointer and increment it after every successful data transfer cycle. Luckily, this counter can be small, depending on the address block size of the target (set in the base address register). The counter must be able to support bursts throughout the entire address range of the base address register.

For example, if the target decodes a 4 Kb block of memory space, then the address counter needs to keep track of addresses within the 4 Kb block. A 10-bit loadable binary counter will suffice. Although 4 Kb requires 12-bits to cover the address space, bits 0 and 1 will always equal zero for 32-bit transfers over the PCI Bus. If an initiator attempts to burst beyond the 4 Kb block boundary, then the target should issue a disconnect so that when the initiator resumes, the next address does not fall in the range of this base address register.

The upper 20 bits of the address pointer are a simple register, loaded during the address cycle of the transaction when ADDR_VLD is asserted. Likewise, the starting address within the address block is loaded into the 10-bit binary counter during the address cycle. The upper 20 bits, which are seldom required in the user application, can be eliminated.

Table 7-1: **Example Target Address Pointer**

31 12	11 2	1	0
20-bit address register (optional)	10-bit loadable binary counter	0	0

During target writes, the counter portion of the target address pointer should be incremented when S_DATA_VLD is asserted. During target reads, the address pointer

should be incremented when S_SRC_EN is asserted. A full example of the count enable logic is presented later in this chapter.

If the user application supports multiple base address registers, a single address pointer should suffice; but the counter must support the largest block of address space. If one base address register supports a 4 Kb block while the other supports a 16 Mb block, then the counter must support the 16 Mb block, which requires a 22-bit loadable binary counter.

Sinking Data in Burst Transfers

During target writes, the core interface transfers burst data using a pipelined data path. The data valid signal, S_DATA_VLD, is used to advance the target address pointer (and any other data pointers in the user application logic). At the same time the target data pointer is advanced, the user application also captures valid data from the internal ADIO bus.

Using S_DATA_VLD to capture burst data is very similar to the simple case of single transfers. The user application must enable different registers or RAM addresses based on the target address pointer.

A target burst write is shown in Figure 7-1. This waveform includes both PCI Bus signals and internal user application signals.

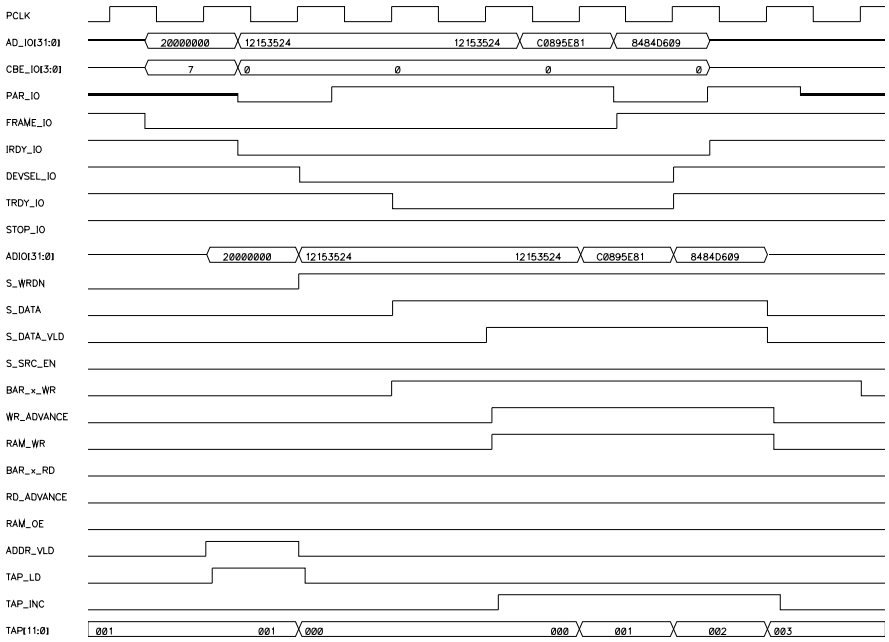


Figure 7-1: **Target Burst Write Transaction**

Sourcing Data in Burst Transfers

During target reads, the core interface transfers burst data using a pipelined data path. The data source enable signal, S_SRC_EN, is used to advance the target address pointer (and any other data pointers in the user application logic). The result is that the user application drives new data onto the internal ADIO bus.

Internally, the core interface captures the data value provided by the user application on the ADIO bus and holds this value in the output flip-flops driving the AD_IO pins. The user application then presents the next data word on ADIO, instead of holding the previous data

word until the current data phase completes. This approach results in higher data throughput.

A target burst read is shown in Figure 7-2. This waveform includes both PCI Bus signals and internal user application signals.

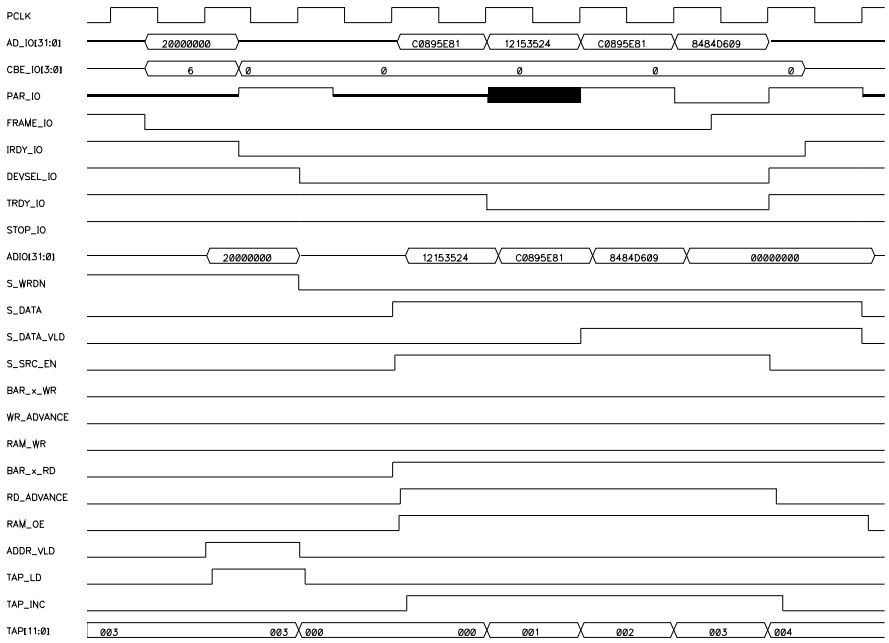


Figure 7-2: **Target Burst Read Transaction**

Using S_SRC_EN to present data for the next data phase may require additional control logic, depending on the type of data source present in the user application. The S_SRC_EN signal advances the target address pointer in anticipation of the next data phase, which may or may not complete with successful data transfer.

If the target address pointer is advanced, and the data is never transferred, then the user application must decide what to do with the non-transferred data. In the case of pre-fetchable data sources (such as RAM or a register file) the data can be discarded. The original data remains in the RAM or the register file for future use.

This also applies in cases where a FIFO is used as a rate matching buffer and the contents of the FIFO are flushed after a transaction. Any non-transferred data is discarded from the FIFO, but the original data still remains in the source that originally provided it. This technique is used in designs that implement PCI delayed read requests.

For non-prefetchable data sources, as is the case when a FIFO itself is the data source, pulling data out of the FIFO may be destructive. The unused data must be restored so it is available for future use should it not be transferred. This may require decrementing internal counters or keeping a shadow copy of the previous data values.

With a non-prefetchable data source, this condition may arise at the end of a target read burst transfer, particularly when the transaction is terminated by the initiator. In this case, the user application is not immediately aware of the termination condition, and will have advanced the data source too many times.

One way to determine the number of times the target address pointer has been over-advanced during a burst read is to monitor the difference in the number of cycles S_SRC_EN and S_DATA_VLD have been asserted during a transaction. During target

reads, the signal S_DATA_VLD represents the number of data phases that actually complete with data transfer.

Design Examples

The following design examples demonstrate the use of pre-fetchable and non-prefetchable data sources.

Example 1: Prefetchable Data Source

Prefetchable data sources, such as RAM and general purpose register files, do not exhibit side effects from reads (that is, the state is not altered). Figure 7-3 shows a target address pointer with a simple RAM as they might appear in a user application.

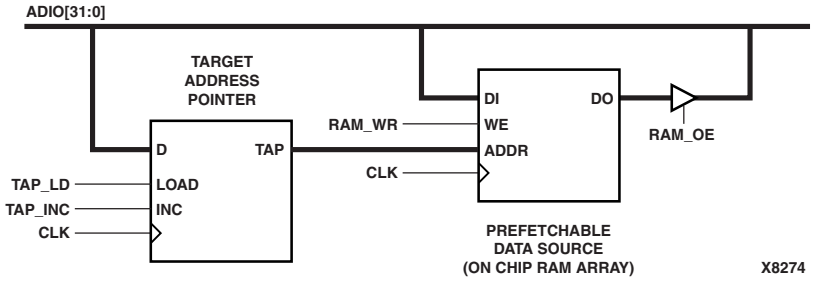


Figure 7-3: **Prefetchable Data Source**

Assume that the target address space, as specified in the base address register, is 4 Kb. After implementing the target transaction decode logic that generates BAR_x_RD and BAR_x_WR, the next step is to implement the target address pointer.

In the specific case of a 4 Kb address space, the two lowest bits of the address counter are always logic zero and do not need to be explicitly represented. The middle ten bits must be implemented as a loadable counter, and the high twenty bits are optional.

```
assign RD_ADVANCE = BAR_x_RD & S_SRC_EN;
assign WR_ADVANCE = BAR_x_WR & S_DATA_VLD;
assign TAP_INC = RD_ADVANCE | WR_ADVANCE;
assign TAP_LD = ADDR_VLD;

always @(posedge CLK or posedge RST)
begin : target_address_pointer
   if (RST) TAP = 10'h0;
   else if (TAP_LD) TAP = ADIO[11:2];
   else if (TAP_INC) TAP = TAP + 10'h1;
end
```

The remainder of the design is trivial. The target address pointer, TAP, is routed to the address input of the RAM array. The remaining control signals may be generated as shown below.

```
assign RAM_OE = BAR_x_RD & S_DATA;
assign RAM_WR = BAR_x_WR & S_DATA_VLD;
```

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
   if (RST) S_READY = 1'b0;
   else S_READY = 1'b1;
end

always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
   if (RST) S_TERM = 1'b1;
   else S_TERM = 1'b0;
end
```

The waveforms shown in Figure 7-1, and Figure 7-2 are the result of this example.

In this design, an initiator can perform burst data transfers to and from the RAM throughout the entire address range of the target. If the initiator attempts to continue a burst beyond the 4 Kb address boundary, the target address pointer wraps around and transfers will continue from address zero. This behavior is undesirable; the user application should signal a target abort and disable the RAM_WR signal when a wraparound is detected.

Example 2: Non-Prefetchable Data Source

In Xilinx FPGAs, FIFOs to support PCI burst transfers are efficiently implemented using the distributed on-chip RAM. In user applications employing FIFOs for target burst, the burst size (and aggregate bandwidth) is limited by the depth of the FIFOs. Most FIFO designs of 64 entries deep or less are feasible. Larger FIFOs are possible depending on the FIFO configuration and the device speed grade.

Non-prefetchable data sources, such as FIFOs, exhibit "side effects" from reads (that is, the state is altered or lost). Special care must be taken during target burst reads so that state information is not lost. The use of S_SRC_EN results in reading the data source ahead of the actual transfer. Unless special precautions are taken, the data will be lost.

Figure 7-4 shows a FIFO suitable for target burst reads in a user application. To present a concise example, this example uses a single FIFO with both ports accessible through the PCI interface. In practice, the best structure for most applications is a dual-FIFO design with separate read and write FIFOs.

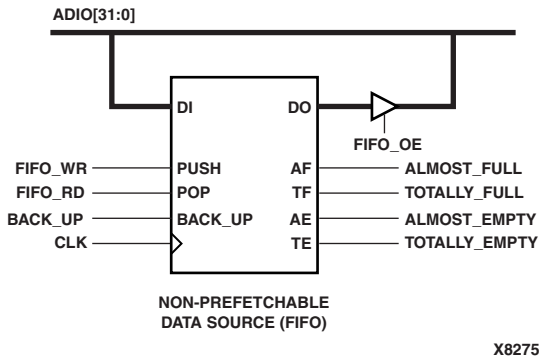


Figure 7-4: **Non-prefetchable Data Source**

As in the previous example, the user application must decode the target access and generate appropriate read and write signals.

```
assign FIFO_OE = BAR_x_RD & S_DATA;
assign FIFO_WR = BAR_x_WR & S_DATA_VLD;
assign FIFO_RD = BAR_x_RD & S_SRC_EN;
```

The most crucial element is the FIFO. It must have an additional control signal to back up, (or undo) up to two reads. A typical FIFO implementation consists of a circular buffer implemented with RAM, a read pointer, and a write pointer.

The backup feature can be incorporated in a FIFO by making the actual depth two less than the possible maximum, and by using a bi-directional read pointer. This prevents new data entering the FIFO from overwriting existing data that may need to be restored.

The FIFO must also have a set of flags that provide FIFO status information. These flags, and their uses, are listed below:

- **TE**. Indicates a totally empty condition. If the FIFO is totally empty at the beginning of a read transaction, the user application should respond with a retry.
- **TF**. Indicates a totally full condition. If the FIFO is totally full at the beginning of a write transaction, the user application should respond with a retry.
- **AE**. Indicates an almost empty condition. When the FIFO becomes almost empty during a read transaction, the user application should signal a disconnect. The threshold for almost empty depends on the type of disconnect that is signalled (with or without data).
- **AF**. Indicates an almost full condition. When the FIFO becomes almost full during a write transaction, the user application should signal a disconnect. The threshold for almost full depends on the type of disconnect that is signalled (with or without data).

The logic for `S_TERM` and `S_READY` is then a function of the four FIFO flags and the signal `S_WRDN`. Again, the exact implementation depends on the version of the PCI interface and the desired type of disconnect, but all of the techniques demonstrated in earlier chapters apply.

To determine the number of times the FIFO must be backed up after a target read, the user application must include a small state machine to monitor the difference between anticipated transfers and actual transfers. When a target read has completed, the state machine should back up the FIFO if necessary.

```
assign ANTICIPATED = BAR_x_RD & S_SRC_EN & !TE;
assign ACTUAL = BAR_x_RD & S_DATA_VLD;

always @(posedge CLK or posedge RST)
begin : oops_counter
   if (RST) OOPS = 2'b00;
   else
   case({ANTICIPATED, ACTUAL, BACK_UP, OOPS})
      5'b00000: OOPS = 2'b00;
      5'b00001: OOPS = 2'b01;
      5'b00010: OOPS = 2'b10;
      5'b00011: OOPS = 2'b11;
      5'b00100: OOPS = 2'b00;
      5'b00101: OOPS = 2'b00;
      5'b00110: OOPS = 2'b01;
      5'b00111: OOPS = 2'b10;
      5'b01000: OOPS = 2'b00;
      5'b01001: OOPS = 2'b00;
      5'b01010: OOPS = 2'b01;
      5'b01011: OOPS = 2'b10;
      5'b01100: OOPS = 2'b00;
      5'b01101: OOPS = 2'b00;
      5'b01110: OOPS = 2'b00;
      5'b01111: OOPS = 2'b01;
      5'b10000: OOPS = 2'b01;
      5'b10001: OOPS = 2'b10;
      5'b10010: OOPS = 2'b11;
      5'b10011: OOPS = 2'b11;
      5'b10100: OOPS = 2'b00;
      5'b10101: OOPS = 2'b01;
      5'b10110: OOPS = 2'b10;
      5'b10111: OOPS = 2'b11;
      5'b11000: OOPS = 2'b00;
      5'b11001: OOPS = 2'b01;
      5'b11010: OOPS = 2'b10;
      5'b11011: OOPS = 2'b11;
      5'b11100: OOPS = 2'b00;
      5'b11101: OOPS = 2'b00;
      5'b11110: OOPS = 2'b01;
      5'b11111: OOPS = 2'b10;
      default : OOPS = 2'b00;
   endcase
end
```

```
assign BACK_UP = (| OOPS) & !S_DATA;
```

This state machine describes a two bit saturating up/down counter with one increment input and two decrement inputs. As required, it tracks the number of actual transfers versus the number of anticipated transfers. The BACK_UP signal is asserted when the transfer is over (S_DATA is deasserted) and the OOPS counter is non-zero. This backs up the FIFO and decrements the OOPS counter.

A three data phase target burst read using a FIFO like the one shown in Figure 7-4 and an OOPS counter is shown in Figure 7-5. This waveform includes both PCI Bus signals and internal user application signals.

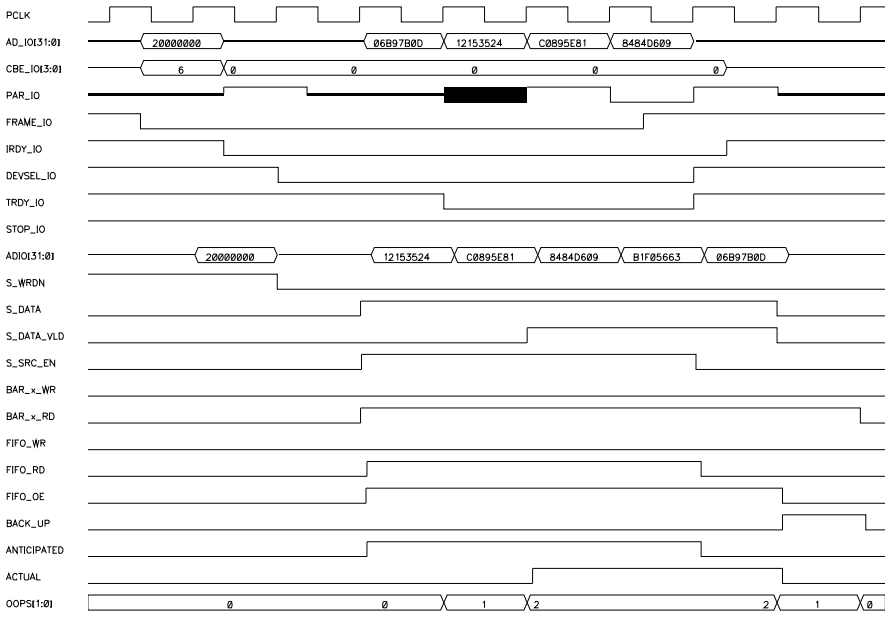


Figure 7-5: **Burst Read of Target FIFO**

Chapter 8

Target 64-bit Extension

This chapter provides additional details about the target 64-bit extension. Implementation of a user application that is a 64-bit target is almost identical to the implementation of a 32-bit version. The notable exception is that the data path is twice as wide. The target control signals behave the same regardless of the data path width.

Note that 64-bit transfers only apply to memory spaces. Other spaces do not support this capability. For a 64-bit implementation of the core interface to support 64-bit transfers as a target, at least one of the base address registers must be configured for memory space and be 64-bit enabled. Memory spaces which are 64-bit enabled support both 32-bit and 64-bit transfers. See Chapter 4, "Customizing the Interface" for more information.

Target Extension Signals

In addition to the target signals presented in Chapter 5, "Target Data Transfer and Control," the following additional signals are included in 64-bit implementations of the PCI interface.

- **ADIO[63:32]**. Bi-directional bus that provides the means for 64-bit data and address transfer to and from the core interface.
- **S_CBE[7:4].** Input that is a registered version of the CBE_IO lines, and is delayed by one cycle. It indicates the PCI command and byte enables during a target transaction. This signal is used primarily for byte enable information, as the command is presented on the lower half of the S_CBE bus.
- **S_CYCLE64**. Input that indicates to the user application that the PCI interface has claimed a 64-bit target transaction. It is asserted at the same time as **BASE_HIT** and remains asserted until the transaction is complete.
- **SLOT64**. Enables the 64-bit extension signals (AD_IO[63:32], CBE_IO[7:4], and PAR64_IO). This signal must remain static at all times except at power-on and immediately after a PCI Bus reset. The method for determining the appropriate value for this signal is design and device family dependent. See the *Initiator/Target v3.1 for PCI Getting Started Guide* for more information about SLOT64.

Handling 64-bit Transfers

The concepts presented in earlier chapters apply to 64-bit transfers. The methods for transaction decoding and data phase control are the same.

For burst transfers, S_CYCLE64 should be used to determine the correct "increment value" for target address pointers. Note that 64-bit enabled memory spaces will accept the following types of transactions:

- 64-bit transfers aligned on a QWORD boundary

- 32-bit transfers aligned on a DWORD boundary

The *PCI Local Bus Specification* forbids initiators from requesting unaligned 64-bit transfers. The user application should respond to such transfer attempts with target abort. The following logic will produce this result.

```
assign S_ABORT = S_CYCLE64 & ADDR[2];
```

Figure 8-1: **Unaligned 64-bit Target Read Transaction**

Using this logic, Figure 8-1 shows the core interface responding to an unaligned 64-bit read attempt with a target abort.

Figure 8-3 demonstrates an aligned 64-bit read attempt. Note that the behavior of the 64-bit extension signals mirrors that of the standard 32-bit signals. In Figure 8-1 the S_CYCLE64 signal is asserted.

Figure 8-2: **Aligned 64-bit Target Read Transaction**

For adequate support of 32-bit transfers, the user application must monitor S_CYCLE64 to behave appropriately. Typically, this will involve changes to the target address pointer, changes to any back up counters, and the addition of multiplexers on the data path.

To avoid the added complexity involved with elaborate address pointers and backup counters, an alternate solution is to only support burst transfers when the transaction request is 64-bit (that is, when S_CYCLE64 is asserted). When 32-bit transfers are requested, the user application can instruct the core interface to disconnect with data after a single data phase. Note that this does not eliminate the need for multiplexers on the data path.

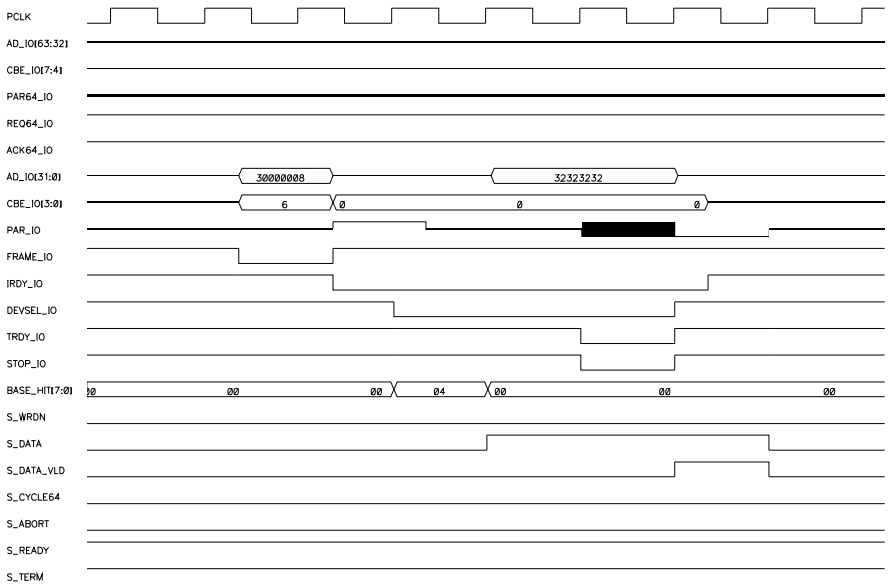


Figure 8-3: **Aligned 32-bit Target Read Transaction**

Figure 8-3 shows a 32-bit transfer attempt to a 64-bit enabled memory space. This may occur if 32-bit initiators reside in a 64-bit system, or if the 64-bit core interface is used in a 32-bit slot. In this case, S_CYCLE64 is not asserted and the core interface does not drive the 64-bit extension signals.

Chapter 9

Target Only Designs

This chapter provides information for disabling the initiator functions in the core interface to ensure correct implementation for target-only designs. While previous versions of the core interface included target/initiator and target-only versions, current versions are available only in the target/initiator configuration.

Logic Design Considerations

Conceptually, creating a target-only design is as simple as driving all initiator control signals to a benign (deasserted) state. In addition, the user application should be designed without making use of any initiator status or state outputs from the core interface.

However, connecting all unused control signals to logic high or logic low may have unwanted side-effects—the map program may optimize these signals away during the implementation step, which in itself is not problematic, but the guide files, which are required to guarantee timing in some designs, fail to work properly if certain signals are optimized away.

The solution to this problem is to drive the initiator control signals from the output of flip-flops. The following shows an example of this solution:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized
   if (RST) FAKE_LOGIC_0 = 1'b1;
   else FAKE_LOGIC_0 = 1'b0;
end
```

Table 9-1 lists the initiator control signals that must be tied off and the appropriate benign values. The table also identifies which signals must be driven from a flip-flop. Some signals are present only in 64-bit implementations of the interface.

Table 9-1: **Initiator Control Signals**

Signal Name	Benign Value	Flip-Flop
REQUEST	0	No
REQUESTHOLD	0	No
COMPLETE	1	Yes
M_WRDN	0	Yes

Table 9-1: **Initiator Control Signals *(Continued)***

Signal Name	Benign Value	Flip-Flop
M_READY	1	Yes
M_CBE[3:0]	0110	Optional
CFG_SELF	0	No
REQUEST64	0	No
M_CBE[7:4]	0000	Optional

System Level Considerations

During the design phase of a target-only design, a system-level issue to consider is that a target-only design does not need to drive REQ_O or monitor GNT_I on the PCI Bus.

The REQ_O and GNT_I pins can be connected to the PCI Bus (as they would be for initiator designs) or left unconnected, at the discretion of the designer. If GNT_I is not connected to the PCI Bus, it must be passively pulled up by an external resistor.

Chapter 10

Initiator Data Transfer and Control

This chapter describes some of the design issues that are associated with building the initiator portion of an application. It also provides basic information about controlling initiator transactions. Information about more elaborate initiator transactions is provided in Chapter 11, "Initiator Data Phase Control," Chapter 12, "Initiator Burst Transfers," and Chapter 13, "Initiator 64-bit Extension" of this guide.

Before you Begin

The following design guidelines are provided to assist you in determining the best method for building the initiator portion of an application. For best results, start by writing a statement about what it must accomplish. The following sections will help you determine design requirements.

Determine the Required Transfers

What data must be moved by the initiator?

- How big is each transfer and what is its alignment? These affect the transfer counter size and the address counter size.
- How fast can the intended target accept or send data?
- How fast can the user application provide or receive data?

Determine Termination Behavior

Invariably, a target signals some form of termination condition. How will the user application respond? What should it do?

- An initiator is obliged to retry an operation again if the target signals a target retry condition. Restart the operation from the beginning.
- An initiator should not retry an operation if it detects a target abort or a master abort condition. This indicates that the operation is either illegal for the selected target or that no target exists.
- Handling a disconnect condition is at the discretion of the user application. Initiators are not required to retry an operation terminated with a disconnect. An initiator can also be terminated if the internal latency timer expires while `GNT_I` is deasserted.

Determine Transaction Ordering Rules

When the user application requests the bus for an initiator operation, it may not be granted the bus for quite some time. In the meantime, another agent may initiate a target access to

the user application. How should the user application respond? Does the user application accept the target access? It may contain important information relevant to the pending initiator transaction.

Denying the other agent access by forcing a target retry is disastrous for a system with priority-based arbitration. The initiating agent may keep retrying the transaction because it has higher priority, and the user application will never access the bus, resulting in deadlock. However, in a round-robin system forcing a target retry is a good way for the user application to perform its pending transaction first. It can then respond to the target access when it is later retried by the other agent.

Assemble the Design

Transferring data as an initiator is a multi-step process. Some of the timing depends on other PCI agents, such as the arbiter and the selected target. The fundamentals of an initiator design, including a simple state machine, are provided in this chapter and are sufficient for implementing non-burst transfers to other PCI agents.

For more elaborate designs, Chapter 11, "Initiator Data Phase Control" presents information about inserting initiator wait states and controlling the length of initiator transfers. This information is useful for both non-burst and burst designs.

For burst designs, see Chapter 12, "Initiator Burst Transfers" for a demonstration of how to support initiator burst transfers through modifications to the simple state machine and the inclusion of additional logic.

Chapter 13, "Initiator 64-bit Extension" discusses the use of the 64-bit extension for increased performance.

Typical Initiator Data Interface

In applications that do not use initiator burst transactions, data is generally transferred to and from registers in the user application. These registers are connected to control signals required for initiator data transfer and to any additional control and data path logic provided by the user. These registers may also connect to internal FIFOs or to I/O pins on the user application.

Figure 10-1 shows a typical interfaced initiator data register.

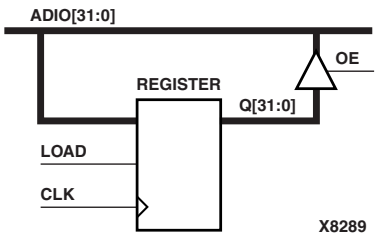


Figure 10-1: **Example Initiator Register**

The purpose of this chapter is to demonstrate the logic required in the user application to complete simple initiator transfers and thereby generate the load and output enable signals for the data register.

Initiator Interface Signals

The following signals control initiator data transfer to and from the core interface. For basic transfers, only a subset of these signals need be used. More elaborate designs involving initiator wait state insertion or initiator burst are covered in later chapters. Note that references to inputs and outputs are made with respect to the user application.

- ADIO[31:0]. Bidirectional bus provides the means for data and address transfer to and from the core interface.
- M_DATA_VLD. Input has two interpretations depending on the direction of data transfer. When the user application is sinking data (initiator reads), M_DATA_VLD indicates that the user application should capture valid data from the ADIO bus. When the user application is sourcing data (initiator writes), M_DATA_VLD indicates that a data phase has completed on the PCI Bus.
- M_SRC_EN. Input only used during initiator burst writes. It indicates to the user application that the data source which drives output data onto the ADIO bus must provide the next piece of data. In most applications, this signals the user application to advance the data pointer for the source that is providing data.
- TIME_OUT. Input indicates to the user application that the latency timer has expired. This means that the user application has exceeded the maximum number of clock cycles allowed by the system configuration software. It is only of importance to designs that perform initiator bursts.
- CSR[39:0]. Input provides status information about the current transfer. This is used primarily in initiator burst applications with non-prefetchable sources to determine if any associated address pointers must be backed up.
- REQUEST. Output from the user application instructs the core interface to request the PCI Bus and to begin an initiator transaction once GNT_I is asserted.
- REQUESTHOLD. Output from the user application indicates that it desires to continue requesting the PCI Bus for an extended period of time.
- M_WRDN. Output indicates the direction of data transfer for the current initiator transaction. Logic high indicates that the user application is sourcing data (i.e. initiator write). The user application should not change this output during a transaction.
- M_CBE[3:0]. Output indicates the PCI command and byte enables during an initiator transaction. The command should be presented during the assertion of M_ADDR_N by the core interface, and the byte enables should be presented during each data phase.
- M_READY. Output from the user application indicates that it is ready to transfer data, and can be used to insert wait states during the first data phase of a transaction.
- COMPLETE. Output from the user application informs the initiator state machine that it should complete the current transaction.

The following signals are output by the initiator state machine in the core interface. These states are defined in Appendix B of the *PCI Local Bus Specification*.

- I_IDLE. Input indicates that the initiator state machine is in the idle state and that it is not actively driving the PCI Bus.
- DR_BUS. Input indicates that the initiator state machine is driving the PCI Bus because the arbiter has parked the bus on the core interface (GNT_I asserted with no pending or active request for bus grant from the user application).

- M_ADDR_N. Input indicates that the initiator state machine expects the user application to drive ADIO with the PCI Bus address for a requested transaction. This is *not* a confirmation that a bus transaction will begin.
- M_DATA. Input indicates that the initiator state machine is in the data transfer state.

Note: The core interface uses a single cycle of address stepping during address phases on the PCI bus, which is fully compliant with the *PCI Local Bus Specification*. However, it is possible for the core interface to lose bus grant during this clock cycle, forcing the core interface to relinquish the bus. Under this condition, the core interface will have asserted M_ADDR_N, but will not transition to M_DATA. The core interface will automatically re-request the bus without user intervention. In some extreme cases, this sequence of events can occur more than once, which will appear as a train of M_ADDR_N assertions, followed by a final M_DATA assertion. For this reason, the user application design should not assume that M_ADDR_N is asserted once per transaction, nor should the user application assume that an assertion of M_ADDR_N indicates a subsequent assertion of M_DATA in the next clock cycle.

Initiator Control

Figure 10-2 shows a complete initiator transaction consisting of several steps. The proper sequencing of the core interface control signals is best performed by a state machine in the user application. In addition to a state machine and a data register, additional logic is required to drive outputs to the core interface and to generate inputs used by the state machine.

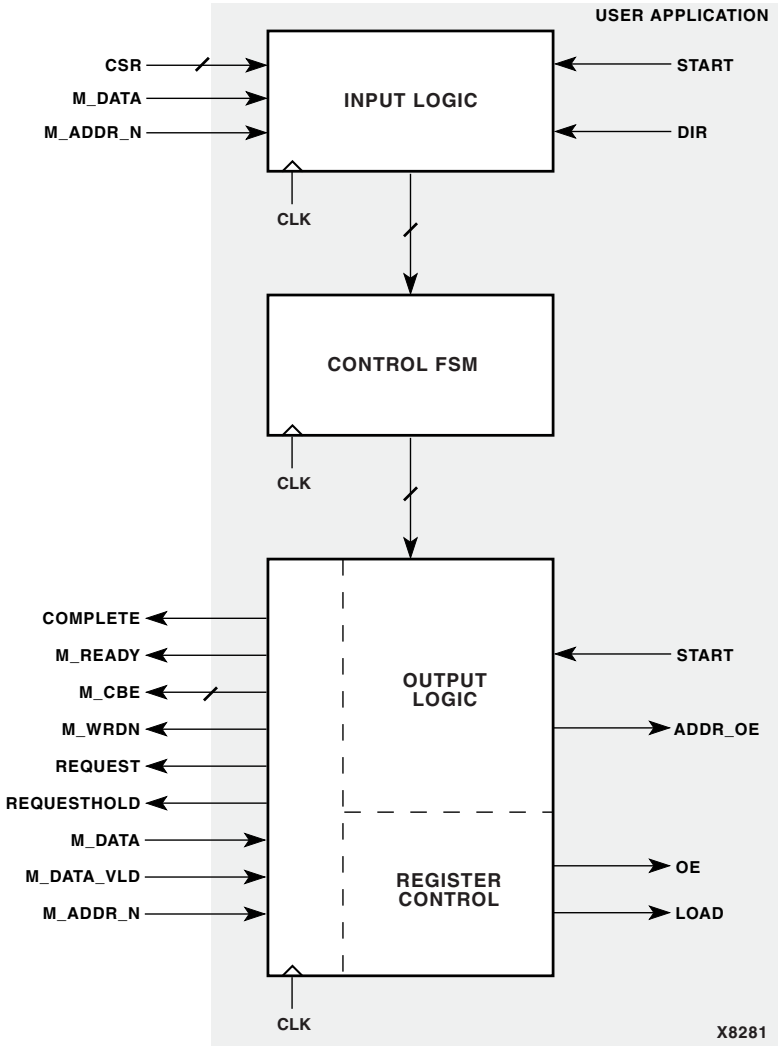


Figure 10-2: **Initiator Control Block Diagram**

The following presents a sample design that is suitable for performing single data phase initiator transfers in the form of memory reads and memory writes. The extension of this example to add support for burst transfers is discussed later in this guide. The design breaks down into the following distinct sections:

- Inputs to the state machine
- The state machine itself
- Outputs to the core interface
- Data register control signals

Inputs to the State Machine

The sample state machine requires one input to tell it to start, another input to indicate the direction of the desired transfer, and five additional inputs to effectively monitor the progression of the transaction.

START indicates to the state machine that it should begin a transaction. This signal is generated by the user application. The source of this signal will vary depending on the specific user application function.

DIR indicates the direction of the desired transfer, and is generated by the user application. The source of this signal will vary depending on the specific user application function. The DIR signal should not change from the time START is asserted until the end of the initiator transfer. In this particular example, initiator writes are selected by driving M_WRDN to logic high.

M_ADDR_N and M_DATA are outputs of the user interface that are used by the state machine. The signal M_DATA_FELL indicates that a falling edge has been detected on M_DATA. This is generated by the following code.

```
always @(posedge CLK or posedge RST)
begin : edge_detect
   if (RST) M_DATAQ = 1'b0;
   else M_DATAQ = M_DATA;
end

assign M_DATA_FELL = !M_DATA & M_DATAQ;
```

FATAL and RETRY are derived from PCI interface signals. These signals indicate how a transaction attempt has terminated. FATAL indicates that a master abort or target abort has occurred. RETRY indicates that the target issued a disconnect without data. As the initiator state machine only performs single transfers, this implies a retry by the target.

This information is obtained from the extended status signals, CSR[39:32]. The extended status information is valid one clock cycle after a particular event has occurred on the PCI Bus. Unless the status is used during that cycle, it must be registered to preserve it for later use. The following code performs the task of preserving the status information from the final data phase.

```
always @(posedge CLK or posedge RST)
begin : watch_status
   if (RST)
   begin
      FATAL = 1'b0;
      RETRY = 1'b0;
   end
   else if (!M_ADDR_N)     // clear at beginning
   begin
      FATAL = 1'b0;
      RETRY = 1'b0;
   end
   else if (M_DATA)        // latch until end
   begin
      FATAL = CSR[39] | CSR[38];
      RETRY = CSR[36];
   end
end
```

The State Machine

Figure 10-3 illustrates the sample state machine. This state machine is suitable for performing single data phase initiator transfers.

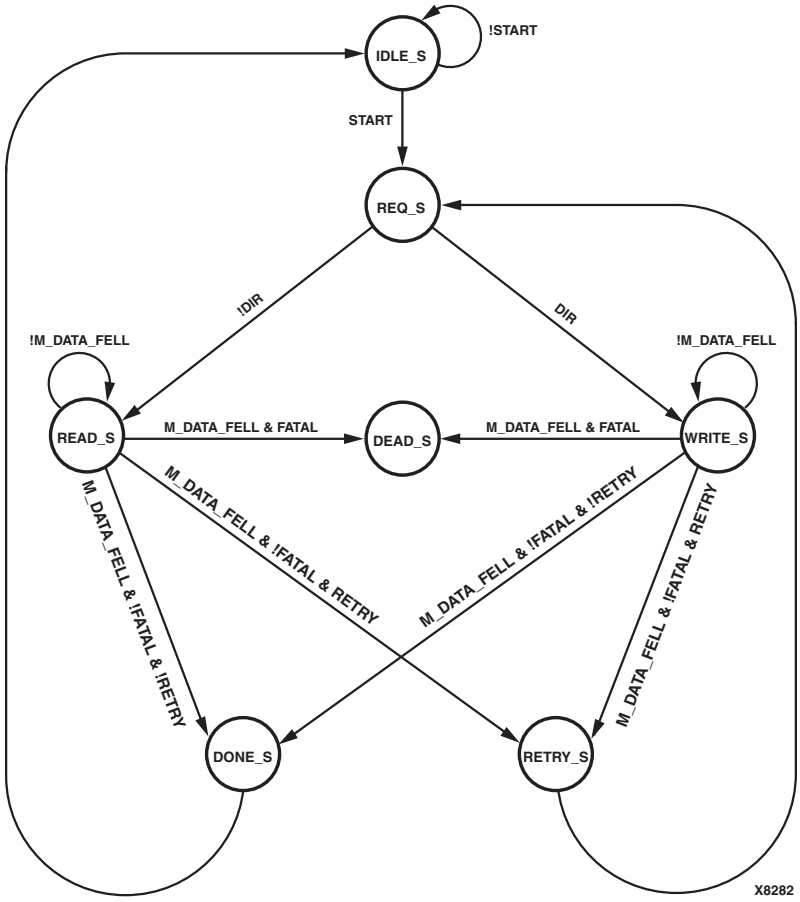


Figure 10-3: **Sample User Application State Machine**

Each state performs a specific step in the sequence. The individual states and their purposes are listed below.

IDLE_S—idle state. During this state, no initiator activity takes place, and the state machine waits for the user application to assert START. Once START has been asserted, the state machine proceeds to the REQ_S state.

REQ_S—request state. During this state, a request is made to begin an initiator transaction. The state machine then branches based on the value of the DIR signal. If DIR indicates an initiator read, the next state is READ_S. Otherwise, the next state is WRITE_S.

READ_S and **WRITE_S**—data transfer states. The state machine stays in READ_S or WRITE_S until it detects that the transaction is over. The transaction is over when M_DATA_FELL is asserted.

These states are identical except that the state machine outputs differ. The output logic is discussed in the next section. Since DIR is available in the user application, these two states may be reduced into a single XFER_S state if DIR is used in the data register control equations. The states are separate in this example for clarity. In more elaborate designs, it is often desirable to completely split the state machine into separate read and write state

machines. For very simple designs, it is possible to further reduce the example state machine presented here.

Whether in READ_S or WRITE_S, the state machine makes a three way branch after the transfer state, based on the FATAL, and RETRY signals. If a fatal error has occurred, the state machine moves to the DEAD_S state. If no fatal error occurred and no data transfer occurred, the state machine moves to the RETRY_S state. Otherwise, the state machine moves to the DONE_S state.

DEAD_S is the terminal state for handling fatal errors. In practice, the user application should provide some method for resetting the state machine to handle such errors.

RETRY_S is the retry state. In this simple example, no actions are performed in the RETRY_S state, and the state machine immediately transitions to the REQ_S state and attempts the transaction again. More elaborate designs may implement a retry counter to discard transactions after a certain number of retries, as permitted in the *PCI Local Bus Specification*.

DONE_S is the done state. As in the RETRY_S state, no specific actions are performed in this state, and the state machine immediately transitions back to the IDLE_S state.

The state machine shown in Figure 10-3 could be implemented as shown:

```
always @(posedge CLK or posedge RST)
begin : initiator_fsm
   if (RST) STATE = IDLE_S;
   else case (STATE)
      IDLE_S :  begin
                if (START) STATE = REQ_S;
                else STATE = IDLE_S;
                end
      REQ_S :   begin
                if (DIR) STATE = WRITE_S;
                else STATE = READ_S;
                end
      WRITE_S : begin
                if (M_DATA_FELL)
                begin
                   if (FATAL) STATE = DEAD_S;
                   else if (RETRY) STATE = RETRY_S;
                   else STATE = DONE_S;
                end
                else STATE = WRITE_S;
                end
      READ_S :  begin
                if (M_DATA_FELL)
                begin
                   if (FATAL) STATE = DEAD_S;
                   else if (RETRY) STATE = RETRY_S;
                   else STATE = DONE_S;
```

```
                    end
                    else STATE = READ_S;
                    end
                RETRY_S : STATE = REQ_S;
                DONE_S :  STATE = IDLE_S;
                DEAD_S :  STATE = DEAD_S;
                default : STATE = IDLE_S;
            endcase
        end
```

Outputs to the Core Interface

The state machine in the user application is responsible for driving several signals which are output to the core interface. These signals are discussed below along with the logic required to generate them. This discussion assumes the existence of a 32-bit ADDRESS signal provided by other logic in the user application. This ADDRESS represents the PCI Bus address of the desired target. The source of this signal varies, depending on the specific user application function.

Before requesting a transaction, the user application should indicate that it is ready to transfer data. Although it is possible to initiate a transaction and then insert wait states using the M_READY signal, bus bandwidth is conserved by requesting a transaction only after the user application is ready to transfer data. In this example, the user application is always ready.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
    if (RST) M_READY = 1'b0;
    else M_READY = 1'b1;
end
```

The M_READY signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or logic zero.

The state machine initially requests access to the PCI Bus by asserting REQUEST. This is done in the REQ_S state and results in the assertion of REQ_O. The REQUEST signal must only be asserted for a single cycle to initialize a request. Note that the bus master enable bit in the command register must be set before the core interface is able to request the bus. This is the responsibility of the host bridge and system configuration software.

```
assign REQUEST = (STATE == REQ_S);
```

The REQUESTHOLD signal is not used in this example, and is simply assigned to logic zero.

```
assign REQUESTHOLD = 1'b0;
```

When the core interface has received a bus grant, the user application must drive the target address on the ADIO bus. This address is presented on the PCI Bus during the address phase.

```
assign ADDR_OE = M_ADDR_N;
assign ADIO = ADDR_OE ? 32'bz : ADDRESS;
```

Simultaneously, the user application must also drive M_CBE to indicate the desired PCI Bus command. At other times, the M_CBE signal is used to indicate byte enables. In this example, the initiator performs memory read and memory write transactions (commands 0x6 and 0x7, respectively), depending on the value of the DIR signal. All bytes are enabled. The bus command and byte enables are easily modified to accommodate the requirements of the user application.

```
assign COMMAND = {3'b011, DIR};
assign BYTE_ENABLE = 4'b0000;
assign M_CBE = M_ADDR_N ? BYTE_ENABLE : COMMAND;
```

Throughout the entire transaction, the user application must drive M_WRDN to indicate the desired transfer direction. This signal must remain constant during a transaction. By assigning it to track the DIR signal, which was stipulated to be constant during a transaction, this requirement is met.

```
assign M_WRDN = DIR;
```

The M_WRDN signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

The final signal, COMPLETE, indicates to the core interface that it should finish the current transaction. In the case of single transfers, COMPLETE must be asserted no later than one cycle after REQUEST, and deasserted when the transfer is complete.

```
always @(posedge CLK or posedge RST)
begin : driving_complete
    if (RST) COMPLETE = 1'b0;
    else case (STATE)
        REQ_S :   COMPLETE = 1'b1;
        READ_S :  COMPLETE = 1'b1;
        WRITE_S : COMPLETE = 1'b1;
        default : COMPLETE = 1'b0;
    endcase
end
```

The COMPLETE signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, if possible. In practice, it is usually driven by combinational logic.

Note: Never tie this signal to logic one or to logic zero.

Data Register Control Signals

As a final consideration, the logic for the load and output enables of the typical initiator data register shown in Figure 10-1 are as follows:

```
assign LOAD = (STATE == READ_S) & M_DATA_VLD;
assign OE = (STATE == WRITE_S) & M_DATA;
```

As is the case in target read transactions, the user application should always drive the entire width of the ADIO bus during initiator writes, even if some byte enable signals are not asserted. For initiator reads, the LOAD signal may be further qualified by the byte enable signals to generate separate load signals for each byte.

Sample Transactions

Figure 10-4 through Figure 10-8 illustrate typical PCI Bus read and write transactions that result from implementing the logic presented in this chapter, as well as several exceptional cases that can arise.

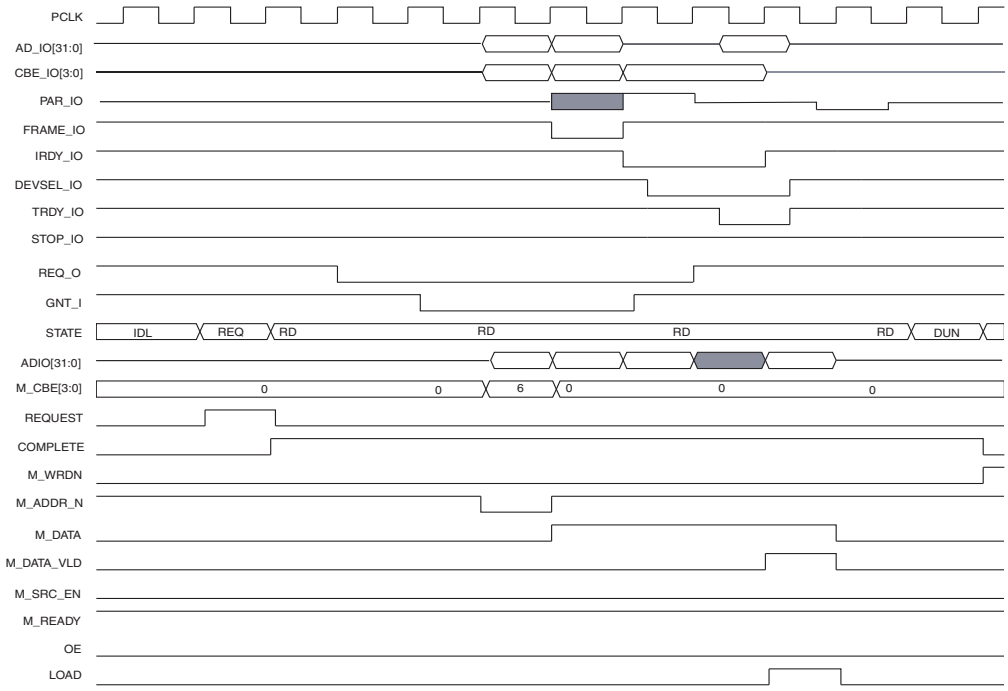


Figure 10-4: **Initiator Read Transaction**

Figure 10-4 demonstrates the initiator issuing a read command to a target. The transaction is terminated in a normal fashion by the initiator. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during an initiator read transaction.

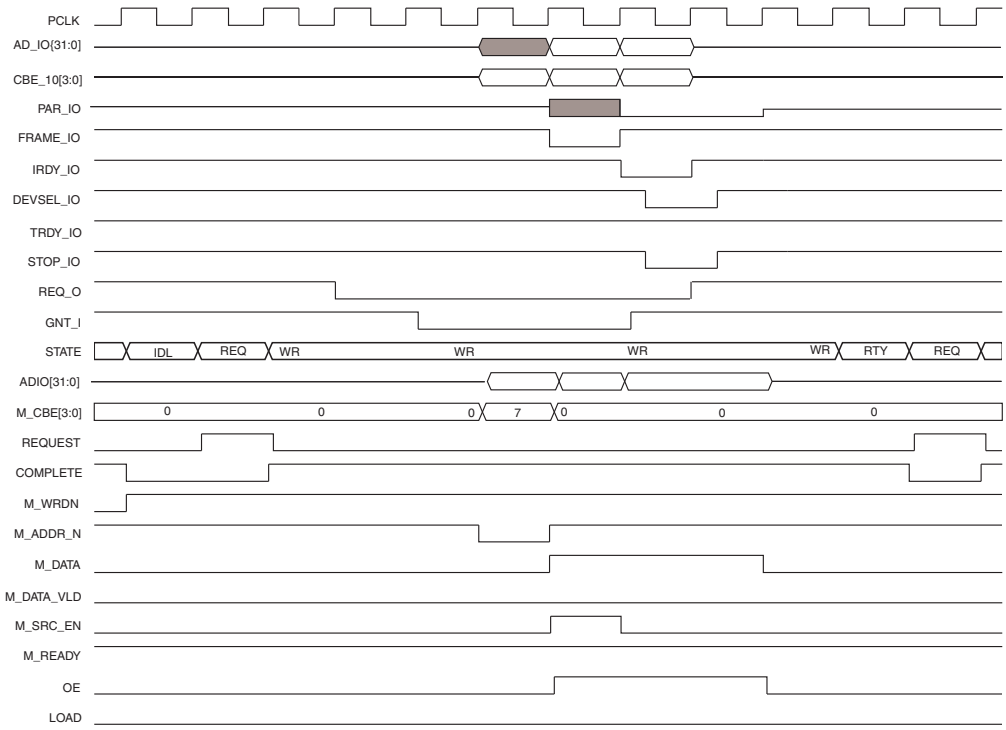


Figure 10-5: **Initiator Write Retried by Target**

Figure 10-5 shows the initiator issuing a write command to a target. The target terminates the transaction with retry. When this occurs, no data is transferred, and the initiator will try again.

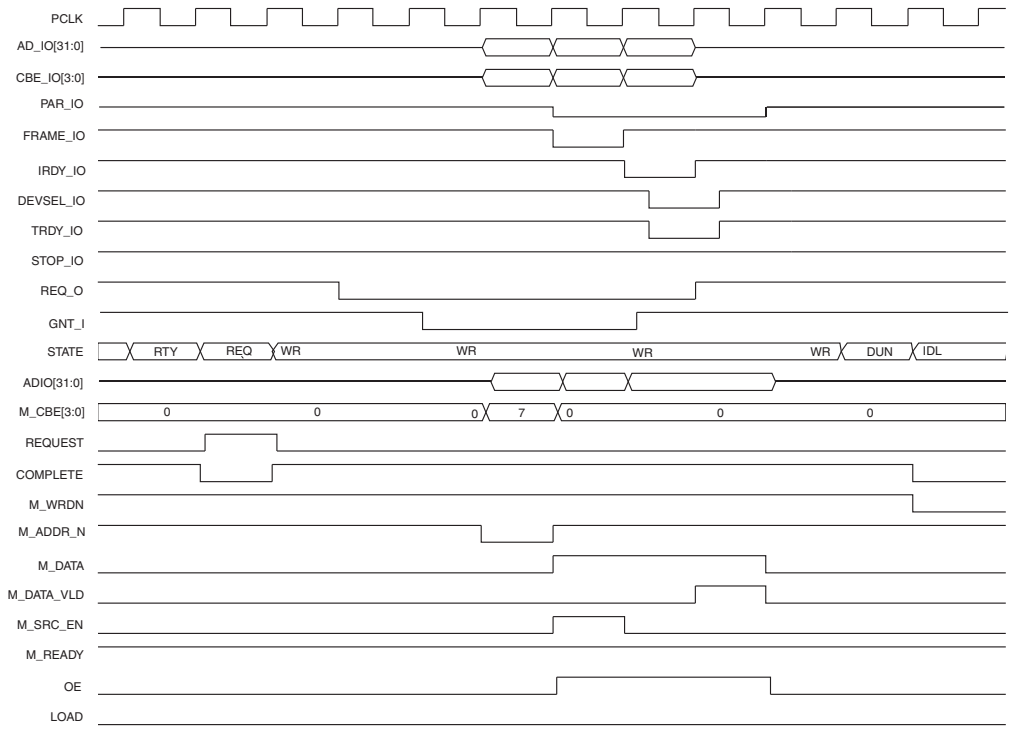


Figure 10-6: **Initiator Write Retried and Completed**

Figure 10-6 shows the initiator re-issuing the original command. This time the target does not retry the transaction. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during an initiator write transaction.

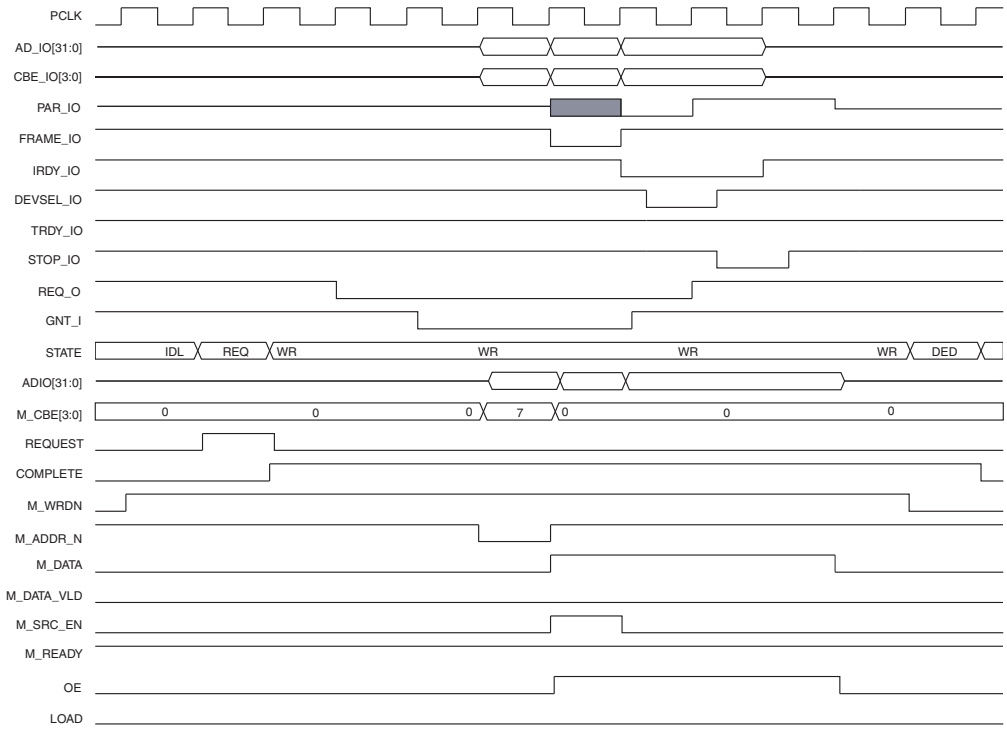


Figure 10-7: **Target Abort**

Figure 10-7 demonstrates the behavior of the core interface when a target signals an abort (target abort). This event often occurs due to incorrect programming or serious system errors. It can also signify that the initiator has incorrectly attempted to burst data beyond the address space of the target. The user application must respond appropriately.

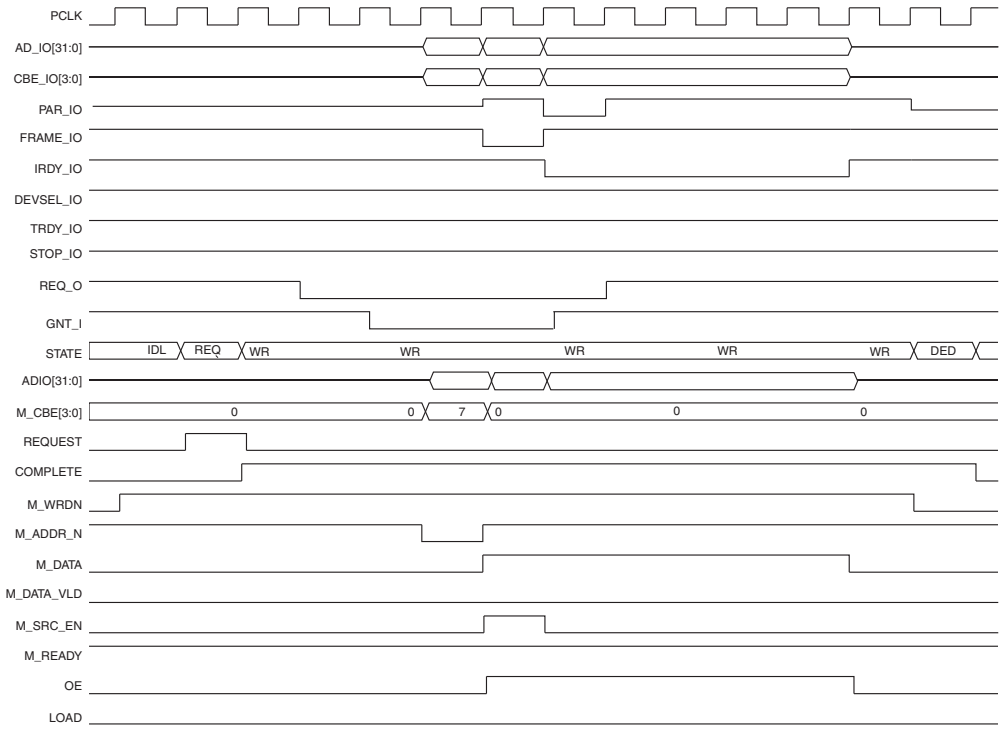


Figure 10-8: **Master Abort**

Figure 10-8 demonstrates the behavior of the core interface when no target responds (master abort). This event is expected during some configuration transactions and special cycle broadcast cycles. However, during normal operation, this would occur due to incorrect programming or serious system errors. It is critical that the user application respond appropriately.

XILINX®

Chapter 11

Initiator Data Phase Control

This chapter describes how the user application can control aspects of initiator transactions to accommodate its own ability to source or sink data. The user application can insert wait states before the first data transfer to allow itself additional time if it is not ready. Additionally, the user application can control the number of data phases by indicating when to complete the transaction.

The generation of initial latency with wait states, while possible, is not recommended. This technique wastes valuable bus bandwidth. From a bandwidth perspective, it is better for the user application to delay making a bus request until it is ready to perform a transaction.

Control Modes

Data phase control is achieved using the **M_READY** and **COMPLETE** signals. Combinations of the two control signals include the following modes:

- **Wait Burst**. Inserts wait states at the beginning of a PCI Bus transaction (holds off the first data phase) by delaying the assertion of **IRDY_IO** by the PCI interface. This particular wait mode is for use with initiator burst transactions. Use of this mode indicates to the PCI interface that the user application is not ready and will attempt more than one data phase.
- **Wait Single**. Inserts wait states at the beginning of a PCI Bus transaction in the same manner as the wait burst mode. However, this mode is for use with single initiator transactions. Use of this mode indicates to the PCI interface that the user application is not ready and will not attempt more than one data phase.
- **Proceed**. Allows PCI Bus data phase(s) to proceed without interruption. While the selected target may insert wait states or terminate the transaction prematurely, the user application must be prepared to transfer data at full speed. This is for use with multiple data phase transfers only.
- **Finish**. Causes the PCI interface to complete the transaction as soon as possible. This is for use with both single and multiple data phase transfers.

Again, the precise disconnect sequence is affected by whether or not the selected target terminates the transaction. The PCI interface automatically generates the correct behavior.

Table 11-1 shows the four modes of operation and the corresponding **M_READY** and **COMPLETE** values.

Table 11-1: **Data Phase Control Signals for Initiators**

Condition	Bus Signals	From User Application	
		M_READY	COMPLETE
Wait Burst	**IRDY_IO** = 1 **FRAME_IO** = 0	Low	Low
Wait Single	**IRDY_IO** = 1 **FRAME_IO** = 1	Low	High
Proceed	**IRDY_IO** = 0 **FRAME_IO** = 0	High	Low
Finish	**IRDY_IO** = 0 **FRAME_IO** = 1	High	High

Changing from one mode to another must not be done in arbitrary sequence. In addition, the timing of mode transitions is critical to ensure precise control over the number of data phases that occur in a transaction. This is especially important when switching to the Finish mode.

The permitted data phase control sequences for initiator designs using the PCI interface are displayed in Figure 11-1. The exact timing details are defined in subsequent sections of this chapter.

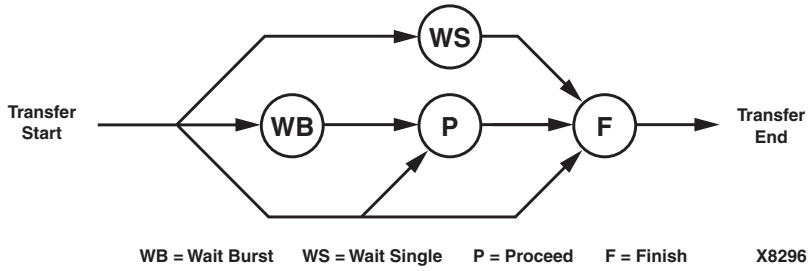


Figure 11-1: **Permitted Data Phase Control Sequences**

The control sequences shown in Figure 11-1 assume that the user application is terminating the transaction. In practice, a transaction may end for reasons the user application cannot control, such as a target termination or a timeout. When the initiator state machine becomes inactive after such a condition, the sequencing rules no longer apply.

Note that the wait modes cannot be used to insert wait states during arbitrary data phases in a transaction. They may only be used to delay the completion of the first data phase of an initiator transaction. This is called master data latency in the *PCI Local Bus Specification*.

Figure 11-1 shows four possible sequences for controlling the number of data phases in a transaction.

- Single transfers with no master data latency
- Single transfers with master data latency
- Burst transfers with no master data latency
- Burst transfers with master data latency

All initiators are required to complete the first data phase of a transaction within 8 clocks from the assertion of **FRAME_IO**. The *PCI Local Bus Specification* strongly discourages the use of master data latency and states that there is generally no reason for using it. The user application is responsible for observing this requirement.

Note: During initiator state machine activity, do not violate the mode sequencing described above. When the initiator state machine is inactive, this requirement does not apply.

Control Pipeline

To meet the stringent PCI Bus performance requirements, the PCI interface pipelines all the bus control signals and the data path. Consequently, the **M_READY** and **COMPLETE** signals must be presented in advance of the desired effect.

The signals **M_READY** and **COMPLETE** connect to the initiator state machine through multiple logic levels. For this reason, it is highly recommended that the logic driving these signals be kept as simple as possible. Although it is advantageous to drive these signals from flip-flops in the user application, this is typically not possible in all but the most simple (non-burst) designs.

The user application must present the correct initial data phase control mode no later than one cycle after asserting **REQUEST**. After this, the user application may change modes as long as it does not violate the sequencing shown in Figure 11-1.

In all cases, once the user application decides to finish an initiator transaction, it must select the finish mode by setting **M_READY** and **COMPLETE** appropriately. After the user application signals that it wants to finish a transaction, it must hold **M_READY** and **COMPLETE** until the end of the transaction. The deassertion of **M_DATA** by the PCI interface indicates that the transfer is over.

Transaction Termination Rules

If the user application is sending or receiving a single data word, the user application must signal the wait single or finish mode no later than one cycle after asserting **REQUEST**. If the user application inserts wait states using the wait single mode, it must switch to the finish mode before it causes the PCI interface to violate the master data latency specification. Again, once the user application signals to finish the transaction, it must hold **M_READY** and **COMPLETE** through the end of the **M_DATA** state.

If the user application is sending or receiving two data words, the user application may start in either the wait burst or proceed modes. If starting in the wait burst mode, the user application must switch to the proceed mode before it causes the PCI interface to violate the master data latency specification. Once in the proceed mode, the user application must switch to the finish mode when both of the following conditions have been met:

- The proceed mode has been signalled for at least one cycle
- The signal **M_DATA** has been asserted for at least one cycle

For burst transfers of three or more data words, the initial mode selection is the same as in the two-transfer case. The user application should switch to the finish mode when three transfers remain and **M_DATA_VLD** is asserted.

Implementation

Although the rules presented above appear complicated when presented textually, the following example demonstrates the logic required to drive **M_READY** and **COMPLETE** in a general case. This example builds on the example presented in an earlier chapter, enabling it to perform simple burst transfers. This code would replace the previous logic that generated **M_READY** and **COMPLETE**.

Most initiator designs use a transfer counter to track the desired burst length. The following logic implements a transfer counter and generates three outputs, which indicate the number of remaining transfers. The **BURST_LENGTH** and **START** signals are generated elsewhere in the user application. Note that for reads and writes, **M_DATA_VLD** is used to indicate a successful data transfer on the PCI Bus.

```
always @(posedge CLK or posedge RST)
begin : transfer_counter
   if (RST) XFER_CNT = 4'h0;
   else if (START) XFER_CNT = BURST_LENGTH;
   else if (M_DATA_VLD) XFER_CNT = XFER_CNT - 4'h1;
end

assign CNT3 = (XFER_CNT == 4'h3);
assign CNT2 = (XFER_CNT == 4'h2);
assign CNT1 = (XFER_CNT == 4'h1);
```

The following logic generates an initiator *ready* signal that is used later in the **M_READY** and **COMPLETE** logic. In user application designs that do not insert wait states as an initiator, this logic can be optimized away. The logic samples a signal called **READY_FLAG**, which is produced elsewhere in the user application. This signal indicates that the user application is ready to transfer data.

```
always @(posedge CLK or posedge RST)
begin : not_recommended
   if (RST)
   begin
      INIT_READY = 1'b1;
      INIT_WAITED = 1'b0;
   end
   else
   begin
      INIT_WAITED = !INIT_READY;
      if (START) INIT_READY = READY_FLAG;
      else if (!INIT_READY) INIT_READY = READY_FLAG;
   end
end
```

The final step is to generate **M_READY** and **COMPLETE**. Typically, initiator wait states are not used and the **INIT_WAITED** signal should be optimized out of the equations. It is critical to reduce the logic as much as possible for timing reasons.

```
assign FIN1 = CNT1 & REQUEST;
assign FIN2 = CNT2 & M_DATAQ & !INIT_WAITED;
assign FIN3 = CNT3 & M_DATA_VLD;

assign ASSERT_COMPLETE = FIN1 | FIN2 | FIN3;
assign COMPLETE = ASSERT_COMPLETE | HOLD_COMPLETE;
assign M_READY = INIT_READY;

always @(posedge CLK or posedge RST)
begin : finish_up
   if (RST) HOLD_COMPLETE = 1'b0;
   else if (M_DATA_FELL) HOLD_COMPLETE = 1'b0;
   else if (ASSERT_COMPLETE) HOLD_COMPLETE = 1'b1;
end
```

Note: **M_READY** and **COMPLETE** must not be assigned static values.

Sample Transactions

The following Figures 11-2 through 11-5 illustrate typical initiator read and write burst transactions resulting from implementing the logic presented in this chapter. Sample transactions for cases of single transfers are presented in earlier chapters.

Figure 11-2 demonstrates the PCI interface performing a burst transfer and also shows the timing relationship between selected user application signals and PCI Bus signals during the transaction.

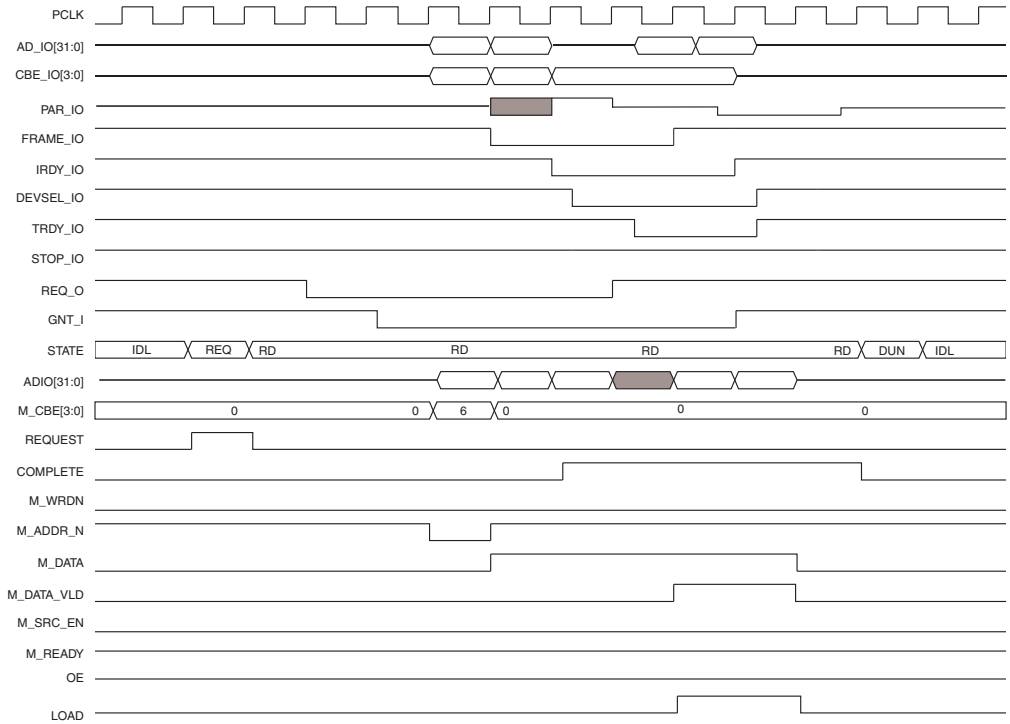


Figure 11-2: **Two DWORD Initiator Read Transfer**

Figure 11-3 demonstrates the PCI interface performing a burst transfer and also shows the timing relationship between selected user application signals and PCI Bus signals during the transaction.

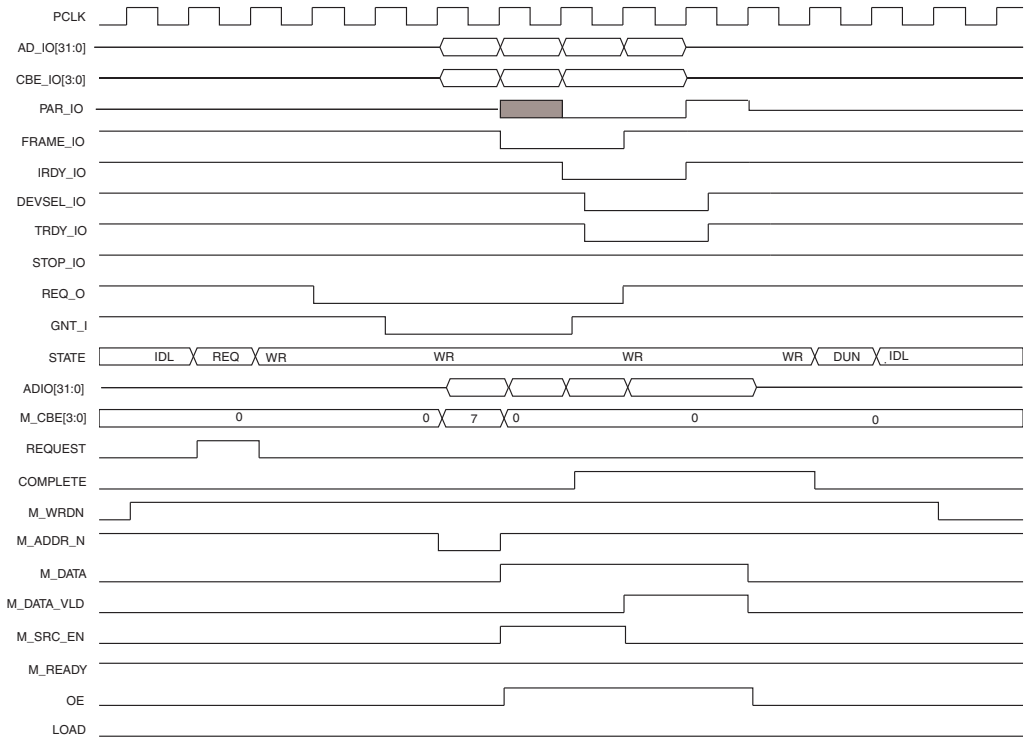


Figure 11-3: **Two DWORD Initiator Write Transfer**

.Figure 11-4 demonstrates the PCI interface performing a burst transfer.

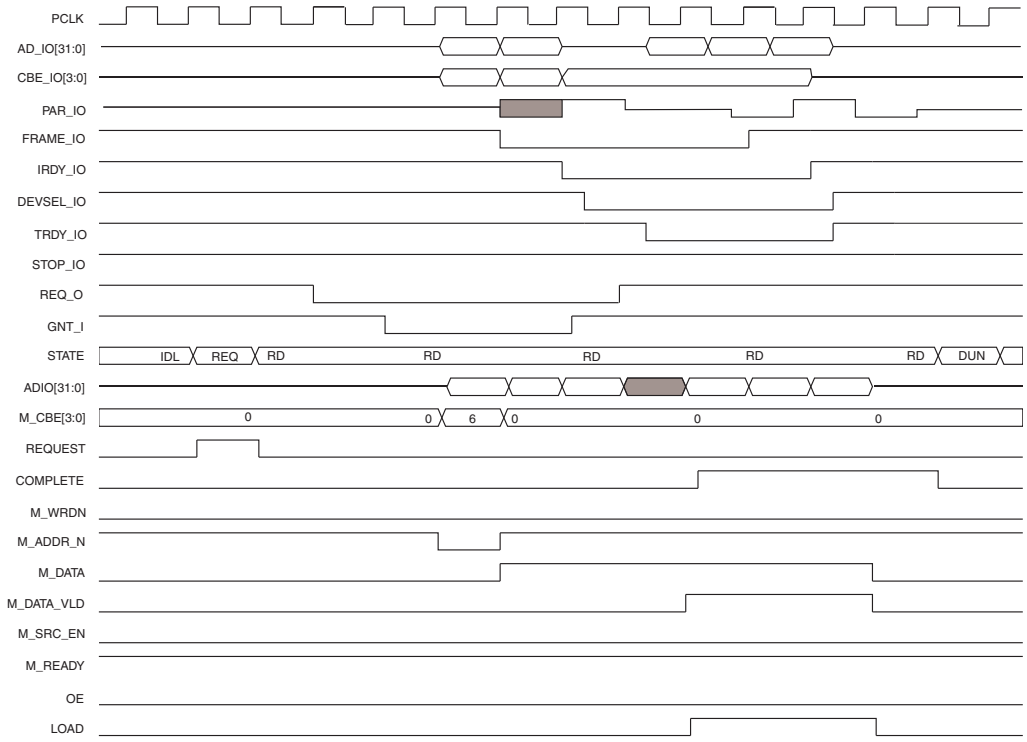


Figure 11-4: **Three DWORD Initiator Read Transfer**

Figure 11-5, demonstrates the PCI interface performing a burst transfer. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during the transaction.

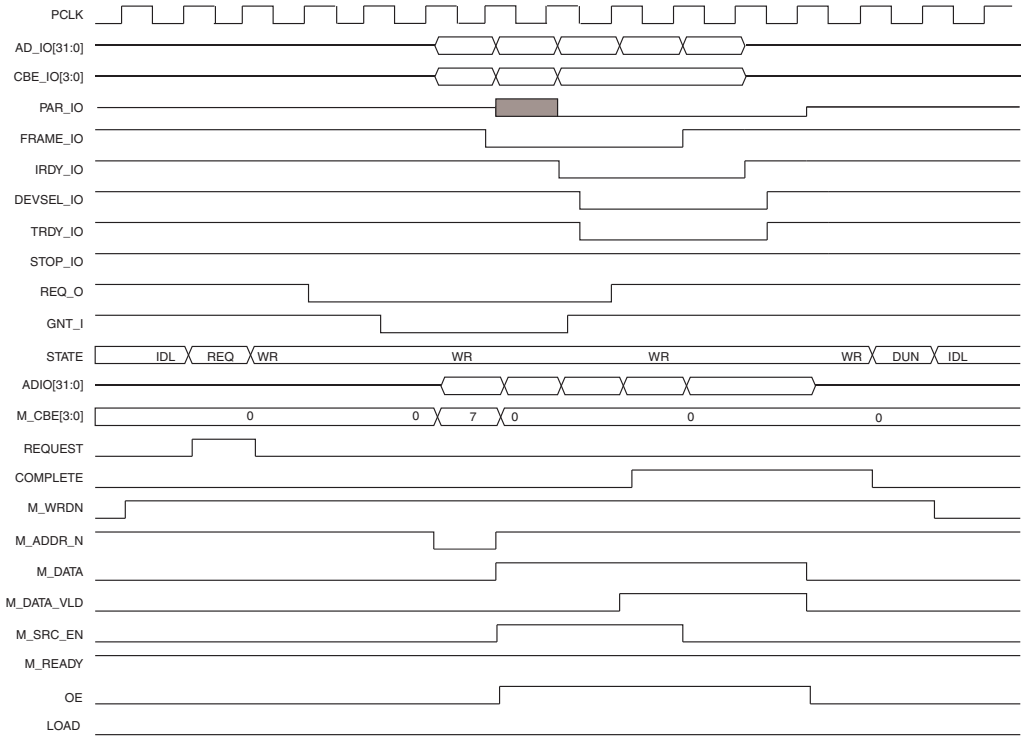


Figure 11-5: **Three DWORD Initiator Write Transfer**

Chapter 12

Initiator Burst Transfers

As is the case in target transactions, single transfers as an initiator waste valuable bus bandwidth. The performance advantage in PCI is derived from burst transactions, where two or more data words are transferred during the transaction.

Building a user application that supports single initiator transfers is moderately complex, while building an application that supports initiator burst transfers is even more so. However, if maximum bandwidth is the goal, building an application that supports initiator burst transfers is worth the effort.

Keeping Track of the Address Pointer

In a PCI transaction, only the starting address is broadcast over the bus. For single transfers as an initiator, a register that holds the target address is sufficient.

For burst transfers, however, the user application must keep track of the current address because the target can terminate the transaction at any time. If the initiator is to continue the transfer during another transaction, it must resume at the appropriate address.

The initiator must always provide and track a full 32 bit address (the lowest two bits are always zero). In some applications, however, a smaller address counter and a larger data register are sufficient to track the current address. The actual size of the counter depends on the alignment and length of the transfers required by the user application. In the worst case, the initiator may require a full 30-bit, loadable binary counter. See Table 12-1.

Table 12-1: **Example Initiator Address Pointer**

31	2	1	0
30-bit loadable binary counter		0	0

Two methods may be used for incrementing the initiator address pointer. The first method is to use `M_DATA_VLD` as a increment enable signal. This signal indicates successful data transfer for both initiator reads and initiator writes.

This method is simple and ensures that the initiator address pointer is always valid. It is most useful in user application designs that do not require explicit addresses during a transfer, for example, bursting data to or from FIFOs. An example of this is presented later in this chapter.

Another method is to use `M_DATA_VLD` as an incrementenable during initiator reads, and use `M_SRC_EN` during initiator writes. This method allows the initiator address pointer to serve as a local pointer used to index storage elements in the user application.

This is particularly useful in designs with addressable RAM or in other cases where explicit addresses are required. However, this method requires that the initiator address pointer be "backed up" in cases where the target terminates the transaction prematurely or the transfer spans multiple bus transactions. The relationship between M_DATA_VLD and M_SRC_EN is identical to that of S_DATA_VLD and S_SRC_EN.

Sinking Data in Burst Transfers

During initiator reads, the core interface transfers burst data using a pipelined data path. The data valid signal, M_DATA_VLD, is used to advance the initiator address pointer (and any other data pointers in the user application logic). At the same time the initiator address pointer is advanced, the user application also captures valid data from the internal ADIO bus.

Using M_DATA_VLD to capture burst data is very similar to the simple case of single transfers. The user application may enable different data sinks, if necessary, based upon the current initiator address pointer. The generation of a local address pointer was discussed in the previous section.

An initiator burst read is shown in the waveform of Figure 12-1. This waveform includes both PCI Bus signals and internal user application signals. In this figure, the initiator address pointer is implemented to provide a local pointer to index storage elements. During initiator reads, this pointer is incremented when M_DATA_VLD is asserted.

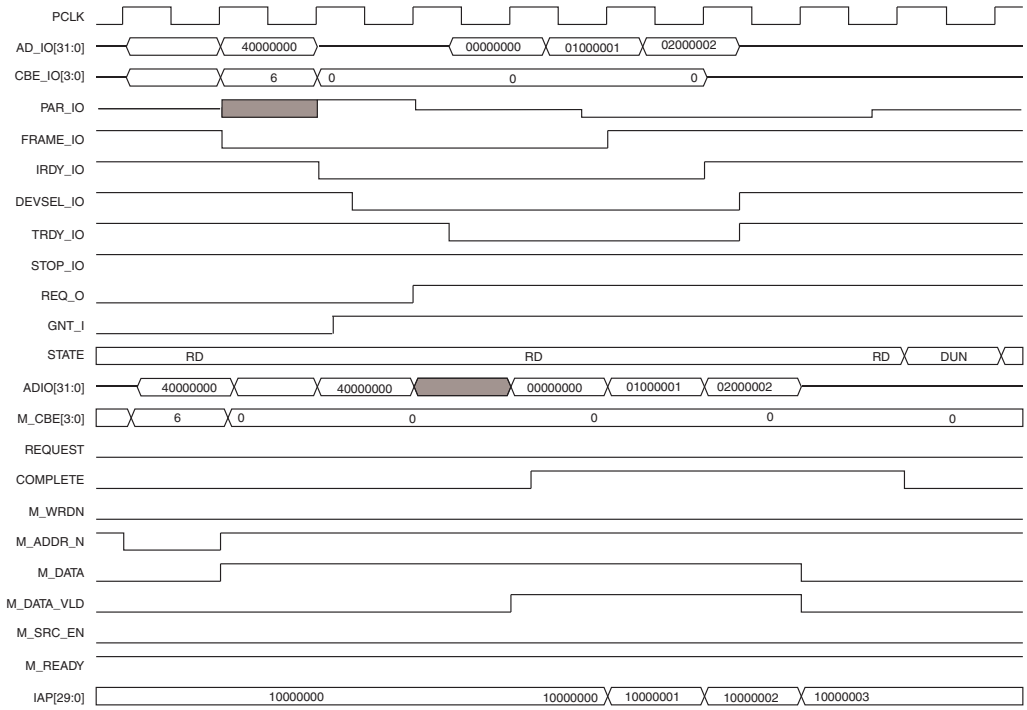


Figure 12-1: **Initiator Burst Read Transaction**

Sourcing Data in Burst Transfers

During initiator writes, the core interface transfers burst data using a pipelined data path. The data source enable signal, M_SRC_EN, is used to advance data pointers in the user

application logic (and possibly the initiator address pointer). The result is that the user application drives new data onto the internal ADIO bus.

Internally, the core interface captures the data value provided by the user application on the ADIO bus and holds this value in the output flip-flops driving the AD_IO pins on the PCI Bus. The user application then presents the next data word on ADIO, instead of holding the previous data word until the current data phase completes.

An initiator burst write is shown in the waveform of Figure 12-2. This waveform includes both PCI Bus signals and internal user application signals. In this figure, the initiator address pointer is implemented to provide a local pointer to index storage elements. During initiator writes, this pointer is incremented when M_SRC_EN is asserted.

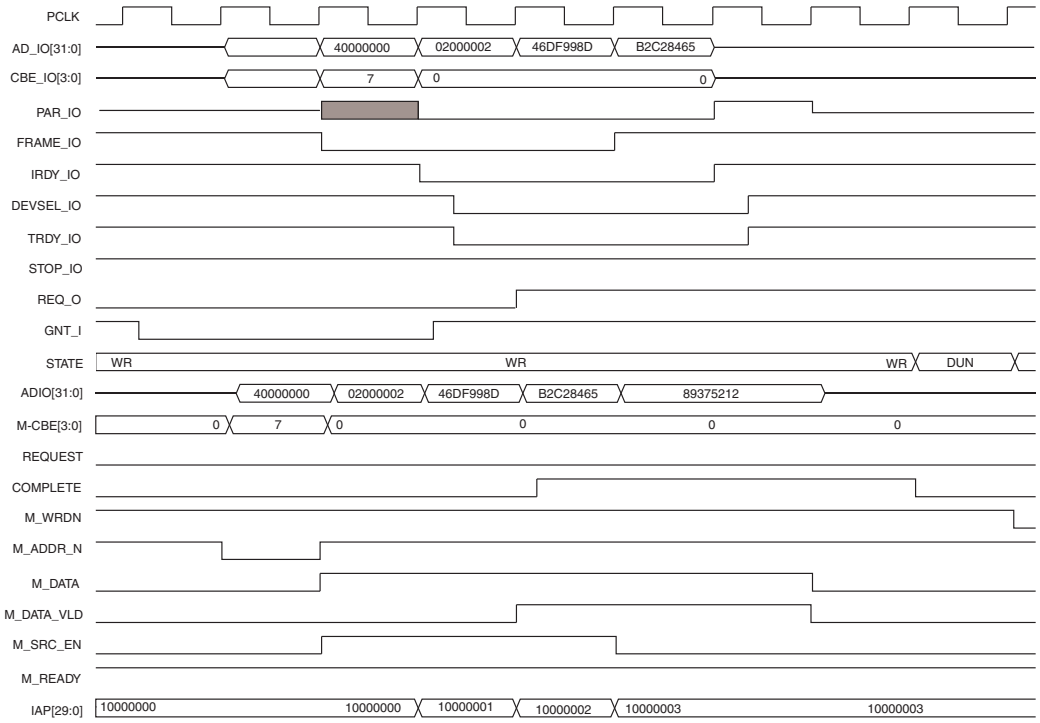


Figure 12-2: **Initiator Burst Write Transaction**

Using M_SRC_EN to present data for the next data phase may require additional control logic depending on the type of data source present in the user application. Keep in mind that the M_SRC_EN signal advances data pointers (and possibly the initiator address pointer) in anticipation of the next data phase, which may or may not complete with successful data transfer.

If a pointer is advanced, and the data is never transferred, then the user application must decide what to do with the non-transferred data. In the case of prefetchable data sources, such as RAM or a register file, the data can be discarded. The original data remains in the RAM or the register file for future use.

This also applies in cases where a FIFO is used as a rate matching buffer and the contents of the FIFO are flushed after a transaction. Any non-transferred data is discarded from the FIFO, but the original data still remains in the source that originally provided it.

For non-prefetchable data sources, as is the case when a FIFO is the data source, pulling data out of the FIFO may be destructive. The unused data must be restored in the data source so it is available for future use should it not be transferred. This may require decrementing internal counters or keeping a shadow copy of the previous data values.

Conditions requiring a backup may arise at the end of an initiator write transfer where the target signals some form of disconnect, terminating the transaction before the initiator is able to complete the full transfer. In these cases, the user application is not immediately aware of the termination condition, and will have advanced the data source too many times. This condition can also arise during data transfers that span multiple bus transactions.

One way to determine the number of times the data pointer (and possibly the initiator address pointer) has been over-advanced during a burst write is to monitor the difference in the number of cycles M_SRC_EN and M_DATA_VLD have asserted during a transaction. During initiator writes, the signal M_DATA_VLD represents the number of data phases that actually complete with data transfer.

Design Example

The following design example demonstrates burst transfers as an initiator using a non-prefetchable data source. In this particular example, explicit local addresses are not required, so the simple initiator address pointer scheme is used.

Non-prefetchable data sources, such as FIFOs, exhibit "side effects" from reads (that is, the state is altered or lost). Special care must be taken during initiator burst writes so that state information is not lost. The use of M_SRC_EN results in reading the data source ahead of the actual transfer. Unless precautions are taken, the data will be lost.

Figure 12-3 shows a FIFO suitable for initiator burst transactions in a user application. To present a concise example, this example uses a single FIFO with both ports accessible through the core interface. In practice, the best structure for most applications is a dual-FIFO design with separate read and write FIFOs.

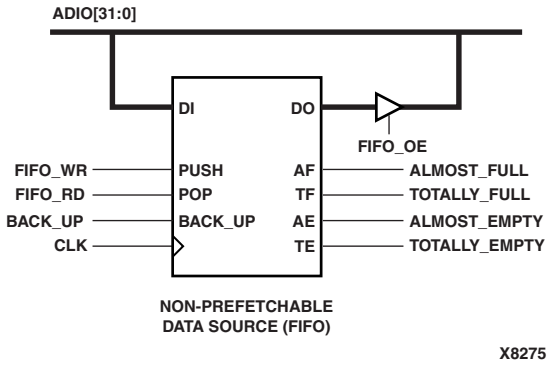


Figure 12-3: **Non-Prefetchable Data Source**

The most crucial element is the FIFO itself. The FIFO must have an additional control signal to back up, or "undo" up to two reads. A typical FIFO implemented in an FPGA consists of a circular buffer implemented with RAM, a read pointer, and a write pointer.

The back up feature can be incorporated in a FIFO by making the actual depth two less than the maximum possible, and by using a bidirectional read pointer. This prevents new data entering the FIFO from overwriting data in the FIFO that may need to be restored.

The FIFO must also have a set of flags that provide FIFO status information. These flags, and their uses, are listed below:

- **TE**. Indicates a totally empty condition. If the FIFO is totally empty and an initiator write transaction is requested, the user application should either ignore the request, postpone it, or flag some sort of error.
- **TF**. Indicates a completely full condition. If the FIFO is totally full and an initiator read transaction is requested, the user application should either ignore the request, postpone it, or flag some sort of error.
- **AE**. Indicates an almost empty condition. When the FIFO becomes almost empty during an initiator write, the user application should signal that it wishes to complete the transaction because it is about to underrun the FIFO.
- **AF**. Indicates an almost full condition. When the FIFO becomes almost full during an initiator read, the user application should signal that it wishes to complete the transaction because it is about to overrun the FIFO.

In most designs, the user application contains a transfer counter which specifies the length of the desired transfer. For designs with transfer lengths less than or equal to the FIFO size, it is not necessary to monitor the AE and AF flags from the FIFO. In this case, the transfer counter may be monitored to determine when the initiator should cease data transfer. That is, there is only one "terminating" condition which occurs when the transfer is complete.

For transfer lengths greater than the FIFO size, the AE and AF flags become important. In this case, the user application must be aware of two "terminating" conditions. One occurs when the transfer is complete, as determined by the transfer counter, and the other occurs when the FIFO becomes almost full (initiator read) or almost empty (initiator write).

In the second case, the FIFO must be emptied (initiator read) or refilled (initiator write) by logic in the user application before it requests another transaction to continue the transfer. The FIFO design may be coupled with a modified initiator control state machine, as discussed in Chapter 10, "Initiator Data Transfer and Control" of this guide. Although the bulk of this example is similar, it is presented here in its entirety to present a complete picture. The entire design breaks down into several distinct sections:

- Inputs to the state machine
- The state machine itself
- Outputs to the core interface
- Control signals

Inputs to the State Machine

The sample state machine requires one input to initiate a start, another input to indicate the direction of the transfer, and six additional inputs to monitor the progression of the transaction.

START indicates to the state machine that it should begin a transaction, and is generated by the user application. The source of this signal varies depending on the specific user application function. The logic in the user application that generates the START signal must not assert START in the following cases:

- Initiator write and FIFO empty
- Initiator read and FIFO full

Failure to observe this will lead to data loss or corruption. Once an initiator transaction is started, at least one data phase will complete.

DIR indicates the direction of the desired transfer, and is generated by the user application. The DIR signal should not change from the time START is asserted until the end of the initiator transfer. Initiator writes are selected by driving DIR to logic high.

M_ADDR_N and **M_DATA** are outputs of the core user interface that are used by the state machine. The signal M_DATA_FELL indicates that a falling edge has been detected on M_DATA. This is generated by the following code.

```
always @(posedge CLK or posedge RST)
begin : edge_detect
   if (RST) M_DATAQ = 1'b0;
   else M_DATAQ = M_DATA;
end

assign M_DATA_FELL = !M_DATA & M_DATAQ;
```

FATAL is derived from core interface signals, and indicates that a master abort or target abort has occurred.

This information is obtained from the extended status signals, CSR[39:32]. The extended status information is valid one clock cycle after a particular event has occurred on the PCI Bus. Unless the status is used during that cycle, it must be registered to preserve it for later use. The following code performs the task of preserving the status information from the final data phase.

```
always @(posedge CLK or posedge RST)
begin : watch_status
   if (RST) FATAL = 1'b0;
   else if (!M_ADDR_N) FATAL = 1'b0;
   else if (M_DATA) FATAL = CSR[39] | CSR[38];
end
```

In the previous example on single transfers, the transaction status was monitored to detect if the target signalled a retry. In this example, the state machine automatically requests transactions until the transfer counter is zero. The actual behavior can be modified to suit the user application by making changes to the state machine.

Note that the *PCI Local Bus Specification* requires initiators to repeat transactions terminated by the target with retry. However, for system robustness, it is advisable to have the initiator monitor the number of times it is retried, so that it can abort transfer attempts that are met with an excessive number of retries from the target.

BACK_UP indicates that the FIFO must be backed up to compensate for speculative reads that can occur during initiator writes. The generation of this signal is discussed after the state machine is presented.

The final signal required by the state machine is the DONE signal. This signal indicates to the state machine that a transaction sequence has completed. This is asserted when the transfer length counter has reached zero.

```
always @(posedge CLK or posedge RST)
begin : transfer_counter
   if (RST) XFER_CNT = 4'h0;
   else if (START) XFER_CNT = BURST_LENGTH;
   else if (M_DATA_VLD) XFER_CNT = XFER_CNT - 4'h1;
```

```
end

assign DONE = (XFER_CNT == 4'h0);
```

This counter is loaded from a BURST_LENGTH signal provided by the user application. Note that this counter does not need to be backed up, as it counts actual transfers, not anticipated transfers.

The State Machine

The state machine is shown in Figure 12-4. This design is intended to perform initiator burst transactions of any length less than the size of the FIFO.

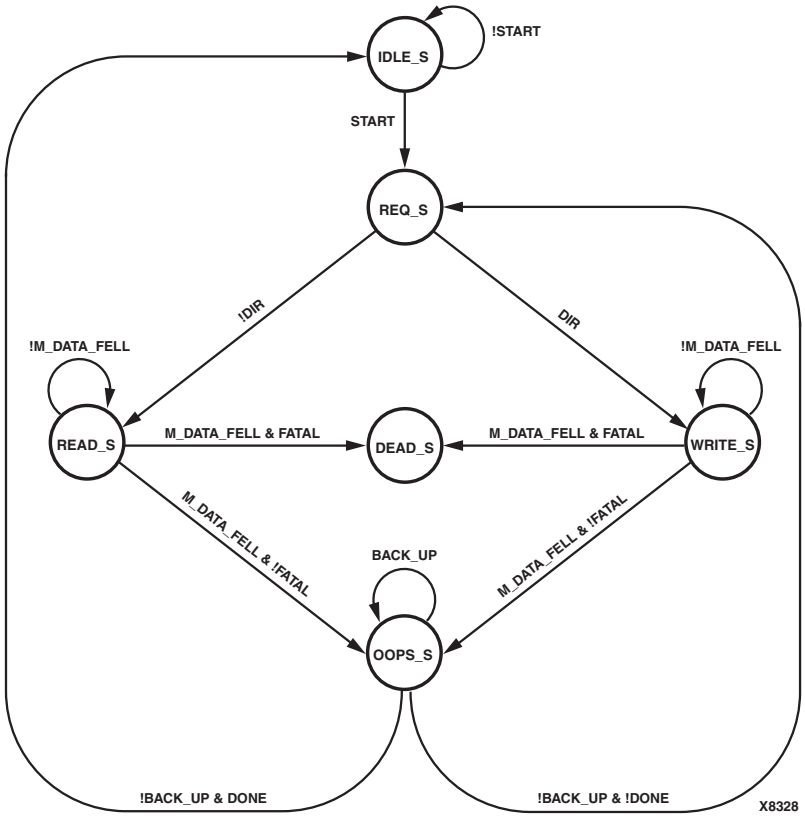


Figure 12-4: **User Application State Machine**

Each state performs a specific step in the sequence. The individual states and their purposes are listed below.

IDLE_S is the idle state. During this state, no initiator activity takes place, and the state machine waits for the user application to assert START. After START has been asserted, the state machine proceeds to the REQ_S state.

REQ_S is the request state. During this state, a request is made to begin an initiator transaction. The state machine then branches based on the value of the DIR signal. If DIR indicates an initiator read, the next state is READ_S. Otherwise, the next state is WRITE_S.

READ_S and **WRITE_S** are data transfer states. The state machine stays in READ_S or WRITE_S until it detects that the transaction is over. The transaction is over when M_DATA_FELL is asserted. This can result from either premature target termination (retry or disconnect), the expiration of the internal latency timer, or natural transaction termination due to the assertion of COMPLETE. The COMPLETE signal is asserted by other logic that monitors FIFO status flags and the transfer counter.

These states are identical except that the state machine outputs differ. The output logic is discussed in the next section. Since DIR is available in the user application, these two states may be reduced into a single XFER_S state if DIR is used in the data register control equations. The states are separate in this example for clarity. In more elaborate designs, it is often desirable to split the state machine into separate read and write state machines.

Whether in READ_S or WRITE_S, the state machine branches after the transfer state based on the FATAL signal. If a fatal error has occurred, the state machine moves to the DEAD_S state. Otherwise, the state machine moves to the OOPS_S state.

DEAD_S is the terminal state for handling fatal errors. In practice, the user application should provide some method for resetting the state machine to handle such errors.

OOPS_S is the transaction evaluation state. If the FIFO must be backed up, it is done in this state. Upon exit from this state, the state machine evaluates the DONE signal. If DONE is asserted, the state machine transitions to IDLE_S, otherwise it transitions to the REQ_S state to request another transaction.

The state machine shown in Figure 12-4 could be implemented as shown:

```
always @(posedge CLK or posedge RST)
begin : initiator_fsm
    if (RST) STATE = IDLE_S;
    else case (STATE)
        IDLE_S :  begin
                  if (START) STATE = REQ_S;
                  else STATE = IDLE_S;
                  end
        REQ_S :   begin
                  if (DIR) STATE = WRITE_S;
                  else STATE = READ_S;
                  end
        WRITE_S : begin
                  if (M_DATA_FELL)
                  begin
                      if (FATAL) STATE = DEAD_S;
                      else STATE = OOPS_S;
                  end
                  else STATE = WRITE_S;
                  end
        READ_S :  begin
                  if (M_DATA_FELL)
                  begin
                      if (FATAL) STATE = DEAD_S;
```

```
                        else STATE = OOPS_S;
                    end
                    else STATE = READ_S;
                    end
            DEAD_S :  STATE = DEAD_S;
            OOPS_S :  begin
                    if (BACK_UP) STATE = OOPS_S;
                    else if (DONE) STATE = IDLE_S;
                    else STATE = REQ_S;
                    end
            default : STATE = IDLE_S;
        endcase
    end
```

Outputs to the Core Interface

The state machine in the user application is responsible for driving several signals which are output to the core interface. These signals are discussed below along with the logic required to generate them. This discussion assumes the existence of a 32-bit START_ADDR signal provided by other logic in the user application. This signal represents the initial PCI Bus address for the desired target. The source of this signal will vary depending on the specific user application function.

Before requesting a transaction, the user application should indicate that it is ready to transfer data. Although it is possible to initiate a transaction and then insert wait states using the M_READY signal, bus bandwidth is conserved by requesting a transaction only after the user application is ready to transfer data. In this example, the user application is always ready.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
    if (RST) M_READY = 1'b0;
    else M_READY = 1'b1;
end
```

The M_READY signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

The state machine initially requests access to the PCI Bus by asserting REQUEST. This is done in the REQ_S state and results in the assertion of REQ_O. The REQUEST signal must only be asserted for a single cycle to initialize a request. Note that the bus master enable bit in the command register must be set before the core interface is able to request the bus. This is the responsibility of the host bridge and system configuration software.

```
assign REQUEST = (STATE == REQ_S);
```

The REQUESTHOLD signal is not used in this example, and is simply assigned to logic zero.

```
assign REQUESTHOLD = 1'b0;
```

When the core interface indicates that it has received a bus grant, the user application must drive the target address on the ADIO bus. This address, stored in the initiator address pointer, is presented on the PCI Bus during the address phase.

```
always @(posedge CLK or posedge RST)
begin : init_addr_pointer
   if (RST) ADDRESS = 30'h0;
   else if (START) ADDRESS = START_ADDR[31:2];
   else if (M_DATA_VLD) ADDRESS = ADDRESS + 30'h1;
end

assign ADDR_OE = M_ADDR_N;
assign ADIO = ADDR_OE ? 32'bz : {ADDRESS, 2'b00};
```

Simultaneously, the user application must also drive M_CBE to indicate the desired PCI Bus command. At other times, the M_CBE signal is used to indicate byte enables. In this example, the initiator performs memory read and memory write transactions (commands 0x6 and 0x7, respectively), depending on the value of the DIR signal. All bytes are enabled. The bus command and byte enables are easily modified to accommodate the requirements of the user application.

```
assign COMMAND = {3'b011, DIR};
assign BYTE_ENABLE = 4'b0000;
assign M_CBE = M_ADDR_N ? BYTE_ENABLE : COMMAND;
```

Throughout the entire transaction, the user application must drive M_WRDN to indicate the desired transfer direction. This signal must remain constant during a transaction. By assigning it to track the DIR signal, which was stipulated to be constant during a transaction, this requirement is met.

```
assign M_WRDN = DIR;
```

The M_WRDN signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

COMPLETE indicates to the core interface that it should finish the current transaction. This is determined by the current transfer counter value. The logic presented below is derived from the example in an earlier chapter on data phase control.

```
assign CNT3 = (XFER_CNT == 4'h3);
assign CNT2 = (XFER_CNT == 4'h2);
assign CNT1 = (XFER_CNT == 4'h1);
```

In this example, initiator wait states are not used. It is critical to reduce the logic as much as possible for timing reasons.

```
assign FIN1 = CNT1 & REQUEST;
assign FIN2 = CNT2 & M_DATAQ;
assign FIN3 = CNT3 & M_DATA_VLD;
assign ASSERT_COMPLETE = FIN1 | FIN2 | FIN3;

always @(posedge CLK or posedge RST)
```

```
begin : finish_up
   if (RST) HOLD_COMPLETE = 1'b0;
   else if (M_DATA_FELL) HOLD_COMPLETE = 1'b0;
   else if (ASSERT_COMPLETE) HOLD_COMPLETE = 1'b1;
end

assign COMPLETE = ASSERT_COMPLETE | HOLD_COMPLETE;
```

The COMPLETE signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, if possible. In practice, it is usually driven by combinational logic.

Note: Never tie this signal to logic one or to logic zero.

To determine the number of times the FIFO must be backed up after an initiator write, the user application must include a small state machine to monitor the difference between anticipated transfers and actual transfers.

Anticipated transfers are signalled by M_SRC_EN, qualified by the TE flag, as an empty FIFO cannot be advanced. Actual transfers are signalled by M_DATA_VLD.

```
assign ANTICIPATED = M_SRC_EN & !TE;

always @(posedge CLK or posedge RST)
begin : oops_counter
   if (RST) OOPS = 2'b00;
   else
   case({ANTICIPATED, M_DATA_VLD, BACK_UP, OOPS})
      5'b00000: OOPS = 2'b00;
      5'b00001: OOPS = 2'b01;
      5'b00010: OOPS = 2'b10;
      5'b00011: OOPS = 2'b11;
      5'b00100: OOPS = 2'b00;
      5'b00101: OOPS = 2'b00;
      5'b00110: OOPS = 2'b01;
      5'b00111: OOPS = 2'b10;
      5'b01000: OOPS = 2'b00;
      5'b01001: OOPS = 2'b00;
      5'b01010: OOPS = 2'b01;
      5'b01011: OOPS = 2'b10;
      5'b01100: OOPS = 2'b00;
      5'b01101: OOPS = 2'b00;
      5'b01110: OOPS = 2'b00;
      5'b01111: OOPS = 2'b01;
      5'b10000: OOPS = 2'b01;
      5'b10001: OOPS = 2'b10;
      5'b10010: OOPS = 2'b11;
      5'b10011: OOPS = 2'b11;
```

```
            5'b10100: OOPS = 2'b00;
            5'b10101: OOPS = 2'b01;
            5'b10110: OOPS = 2'b10;
            5'b10111: OOPS = 2'b11;
            5'b11000: OOPS = 2'b00;
            5'b11001: OOPS = 2'b01;
            5'b11010: OOPS = 2'b10;
            5'b11011: OOPS = 2'b11;
            5'b11100: OOPS = 2'b00;
            5'b11101: OOPS = 2'b00;
            5'b11110: OOPS = 2'b01;
            5'b11111: OOPS = 2'b10;
            default : OOPS = 2'b00;
        endcase
    end

    assign BACK_UP = (| OOPS) & (STATE == OOPS_S);
```

This state machine describes a two bit saturating up and down counter with one increment input and two decrement inputs. As required, it tracks the number of actual transfers versus the number of anticipated transfers. The BACK_UP signal is asserted when the state machine is in the OOPS_S state and the OOPS counter is non-zero. This backs up the FIFO and decrements the OOPS counter until this counter is zero. After BACK_UP is deasserted, the state machine proceeds to the next state.

Control Signals

The logic for the FIFO control signals (Figure 12-3) are as follows:

```
assign FIFO_WR = (STATE == READ_S) & M_DATA_VLD;
assign FIFO_RD = (STATE == WRITE_S) & M_SRC_EN;
assign FIFO_OE = (STATE == WRITE_S) & M_DATA;
```

The generation of the BACK_UP signal, also used by the state machine, was presented in the preceding section.

Sample Transactions

Figures 12-5 through 12-10 are presented to illustrate typical burst read and write transactions that can result from implementing the logic as described in this chapter.

Figure 12-5 demonstrates the core interface filling the FIFO in the user application. The target does not terminate the transaction. This figure also shows the timing relationship between selected user application signals and PCI Bus signals.

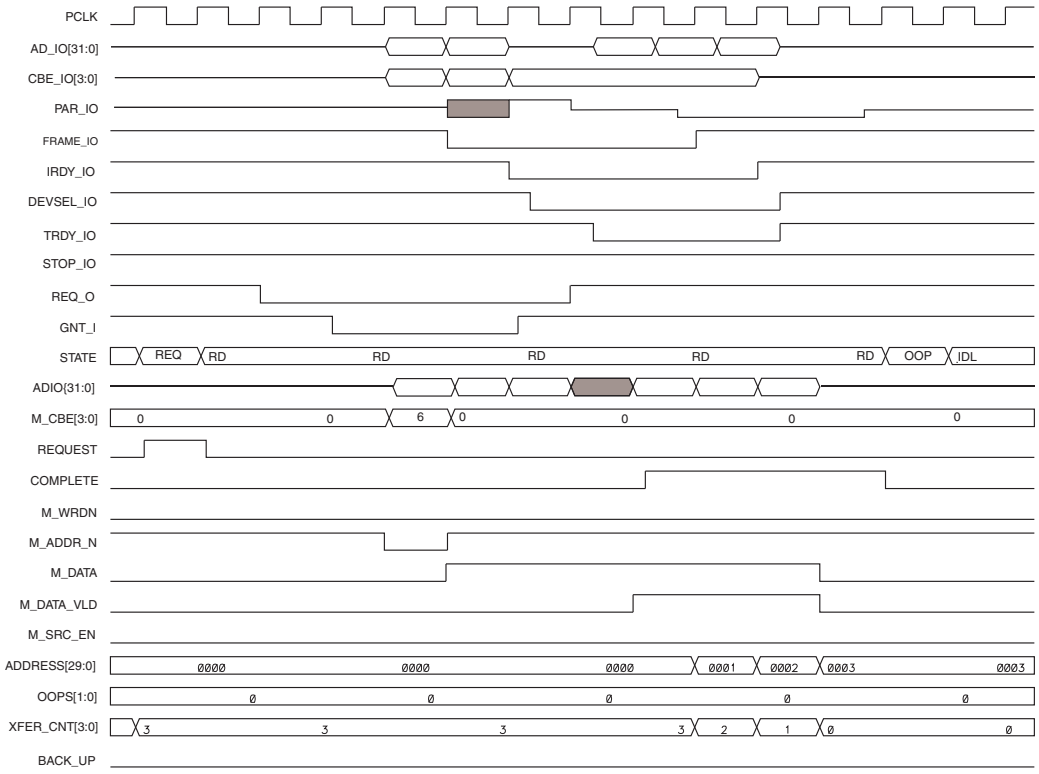


Figure 12-5: **Burst Read Transaction with Normal Termination**

Figure 12-6, demonstrates the core interface emptying the FIFO in the user application. The target does not terminate the transaction. This figure also shows the timing relationship between selected user application signals and PCI Bus signals.

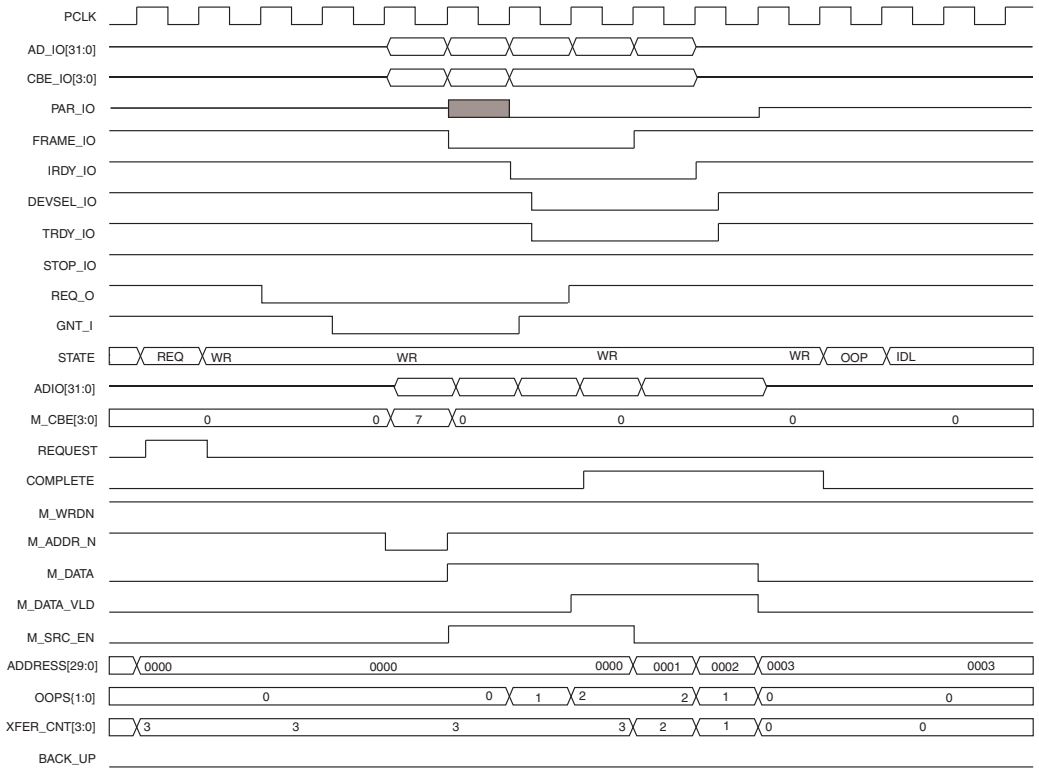


Figure 12-6: **Burst Write Transaction with Normal Termination**

Figure 12-7 demonstrates the core interface filling the FIFO in the user application. During the transaction, the target disconnects.

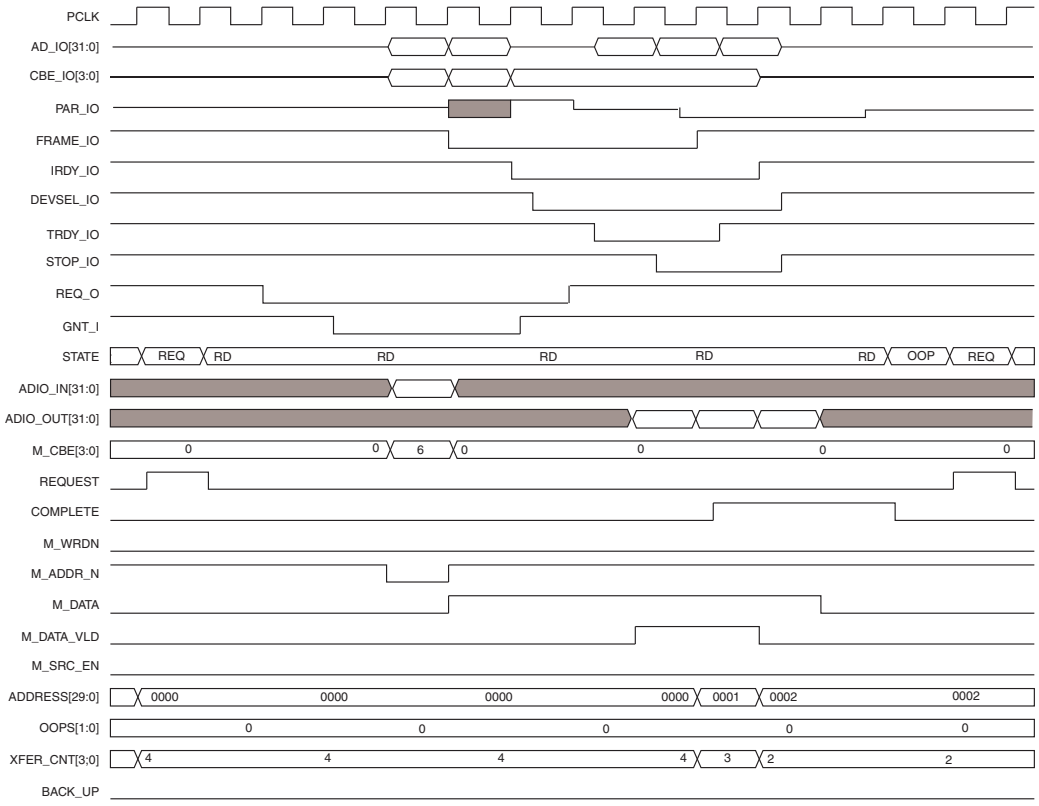


Figure 12-7: **Burst Read with Target Disconnect**

Figure 12-8 shows the user application requesting bus access again to complete the transfer.

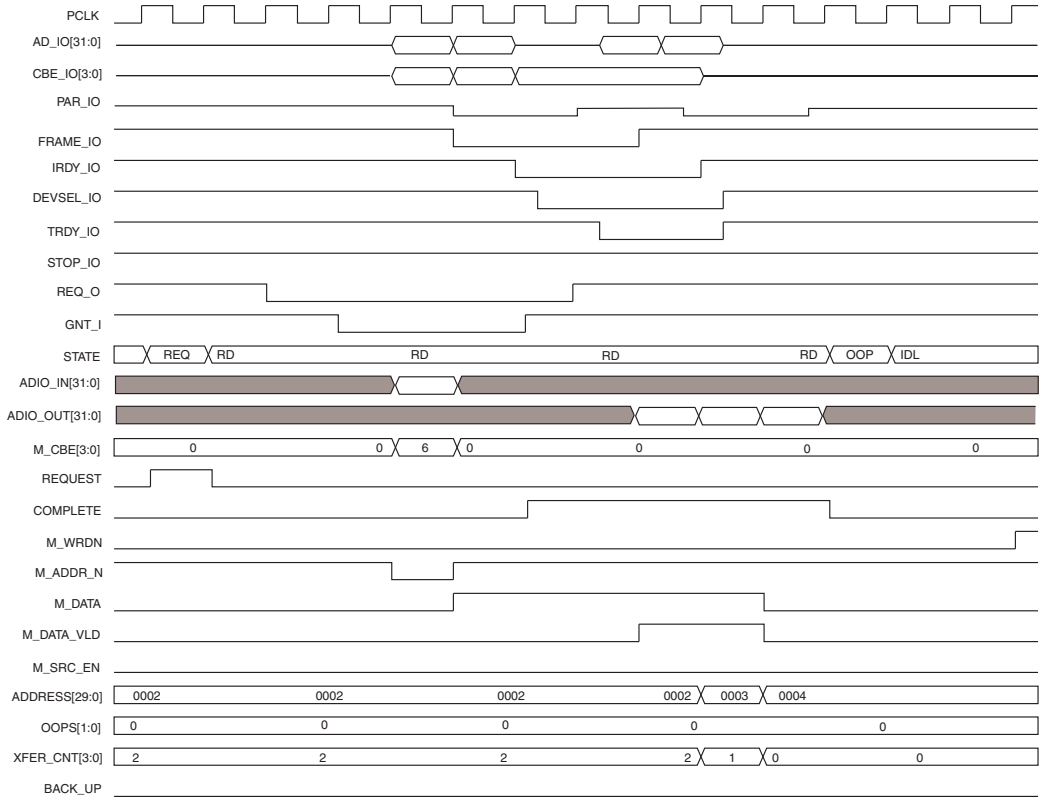


Figure 12-8: **Burst Read Continues after Target Disconnect**

Figure 12-9 demonstrates the core interface emptying the FIFO in the user application. During the transaction, the target disconnects. Notice that the FIFO must be backed up in this situation.

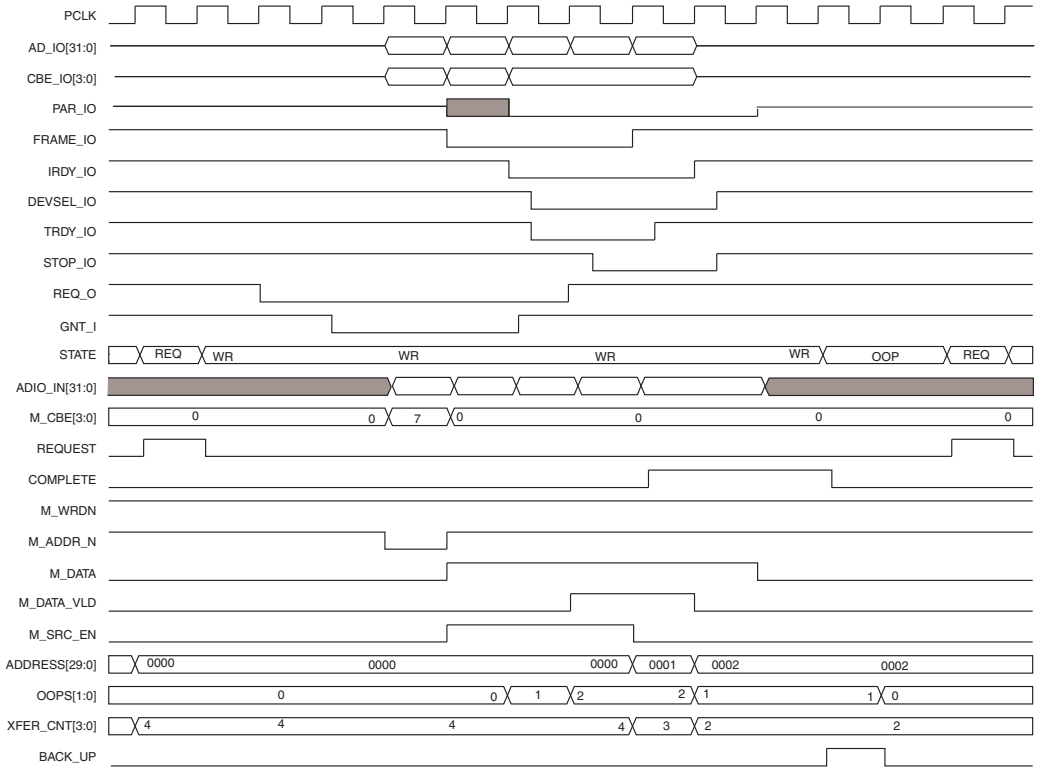


Figure 12-9: **Burst Write with Target Disconnect**

Figure 12-10 shows the user application requesting bus access again to complete the transfer.

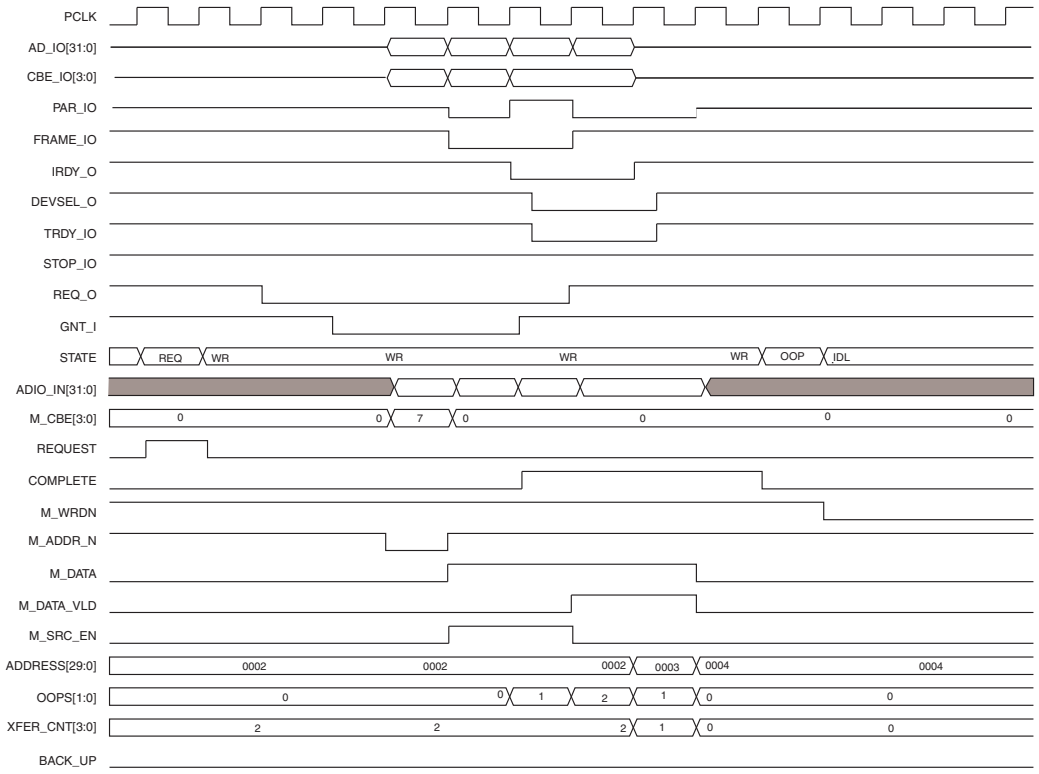


Figure 12-10: **Burst Write Continues after Target Disconnect**

Chapter 13

Initiator 64-bit Extension

This chapter provides detailed information about the initiator 64-bit extension. The implementation of a 64-bit initiator user application is similar to that of a 32-bit version, with some exceptions. In addition to the wider data path, some provisions must be made for exceptional conditions that can occur during 64-bit transfers. Note that 64-bit transfers only apply to memory spaces; other spaces do not support this capability.

Initiator Extension Signals

In addition to the initiator signals presented in Chapter 10, "Initiator Data Transfer and Control," 64-bit implementations of the PCI interface include the following additional signals:

- **ADIO[63:32]**. This bidirectional bus provides the means for 64-bit data and address transfer to and from the PCI interface. When the initiator start address is presented during the assertion of M_ADDR_N by the PCI interface, the high portion of the bus is reserved but must be driven with valid data. When driving data onto the bus, the entire bus width should be driven. Note that 64-bit addressing and dual address cycles are not supported.
- **M_CBE[7:4]**. This output indicates the PCI command and byte enables during an initiator transaction. When the command is presented during the assertion of M_ADDR_N by the PCI interface, the high nibble is reserved but must be driven with valid data. During data phases, this bus should be driven with the byte enables for the extended data path.
- **SLOT64.** This signal enables the 64-bit extension signals (AD_IO[63:32], CBE_IO[7:4], and PAR64_IO). This signal must remain static at all times except at power-on and immediately after a PCI Bus reset. The method for determining the appropriate value for this signal is design and device family dependent. See the *Initiator/Target v3.1 for PCI Getting Started Guide* for more information about SLOT64.
- **REQUEST64.** This output from the user application instructs the PCI interface to request the PCI Bus and to start a 64-bit initiator transaction after GNT_I is asserted.
- **M_FAIL64**. This input indicates that a 64-bit transfer attempt has encountered a 32-bit target. In such situations, the initiator will transfer at most two 32-bit words before terminating the transfer. This signal should be used to adjust the increment (step size) of any initiator address pointers.

Controlling 64-bit Transfers

The concepts described in previous chapters of this guide apply to 64-bit transfers. The steps for transfer requests and data phase control are the same. Extending the data path of

the example presented in Chapter 11, "Initiator Data Phase Control," produces the waveforms below.

Figure 13-1 demonstrates the PCI interface performing a burst read transfer from a 64-bit target. In this case, the REQUEST64 signal is asserted instead of the REQUEST signal The COMPLETE logic remains the same, but the transfer counter (and any address pointer) now must track QWORDS instead of DWORDS.

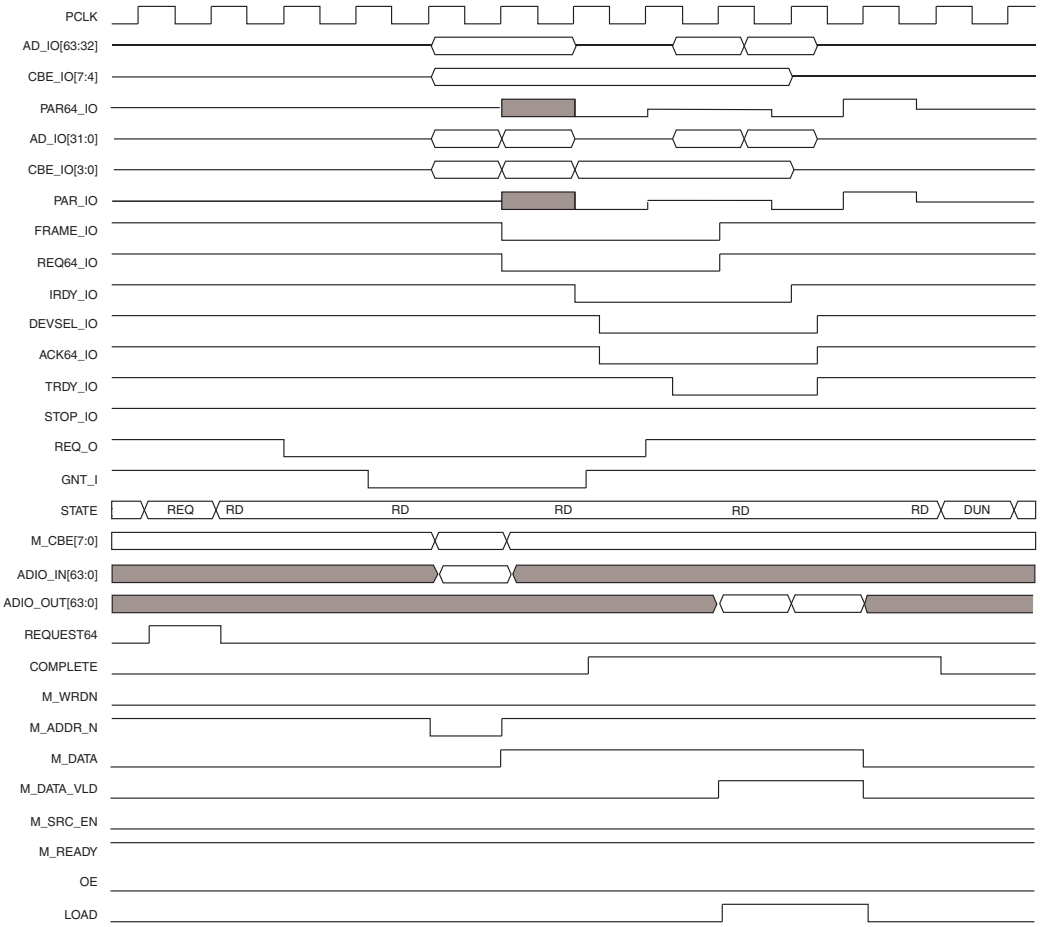


Figure 13-1: **Two QWORD Initiator Read Transfer**

Figure 13-2 demonstrates the PCI interface performing a burst write transfer to a 64-bit target. Again, the REQUEST64 signal is used and the transfer counter (and any address pointer) tracks QWORDs instead of DWORDs.

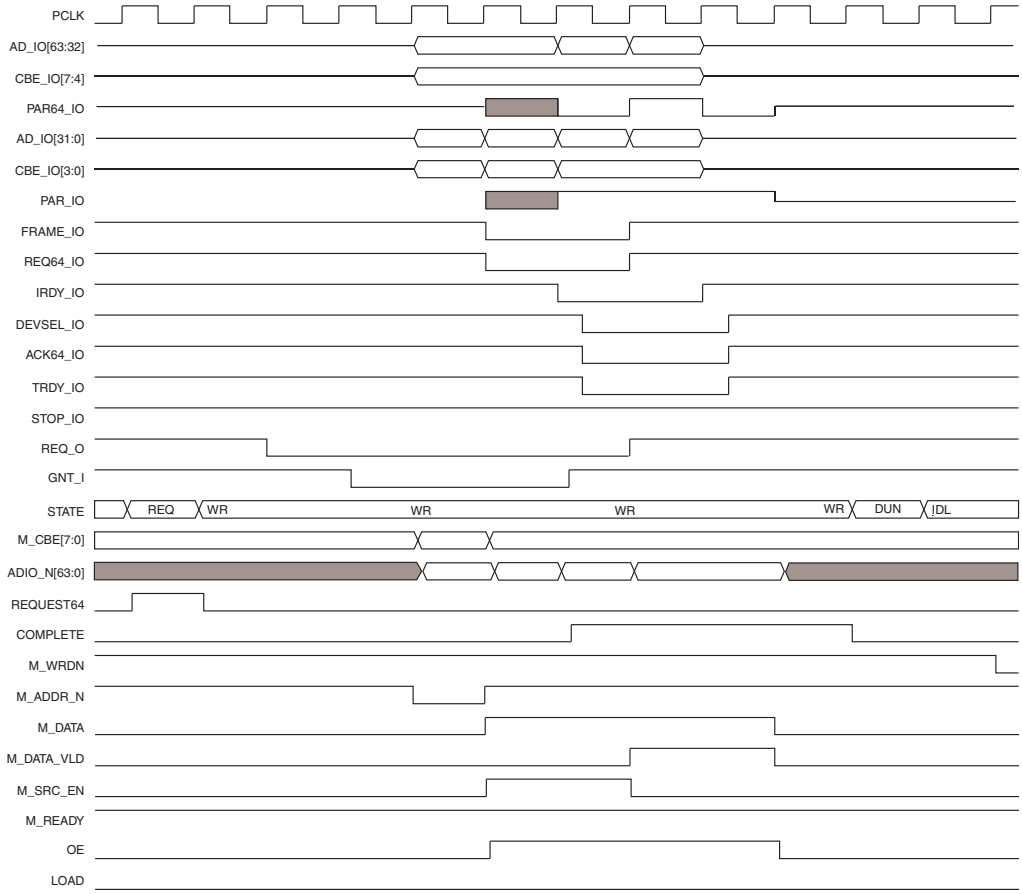


Figure 13-2: **Two QWORD Initiator Write Transfer**

Additional Considerations

The control of 64-bit initiator transfers is similar to the 32-bit case. However, the flexibility of the PCI interface opens the door to potential protocol violations by the user application. For this reason, consider the following when designing an initiator control state machine.

Only Perform Burst Transfers

Do not request single data phase 64-bit transactions. The PCI interface does not know if a 64-bit or a 32-bit target will respond to a 64-bit transaction request. For this reason, the interface does not know when to deassert FRAME_IO and REQ64_IO for a single data phase transfer. Instead, request a 32-bit transfer and perform two data phases.

Only Use Aligned Addresses

The *PCI Local Bus Specification* forbids initiators from requesting unaligned 64-bit transfers. The user application should never initiate a 64-bit transfer with an address that is not aligned on a QWORD boundary. Specifically, the low three address bits must be zero.

Only Use Allowed Commands

The *PCI Local Bus Specification* states that 64-bit transfers apply to memory spaces only. The user application should never initiate a 64-bit transfer using non-memory commands. As a 64-bit initiator, the PCI interface supports the following commands:

- Memory Read
- Memory Write
- Memory Read Multiple
- Memory Read Line

Monitor the Target Response

Under ideal conditions, a 64-bit target will respond to a 64-bit transfer request by the PCI interface. Unfortunately, this response is not guaranteed in an open system. A 32-bit response may occur when:

- The target does not support 64-bit transfers
- The PCI interface is installed in a 32-bit system

In either case, the PCI interface perceives the target as a 32-bit agent on the bus. When the PCI interface encounters a 32-bit target as a 64-bit initiator, it will automatically terminate the transfer after two 32-bit data phases. Depending on the behavior of the target, different results are possible.

If the target terminates the transfer attempt with retry, the user application is required to retry the original 64-bit transaction exactly as it occurred the first time—even though the user application may now be aware that the target is not 64-bit.

When the PCI interface detects a 32-bit target responding to a 64-bit transfer attempt, it asserts `M_FAIL64`. The user application must use this signal to adjust the increment (step size) for any internal counters or pointers. This includes address pointers, data pointers, and back up counters. After the transfer completes, it is the responsibility of the user application to request another transfer and continue as a 32-bit agent.

The following waveforms are an example of possible situations which may arise when a 64-bit transfer request is met with a 32-bit target response. The data path signals in these figures are broken into high and low halves to demonstrate the actual data movement during transfers where `M_FAIL64` is asserted by the PCI interface.

Figure 13-3 shows the PCI interface requesting a 64-bit read burst. The target is a 32-bit agent and does not assert `ACK64_IO` in response to `REQ64_IO`.

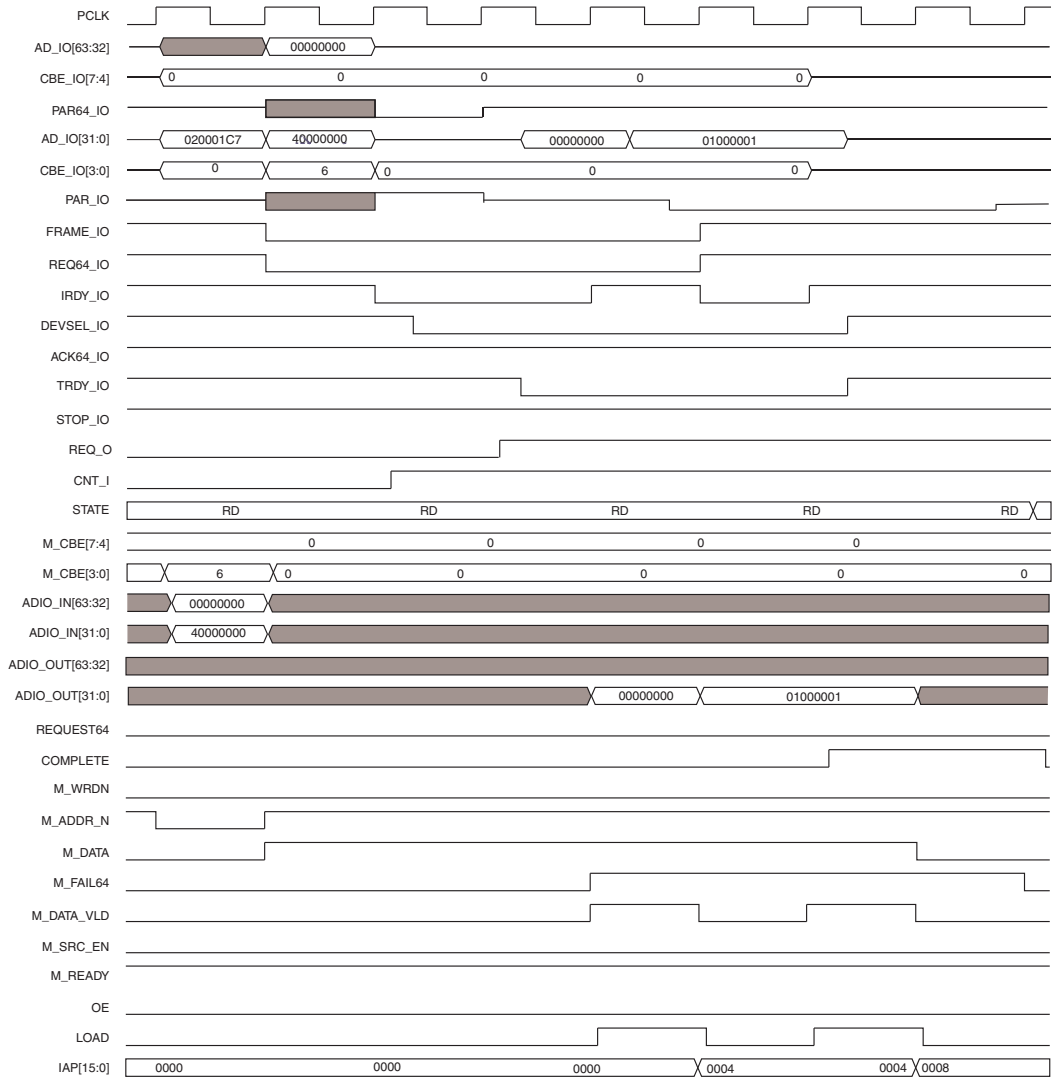


Figure 13-3: **32-bit Target Responds to 64-bit Read**

After the PCI interface detects that the target is a 32-bit agent, it asserts `M_FAIL64`. The PCI interface automatically inserts a wait state after the first data phase. During that wait state, the interface automatically copies the byte enables on `CBE_IO[7:4]` to `CBE_IO[3:0]`. Then the PCI interface completes the transfer on the second data phase. In this example, the target does not disconnect.

The user application must monitor `M_FAIL64` during a transfer to correctly interpret the transaction. If `M_FAIL64` is not asserted, the user application may complete the transfer using the full 64-bit data path. If `M_FAIL64` is asserted, the user application should treat the transfer as a 32-bit transfer. This implies:

- Capturing data from `ADIO[31:0]` and ignoring `ADIO[63:32]`
- Assembling data into 64-bit chunks, if required
- Adjusting the increment (step size) for any counters or pointers

At the end of this transfer, the bus address is `QWORD` aligned. Although the *PCI Local Bus Specification* permits this transfer to be continued by requesting another 64-bit transfer, it is

unwise to do so from a bandwidth perspective. After the target is determined to be a 32-bit agent, the user application should request a 32-bit transfer.

Figure 13-4 demonstrates a transaction similar to the one in Figure 13-3, with the exception of the target disconnecting after the first data phase.

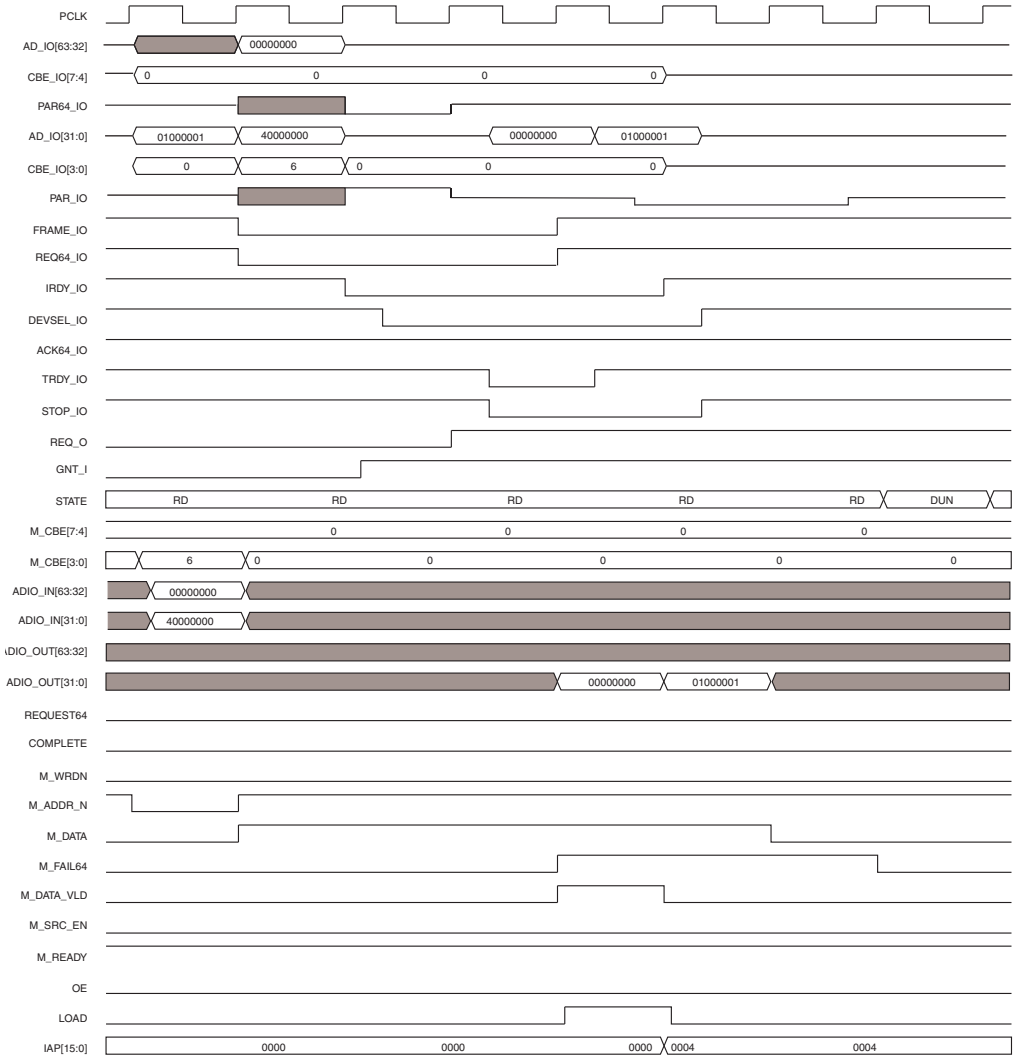


Figure 13-4: **32-bit Target Disconnects with Data on Read**

After the PCI interface detects that the target is a 32-bit agent, it asserts `M_FAIL64`. Because the target disconnects, the PCI interface does not insert a wait state to multiplex the byte enables for the second data phase.

As before, the user application must monitor `M_FAIL64` during the transfer to correctly interpret the transaction. If `M_FAIL64` is not asserted, the user application may complete the transfer using the full 64-bit data path. If `M_FAIL64` is asserted, the user application should treat the transfer as a 32-bit transfer. This implies:

- Capturing data from `ADIO[31:0]` and ignoring `ADIO[63:32]`
- Assembling data into 64-bit chunks, if required
- Adjusting the increment (step size) for any counters or pointers

At the end of this transfer, the bus address is *not* QWORD aligned. The *PCI Local Bus Specification* stipulates that this transfer must be continued as a 32-bit transfer due to the misalignment.

Figure 13-5 shows the PCI interface requesting a 64-bit write burst. The target is a 32-bit agent and does not assert ACK64_IO in response to REQ64_IO.

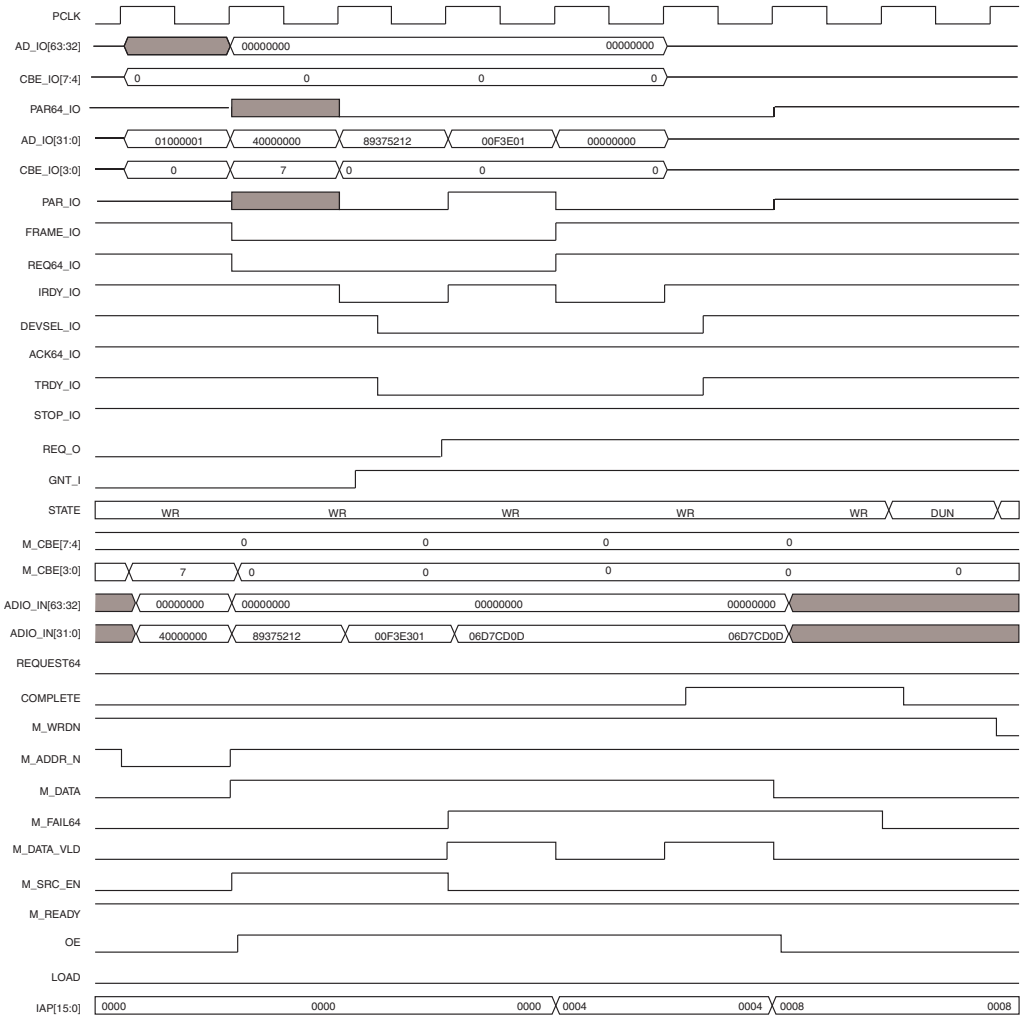


Figure 13-5: **32-bit Target Responds to 64-bit Write**

After the PCI interface detects that the target is a 32-bit agent, it asserts M_FAIL64. The PCI interface automatically inserts a wait state after the first data phase. During that wait state, the interface automatically copies the byte enables on CBE_IO[7:4] to CBE_IO[3:0]. In the next cycle, the interface copies the data from AD_IO[63:32] to AD_IO[31:0]. Then the PCI interface completes the transfer on the second data phase. In this example, the target does not disconnect.

Because M_FAIL64 is asserted, the user application must treat the transfer as a 32-bit transfer and adjust the increment (step size) for any counters or pointers. The PCI interface automatically handles the data multiplexing internally when performing initiator writes.

At the end of this transfer, the bus address is QWORD aligned. Although the *PCI Local Bus Specification* permits this transfer to be continued by requesting another 64-bit transfer, it is

unwise to do so from a bandwidth perspective. After the target is known to be a 32-bit agent, the user application should request a 32-bit transfer.

Figure 13-6 demonstrates a similar transaction to Figure 13-5 with the exception of the target disconnecting after the first data phase.

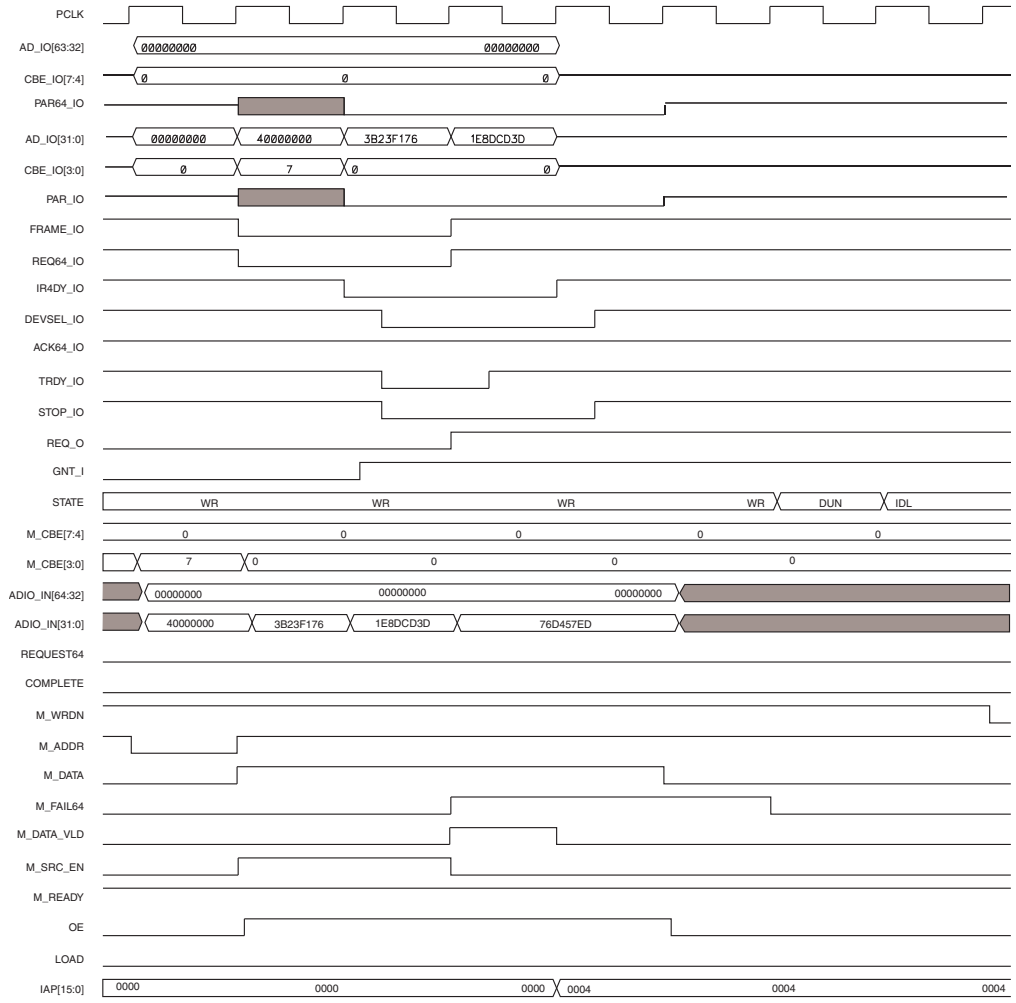


Figure 13-6: **32-bit Target Disconnects with Data on Write**

After the PCI interface detects that the target is a 32-bit agent, it asserts `M_FAIL64`. Since the target disconnects, the PCI interface does not insert a wait state to multiplex the data and byte enables for the second data phase.

Because `M_FAIL64` is asserted, the user application must treat the transfer as a 32-bit transfer and adjust the increment (step size) for any counters or pointers. The PCI interface automatically handles the data multiplexing internally if the target does not disconnect.

At the end of this transfer, the bus address is *not* `QWORD` aligned. The *PCI Local Bus Specification* stipulates that this transfer must be continued as a 32-bit transfer due to the misalignment.

XILINX®

Chapter 14

Other Bus Cycles

This chapter provides information about some of the more esoteric commands in the PCI interface. In addition to supporting memory read and memory write commands, the PCI interface supports a host of other PCI Bus commands as both target and initiator. Many of these commands do not require additional design effort. The examples presented in earlier chapters can be easily modified to support other commands. The first part of the chapter discusses target operations, and the second part discusses initiator operations.

As a target, the PCI interface can support I/O commands if at least one of the Base Address Registers is configured as an I/O space. Additionally, the PCI target can handle all extended memory commands such as memory read multiple, memory read line, and memory write and invalidate, as long as at least one Base Address Register is configured as a memory space.

As an initiator, the PCI interface can issue both I/O commands and all memory commands except memory write and invalidate. The user application simply presents the appropriate command to the PCI interface when initiating a transfer. Note that I/O commands may not be used with multiple data phase transfers, as required by the *PCI Local Bus Specification*.

Supported Commands

Table 14-1 provides a list of commands supported by the PCI interface.

Table 14-1: **Supported PCI Bus Commands**

Command		Support	
Code	**Name**	**Target**	**Initiator**
0000	Interrupt Acknowledge	Yes	Yes
0001	Special Cycle	Ignore	Yes
0010	I/O Read	Yes	Yes
0011	I/O Write	Yes	Yes
0100	Reserved	Ignore	No
0101	Reserved	Ignore	No
0110	Memory Read	Yes	Yes
0111	Memory Write	Yes	Yes
1000	Reserved	Ignore	No
1001	Reserved	Ignore	No

Table 14-1: **Supported PCI Bus Commands *(Continued)***

Command		Support	
1010	Configuration Read	Yes	Yes
1011	Configuration Write	Yes	Yes
1100	Memory Read Multiple	Yes	Yes
1101	Dual Address Cycle	Ignore	No
1110	Memory Read Line	Yes	Yes
1111	Memory Write Invalidate	Yes	No

Note: *Yes* commands indicate full support; *Ignore* commands are not supported by the PCI target; *No* commands are not supported by the PCI initiator and must not be used.

Target Commands

Interrupt Acknowledge

Interrupt acknowledge commands issued on the PCI Bus are implicitly addressed to the system interrupt controller. These bus cycles are typically initiated by the host bridge.

In designs where the user application contains the system interrupt controller, it is necessary for the PCI interface to respond to interrupt acknowledge cycles as a target. This behavior is enabled using the Interrupt Acknowledge configuration option as discussed in Chapter 4, "Customizing the Interface." Enabling this option forces the last available Base Address Register in the PCI interface to respond to interrupt acknowledge cycles.

```
always @(posedge CLK or posedge RST)
begin : decode_int_ack
   if (RST) INT_BAR = 1'b0;
   else if (BASE_HIT[2]) INT_BAR = 1'b1;
   else if (!S_DATA) INT_BAR = 1'b0;
end
```

In this example, assume the existence of a 32-bit signal named **VECTOR** that holds the interrupt vector. This signal is generated elsewhere in the user application.

```
assign ACK_OE = INT_BAR & S_DATA;
assign ADIO = ACK_OE ? VECTOR : 32'bz;
```

As a general rule, interrupt acknowledge cycles may be terminated like other bus cycles. For this example, however, the transfer is not terminated by the PCI interface.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized
   if (RST)
   begin
      S_ABORT = 1'b1;
      S_READY = 1'b0;
      S_TERM = 1'b1;
   end
```

```
            else
            begin
                S_ABORT = 1'b0;
                S_READY = 1'b1;
                S_TERM = 1'b0;
            end
        end
```

Figure 14-1 displays the waveform generated from the code sample above. In this situation, the interrupt vector is returned to the host bridge during the cycle.

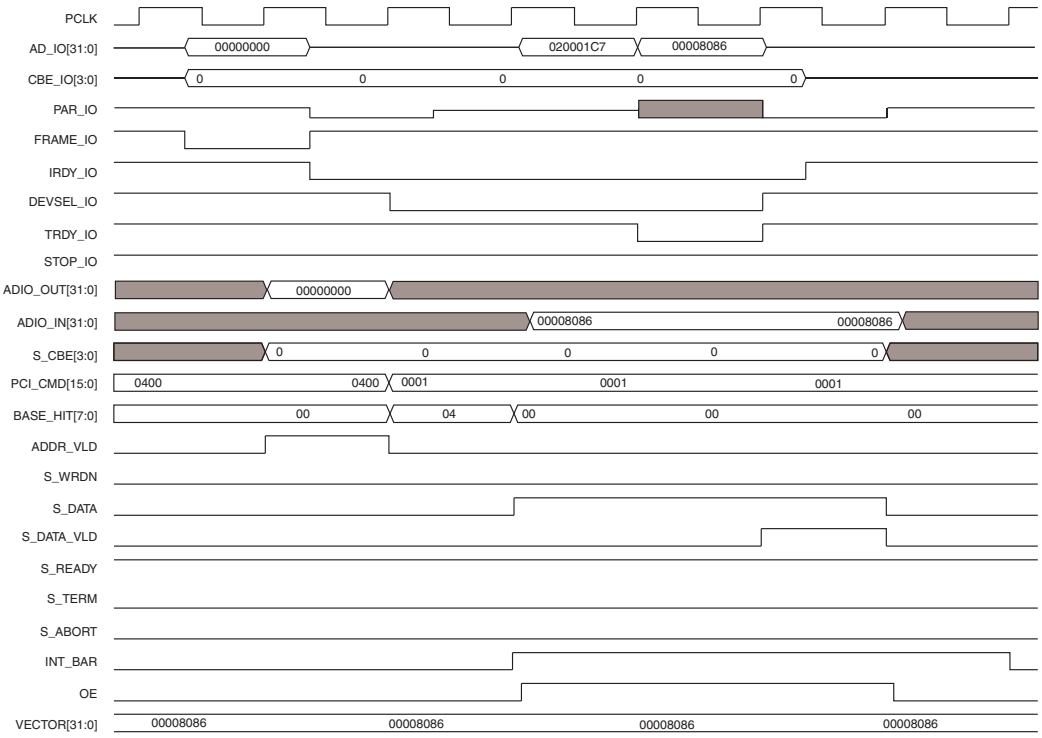


Figure 14-1: **Target Accepts Interrupt Acknowledge Cycle**

Configuration Cycles

The *PCI Local Bus Specification* defines three address spaces:

- Memory space
- I/O space
- Configuration space

The PCI interface, by default, implements the first 64 bytes of a type zero configuration space header, shown in Table 14-2. Additional configuration space is available if the User Config Space option is enabled in the configuration file.

Table 14-2: **PCI Configuration Space Header**

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Rev ID	08h
BIST	Header Type	Lat. Timer	Line Size	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Base Address Register 2				18h
Base Address Register 3				1Ch
Base Address Register 4				20h
Base Address Register 5				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			CapPtr	34h
Reserved				38h
Max_Lat	Min_Gnt	Intr. Pin	Intr. Line	3Ch
User Config Space Begins				40h

All unimplemented configuration space registers return a value of zero during configuration read cycles and no operation occurs during configuration write cycles.

Shaded address locations are not implemented in the PCI interface default configuration. These locations return zero during configuration read accesses. Other locations, such as the CapPtr, or Base Address Registers, will return zeros if disabled.

This section discusses some features of the configuration space that exist in the PCI interface, and presents methods for handling other configuration space accesses. Typically, user application designs will not need to extend the configuration space that is pre-implemented in the PCI interface.

This information is intended mainly for advanced users who wish to implement the user accessible area of the configuration space. This section is divided into the following sub-sections:

- Setup for Typical Applications
- User Definable Configuration Space
- Capabilities List Pointer
- Externally Supplied Subsystem Identification

Typical Application Setup

Configuration transactions are automatically handled by the PCI interface. The signals `C_TERM` and `C_READY` must be asserted by the user application at all times to allow the

PCI interface to respond to the supported configuration space addresses. All unsupported addresses return zero when accessed, as mandated by the *PCI Local Bus Specification*.

For a typical application that uses only the pre-implemented configuration space, **C_TERM** and C_READY must always be asserted to ensure that the PCI interface terminates all configuration accesses by disconnecting with data on the first data phase. The PCI interface does not handle burst configuration cycles and must disconnect.

```
assign C_READY = 1'b1;
assign C_TERM = 1'b1;
```

Note: The **ADIO** bus is actively driven by the PCI interface during configuration cycles. Care should be taken as to avoid contention on the **ADIO** bus.

User-Definable Configuration Space

The user-definable region of configuration space resides at and above address 0x40, up to the top of configuration space at 0xFF. Access to this region is controlled by the User Config Space option in the configuration file.

With this option disabled, the PCI interface returns zeros when this region is accessed. If the option is enabled, the PCI interface does not automatically return zeros, but instead allows the user application to treat the configuration access like any other non-burst target access.

Data is transferred much like non-burst target accesses. A typical configuration register is interfaced as shown in Figure 14-2.

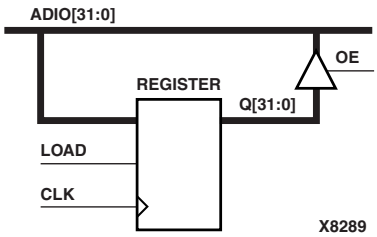


Figure 14-2: **Example Configuration Register**

The following signals are associated with configuration data transfer to and from the PCI interface. These signals are used in conjunction with other signals used for target data transfer. References to inputs and outputs are made with respect to the user application.

- **CFG_HIT**. This input indicates that the PCI interface recognizes that it is the target of a current PCI configuration transaction. This signal is similar to the **BASE_HIT** signals used in target accesses.
- **CFG_VLD**. This input indicates that a valid PCI configuration address is available on the **ADIO** bus, and may be used as a clock enable by the user application to capture a copy of this address. As the PCI interface does not support configuration burst transactions, the latched address present on the ADDR[31:0] bus should suffice. The CFG_VLD signal is asserted for a single cycle, coincident with ADDR_VLD.
- **C_READY**. This output from the user application indicates that it is ready to transfer data, and can be used to insert wait states during the first data phase of a transaction. Together with C_TERM, it is also used to signal different types of target termination for configuration accesses. This signal has the same functionality as **S_READY** for target accesses.

- **C_TERM**. This output from the user application indicates that data transfer should cease. It is also used with C_READY to signal different types of target termination for configuration accesses. This signal has the same functionality as **S_TERM** for target accesses.

The Verilog pseudocode to decode configuration transactions can be represented as follows:

```
always @(posedge CLK or posedge RST)
begin : decode_cfgspace
   if (RST)
   begin
      CFG_RD = 1'b0;
      CFG_WR = 1'b0;
   end
   else
   begin
      if (CFG_HIT)
      begin
         CFG_RD = !S_WRDN;
         CFG_WR = S_WRDN;
      end
      else if (!S_DATA)
      begin
         CFG_RD = 1'b0;
         CFG_WR = 1'b0;
      end
   end
end
```

This is nearly identical to the decoding used for normal target transactions. In this case, the PCI Bus command has been pre-decoded as a configuration read or configuration write command through the logic that generates **CFG_HIT**.

Configuration Writes

During a configuration write operation, data is captured from the **ADIO** bus to a data register in the user application by asserting the load input. The assignment for the load input of the register would be:

```
assign LOAD = CFG_WR & S_DATA_VLD & fn(ADDR[7:2]);
```

If the user application supports byte-addressable registers, separate load signals should be generated for each byte in the register by further gating the expression shown above. The S_CBE signals are available for this purpose.

Configuration Reads

During a target read operation, data from the user application is driven onto the **ADIO** bus. To do this, the user application must assert the output enable for the desired register. The assignment for the output enable would then be:

```
assign OE = CFG_RD & fn(ADDR[7:2]) & S_DATA;
```

Note: Do not drive the ADIO bus from the user application when the configuration transaction is not addressing user configuration space. The PCI interface responds to addresses below 0x40 and will drive ADIO during accesses of this region. When the address is 0x40 and above, always drive the entire ADIO bus with valid data. Unimplemented regions must return zero.

Configuration Data Phase Control

Data phase control is achieved using the C_TERM and C_READY signals. Combinations of the two control signals yield the following three modes:

- **Wait**. Inserts wait states at the beginning of a configuration transaction.
- **Retry**. Terminates the current PCI Bus transaction without data transfer on the final data phase.
- **Disconnect with data**. Terminates the current PCI Bus transaction with data transfer on the first data phase.

Table 14-3 shows the three modes of operation and the corresponding C_TERM and C_READY values. Never assert C_READY while C_TERM is deasserted, because the core target does not support configuration bursts as a target.

Table 14-3: **Configuration Data Phase Control**

Condition	Bus Signals	From User Application	
		C_TERM	C_READY
Wait	**TRDY_IO** = 1 **DEVSEL_IO** = 0 **STOP_IO** = 1	Low	Low
Retry	**TRDY_IO** = 1 **DEVSEL_IO** = 0 **STOP_IO** = 0	High	Low
Disconnect With Data	**TRDY_IO** = 0 **DEVSEL_IO** = 0 **STOP_IO** = 0	High	High

Note: The C_TERM and C_READY control signals affect termination for all of configuration space, not just user configuration space.

If user configuration space must function differently from the pre-implemented configuration space in the PCI interface, use the example shown below as a guide.

```
always @(posedge CLK or posedge RST)
begin : cbl_timer
   if (RST) TIMER = 4'h0;
   else if (CFG_VLD) TIMER = BL_WAIT[3:0];
   else if (TIMER != 4'h0) TIMER = TIMER - 4'h1;
end

assign BLAT_RDY = (TIMER <= 4'h4);
```

```
assign USER_CFG = ADDR[7] | ADDR[6];
assign TERMINATE = (!USER_CFG | BLAT_RDY) & S_DATA;

always @(posedge CLK or posedge RST)
begin : keep_it_registered
   if (RST)
   begin
      C_READY = 1'b0;
      C_TERM = 1'b0;
   end
   else
   begin
      C_READY = (CFG_RD | CFG_WR) & TERMINATE;
      C_TERM = (CFG_RD | CFG_WR) & TERMINATE;
   end
end
```

In this example, the PCI interface inserts wait states until it decodes the address of the configuration access. Configuration space accesses are not bandwidth critical, and the method in this example is simple to implement.

Accesses to address 0x40 and above are delayed by the insertion of additional wait states. The number of wait states are specified by the value of the `BL_WAIT` signal. This example generates transactions like those shown below.

In Figure 14-3, the PCI interface is delaying the completion of a configuration read. This particular transaction is targeting user configuration space.

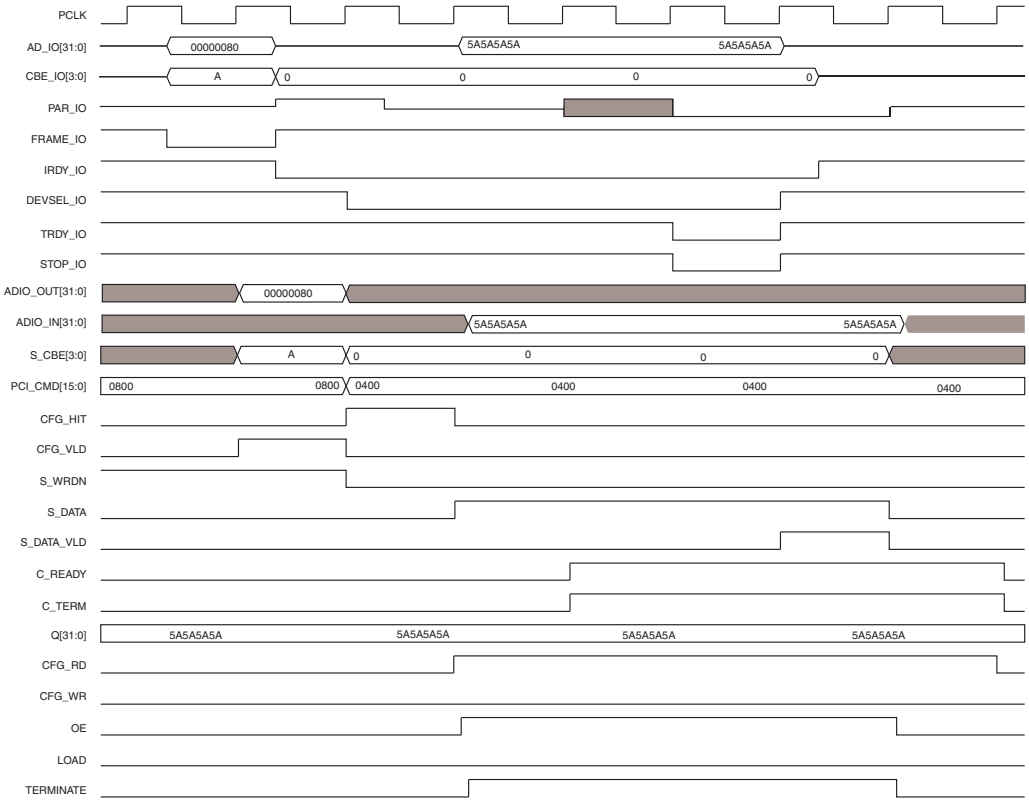


Figure 14-3: **Configuration Read with Wait States**

In Figure 14-4, the PCI interface is delaying the completion of a configuration write. This particular transaction is also targeting user configuration space.

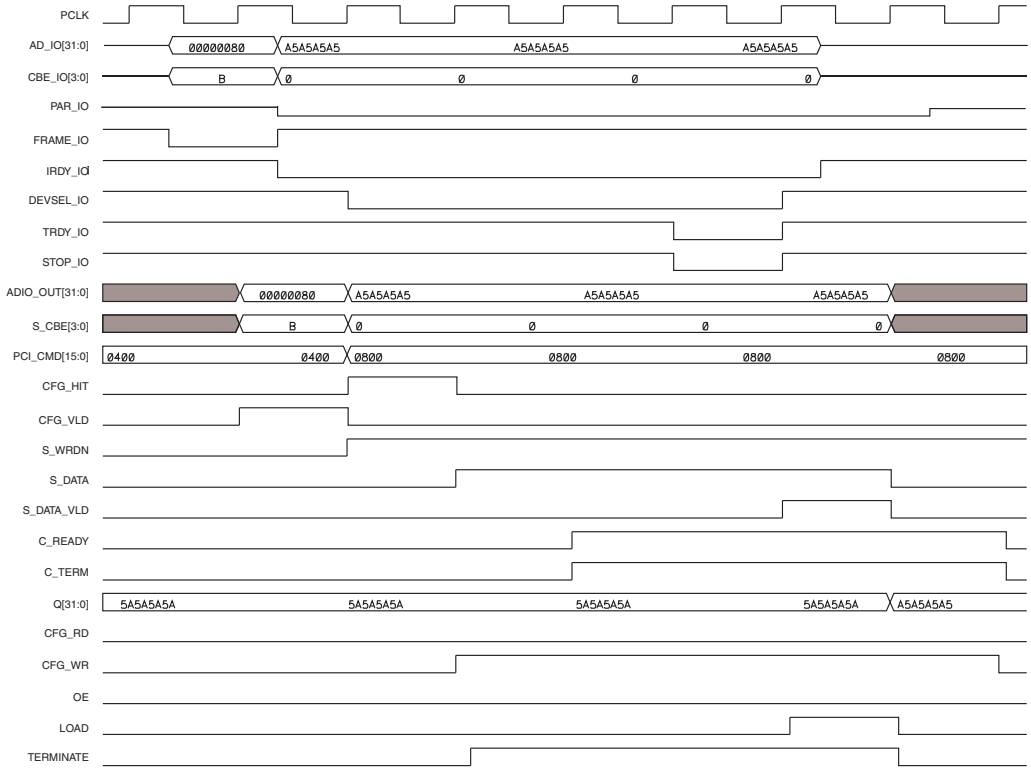


Figure 14-4: **Configuration Write with Wait States**

Capabilities List Pointer

The PCI interface supports a capabilities list through two options in the configuration file. The capabilities bit in the PCI status register indicates whether or not the design implements a linked list of extended capabilities. This bit is set through the Capabilities List Enable option in the configuration file.

If the capabilities bit in the status register is set, the design must implement the Cap_Ptr register at configuration space address 0x34. The Cap_Ptr register contains the configuration space address of the first item in the capabilities list. The capabilities list is a linked list of registers for implementing various features. Power management registers are one such feature that are linked into this list.

The PCI interface implements the Cap_Ptr register. The value of this register is set through the Capabilities List Pointer option in the configuration file. Values 0x00 through 0x3F are not valid values for the Cap_Ptr register because they point into the standard PCI configuration header.

After the capabilities bit is enabled and the Cap_Ptr is set to an appropriate value, the user application must implement a capabilities list in user configuration space.

Externally Supplied Subsystem Identification

The Subsystem Vendor ID and Subsystem ID fields are used to further differentiate designs that are founded on the same base design.

For example, consider a video controller produced by a semiconductor manufacturer. The semiconductor manufacturer sets the Vendor ID and Device ID to identify the video controller as their design, but provides a means for various OEMs to set their own Subsystem Vendor ID. This way, products from different OEMs may be distinguished from each other, even though they use the same graphics chip. The same idea applies to the Subsystem ID, which may be used to distinguish different *trim* levels of a particular product from any single OEM.

For most designs, these values are stored in a configuration ROM table, as specified through the **CFG** bus and the static configuration file. For applications that require a dynamic method of supplying this information, the SUB_DATA bus is available. The user should enable the External Subsystem ID and Subvendor ID option in the configuration file and drive the proper value onto SUB_DATA.

This feature is provided so that a single design (bitstream) may be used in multiple board designs where a different Subsystem Vendor ID or Subsystem ID (or both) are required on each board.

Typical storage implementations for the external value include pullup and pulldown resistors on user I/O pads or an external serial memory. Regardless of the storage method, the value on SUB_DATA must be stable and valid before the PCI interface can respond to configuration cycles.

Using pullup and pulldown resistors on user I/O pads to drive SUB_DATA provides stable and valid data shortly after power is applied. In this case, C_TERM and C_READY may be permanently asserted.

If the user application drives SUB_DATA with a value loaded from an external serial memory, the data may not be immediately available. Depending on the design, the user application may have to generate configuration retries to allow time for SUB_DATA to become valid.

Initiator Commands

Interrupt Acknowledge

Interrupt acknowledge commands issued on the PCI Bus are implicitly addressed to the system interrupt controller. These bus cycles are typically initiated by the host bridge. If the PCI interface and user application are performing the functions of a host bridge, the user application may issue interrupt acknowledge cycles as an initiator.

The example presented in theChapter 10, “Initiator Data Transfer and Control” of this guide is sufficient to issue interrupt acknowledge transactions. The only change required is to alter the COMMAND value to specify the correct bus command:

```
assign COMMAND = 4'b0000;
```

Note that the interrupt acknowledge command uses implied addressing (the system interrupt controller is the implied target). For this reason, the address presented during the address phase of the transaction is not important. However, the AD_IO bus must be driven with stable values. For this reason, do not neglect to drive the ADIO bus with valid data during the assertion of M_ADDR_N, even if initiating an interrupt acknowledge.

Figure 14-5 shows the PCI interface issuing such a transaction.

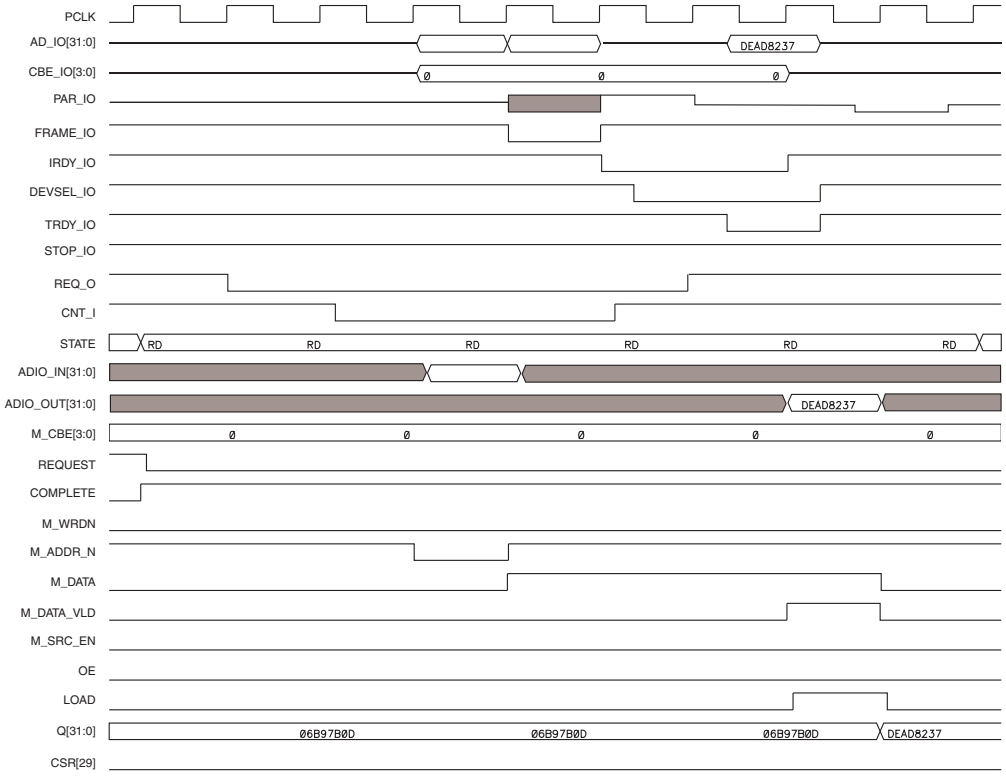


Figure 14-5: **Initiator Issues Interrupt Acknowledge Cycle**

The interrupt vector returned by the system interrupt controller is stored in a register named Q.

Special Cycle Commands

Special cycle commands issued on the PCI Bus are broadcast cycles to one or more agents on the bus. These bus cycles are typically initiated to relay important system information across the bus. The user application may issue special cycles as an initiator. In order for this to be useful, there must be another agent on the bus capable of monitoring special cycles.

Note: The PCI target cannot monitor special cycles.

The example presented in Chapter 10, "Initiator Data Transfer and Control" is sufficient to issue special cycle transactions. The only change required is to alter the COMMAND value to specify the correct bus command:

```
assign COMMAND = 4'b0001;
```

Special cycles are *addressed* to potentially every agent on the PCI Bus. For this reason, the address presented during the address phase of the transaction is not important. However, the AD_IO bus must be driven with stable values. For this reason, do not neglect to drive the **ADIO** bus with valid data during the assertion of **M_ADDR_N**.

No agent on the bus should respond to a special cycle by asserting DEVSEL_IO. Since no agent responds, the PCI interface must perform a master abort to end the transaction. Furthermore, because the PCI interface knows that it is issuing a special cycle (and is expecting a master abort) it should not set the master abort bit in the status register.

Figure 14-6 shows the PCI interface issuing such a transaction.

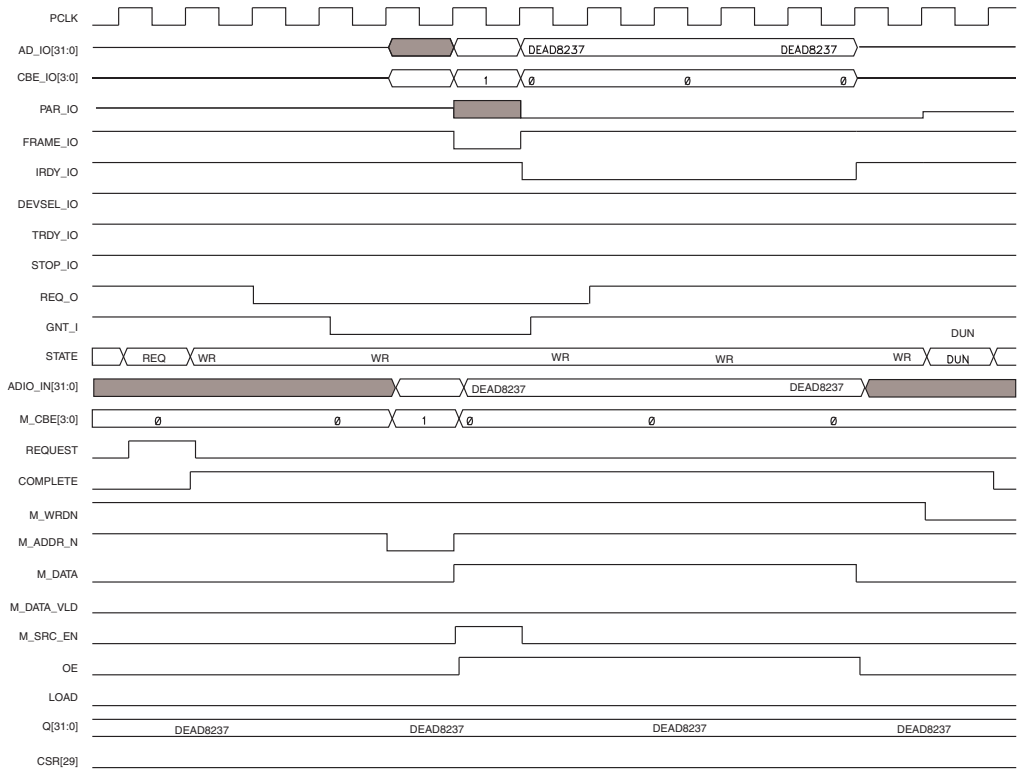


Figure 14-6: **Initiator Issues Special Cycle**

The message field contained in the register named `Q` is broadcast on the PCI Bus. Note that the master abort bit, **`CSR[29]`**, is not set even though the transaction terminates with master abort.

Configuration Cycles

In a typical PCI Bus system, reading and writing of configuration registers is performed by a host bridge. The host bridge, sometimes called a north bridge, is the hardware that bridges a host processor bus and the PCI Bus.

The PCI interface is capable of acting as a host bridge. A host bridge is a subset of a central resource as defined in Section 2.4 of the *PCI Local Bus Specification*. A central resource, and a host bridge as part of a central resource, require additional design considerations beyond those for a generic PCI agent. Some of these design considerations are system dependent.

Note: This chapter does not cover the system-dependent design considerations of a central resource. It is the responsibility of the designer to address issues such arbitration, interrupt handling, error handling, pullups, and the source of the system reset and clock signals. Designers of PC-AT central resources should carefully review Section 3.7.4 of the *PCI Local Bus Specification*, for additional details on compliance requirements specific to the PC architecture.

There are three items of particular interest when designing a host bridge with the PCI interface. The first is the generation of `IDSEL` signals, the second is the inclusion of appropriate logic in the user application to allow the PCI interface to configure itself, and the third is the generation of configuration cycles to external agents.

IDSEL Signal Generation

Section 3.7.4. of the *PCI Local Bus Specification* states that the method used to drive the **IDSEL** signals is left to the discretion of the host bridge designer. However, only a single IDSEL signal may be asserted during the address phase of a configuration transaction. One method mentioned in the specification is to use the upper word of the address bus as the IDSEL signals. Thus, IDSEL of device 0 is connected to AD[16], and so on. This allows for sixteen devices to reside on the PCI Bus.

Due to the extra loading on the AD bus by the IDSEL pins, the IDSEL pin is usually resistively coupled to the AD bus. One exception to this is noted in the electrical specification. If the input pin capacitance of the IDSEL pin of the agent is 8 pF or less, then direct coupling of the AD bus to the IDSEL pin is allowed.

An alternative method is to use separate IDSEL output pins on the host bridge and route them to the various agents on the bus using separate PCB traces.

If a resistor is used, then the signal driving the IDSEL pin may take a long time to reach a valid logic level. Host bridges may solve the timing problem by using address stepping to drive the address, but not assert FRAME#, for one or more clock cycles. This allows the IDSEL pin to reach a valid logic level. Although certain implementations of the PCI interface use address stepping, the PCI interface does not support this technique.

The designer should either use direct coupling or generate separate IDSEL signals. For embedded applications, direct coupling of IDSEL is recommended. The slight increase in bus propagation time may be acceptable in many closed systems. It is the responsibility of the designer to ensure that timing requirements are not violated.

Self Configuration Cycles

Since the initiator and target state machines in the PCI interface are independent, it is possible to use the initiator state machine to generate a configuration transaction aimed at the target state machine.

The steps for generating self configuration cycles are very similar to the steps for generating single data phase initiator transactions. The initiator control signals are driven the same as in any other initiator transaction, and the user application presents a configuration read or configuration write command on M_CBE.

The IDSEL signal routed to IDSEL_I on the PCI interface must be asserted during the address phase of the transaction. This is easily accomplished by using the direct coupling method for IDSEL generation described in the previous section. When the user application drives an address onto the ADIO bus during M_ADDR_N assertion, the bit pattern in the upper word of the address should result in IDSEL_I assertion for the interface.

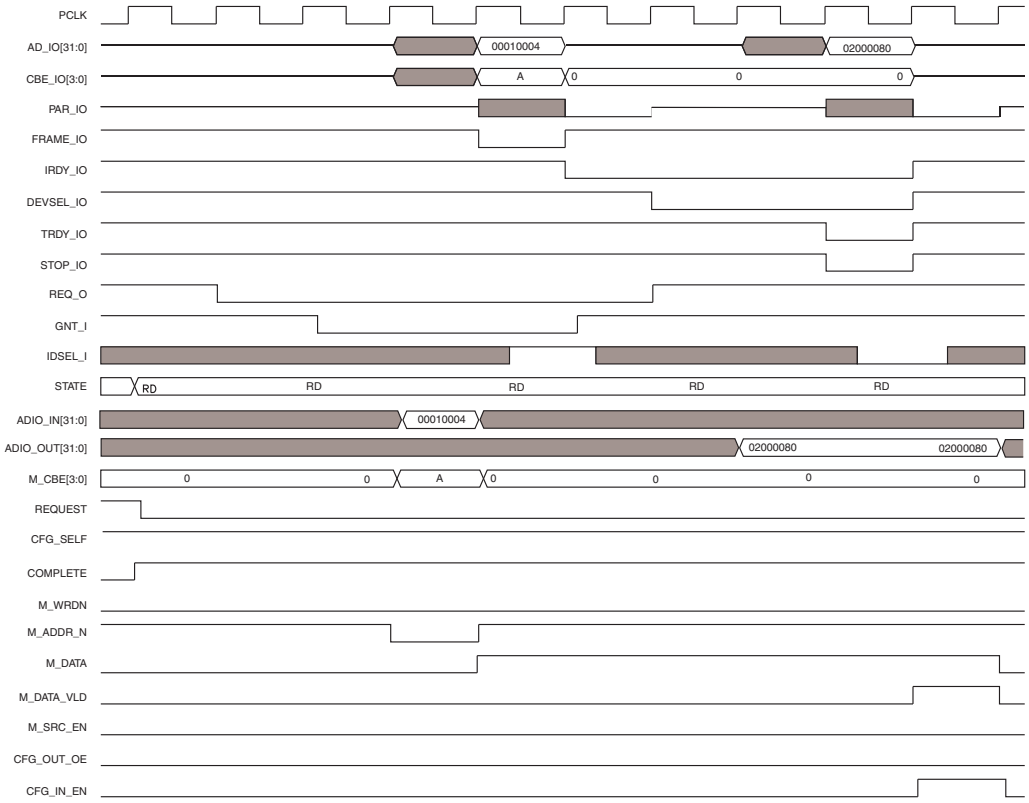


Figure 14-7: **Self-Configuration Read**

A self-configuration cycle must be signaled by the user application no later than one cycle before REQUEST is asserted. This is done by asserting CFG_SELF. This signal must remain asserted until the transaction is complete, which is signalled by the deassertion of M_DATA. Figure 14-7 shows the correct behavior and demonstrates the PCI interface reading its own command and status register.

Internally, the interface uses the CFG_SELF signal to temporarily force the value of the bus master enable bit of the command register. This allows the interface to generate configuration transactions as a bus master even when the bus master enable bit is cleared after a system reset.

The CFG_SELF signal also affects the internal data path, indicating to the PCI interface that data will be transferred between the user application and the interface over the ADIO bus. For this reason, CFG_SELF must always be asserted during self configuration transactions, even after the bus master enable bit has been set.

During self configuration reads, the AD_IO bus is driven with invalid data during the data phase of the transfer. The valid data is available internally on the ADIO bus and is available to the user application. This is generally not a concern, as the PCI interface is both the initiator and the target.

No other agents will respond to the configuration transaction because no other agent will sample its **IDSEL** asserted. Any agent attempting to snoop the transaction (such as a bus analyzer) will not obtain meaningful data.

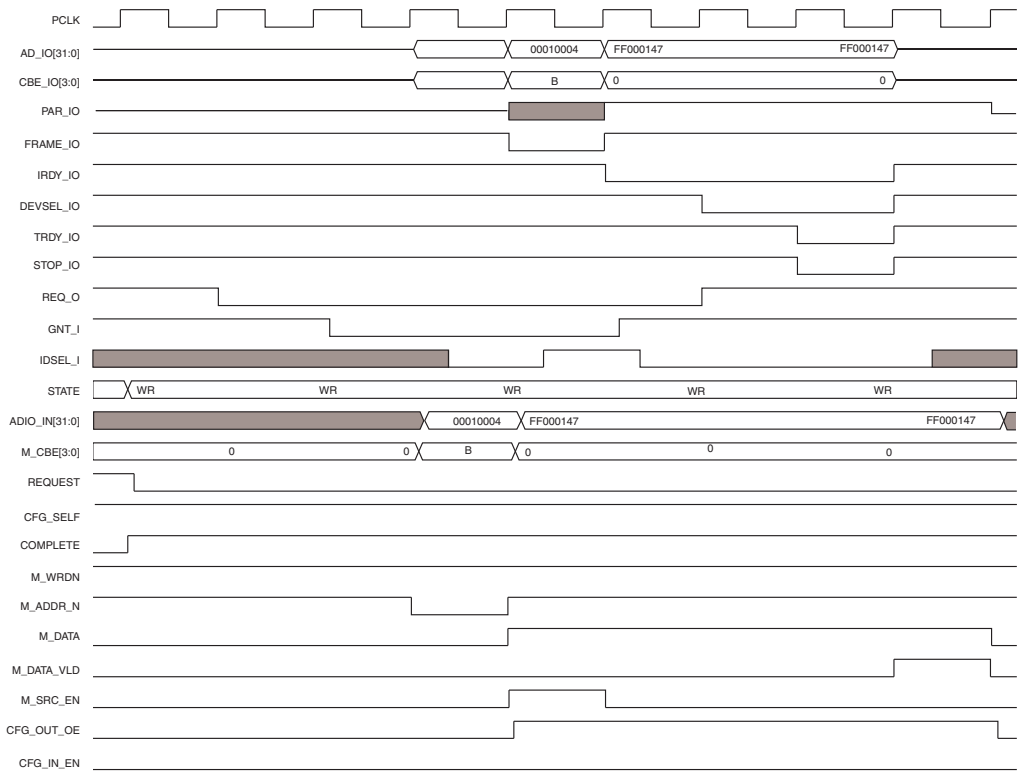


Figure 14-8: **Self-Configuration Write**

Figure 14-8 also demonstrates the correct behavior. In this instance, the PCI interface is enabling itself by writing to its command and status register. Notice that in both figures, `IDSEL_I` is sampled during the address phase; at other times it is ignored.

Generation of Configuration Cycles to Other Agents

After the host bridge is configured and enabled using self configuration cycles, it can be used to configure other agents on the PCI Bus.

The steps for generating configuration cycles are identical to the steps for generating other initiator transactions. The initiator control signals are driven the same as in any other initiator transaction, and the user application presents a configuration read or configuration write command on `M_CBE`.

The `IDSEL` signal for the configuration target must be asserted during the address phase of the transaction. Again, this may be accomplished by using the direct coupling method for `IDSEL` generation. When the user application drives an address onto the `ADIO` bus during `M_ADDR_N` assertion, the bit pattern in the upper word of the address should result in **`IDSEL`** assertion for the configuration target.

When configuring external agents, the signal `CFG_SELF` must not be asserted. Asserting this signal will result in a corrupt data transfer.

Other Considerations

While the *PCI Local Bus Specification* permits burst configuration transactions and the PCI interface is capable of issuing such transactions, it must respond to them with disconnect

with data. Internally, the PCI interface does not keep an address pointer for configuration space addresses. To ensure this, tie the C_TERM and C_READY signals to logic high.

Do not attempt self configuration burst transactions. It is further recommended that burst cycles not be used for any configuration transactions. The use of burst configuration will greatly complicate the design of the user application.

Configuration transactions terminated with retry must be reissued, as is the case with all other transactions terminated with retry. The host bridge designer needs to determine the best order to perform retries.

One method, since configuration sequences are not usually time critical, is to retry the current transaction until it is completed or has timed out. Another method is to push this responsibility on to the configuration software by reporting retries to the host, and have the configuration software accept responsibility for reissuing the transaction at a later time.

During the polling sequence for external agents, it is fully expected that some master abort terminations will occur when polling empty PCI slots. The host bridge application needs to be able to report this condition to the host. Master abort terminations, as well as target abort terminations, should not disable the host bridge from performing further configuration cycles.

Chapter 15

Error Detection and Reporting

The PCI interface generates and checks parity and reports errors as required by the *PCI Local Bus Specification*. For more information regarding these specifications, see Section 3.8 of the *PCI Local Bus Specification*. This chapter contains information about the behavior of the PCI interface when it encounters parity errors.

These functions are completed transparently to the user application. Certain classes of user applications, however, may need to know if an error has occurred. Figure 15-1 illustrates a target write not disrupted by parity errors. Subsequent sections of this chapter discuss the response of the PCI interface when parity errors are introduced during the address phase and data phases.

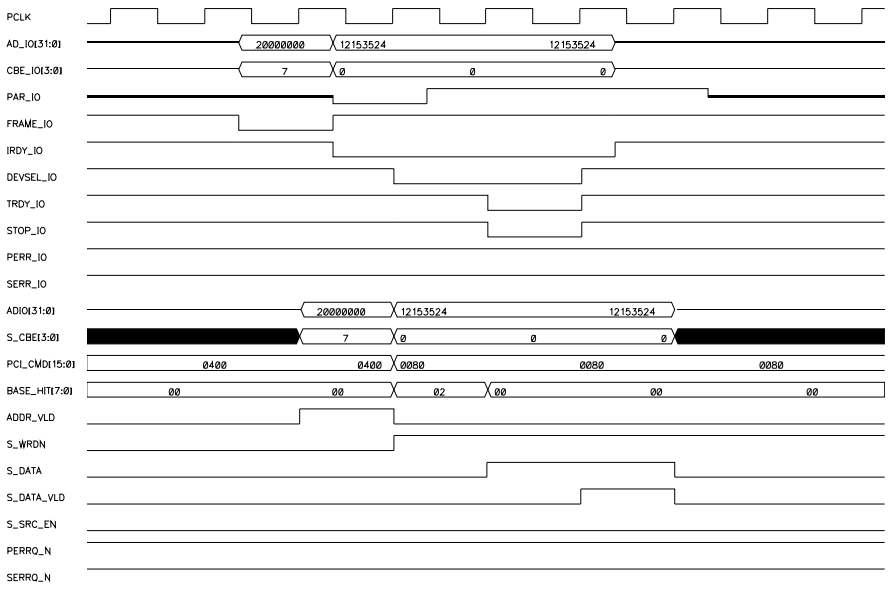


Figure 15-1: **Target Write with no Parity Errors**

The above figure shows the error reporting signals on the PCI Bus (PERR_IO and SERR_IO) and the registered copies of these signals provided to the user application (PERRQ_N and SERRQ_N).

Address Parity Errors

The PCI interface checks for parity errors during address phases on the PCI Bus. As required by the *PCI Local Bus Specification*, the PCI interface reports address parity errors

via SERR_IO. A registered version of this signal is available to the user application as SERRQ_N.

If an address parity error is detected, the PCI interface either claims the transaction and issues a target abort, or does not claim the transaction at all. Figure 15-2 shows the same transaction presented in Figure 15-1 but with an address parity error. Because the PCI interface does not respond to the transaction, the initiator terminates with master abort.

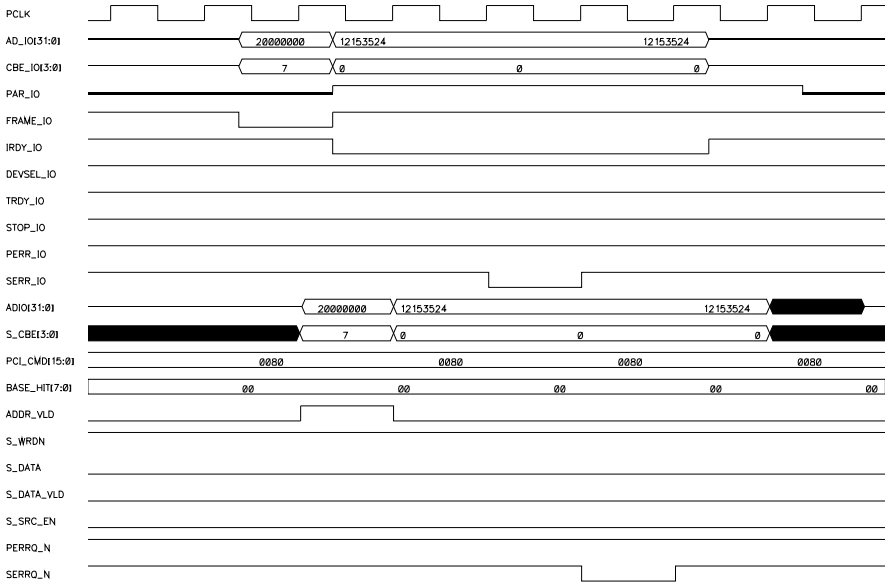


Figure 15-2: **Target Write with Address Parity Error**

The user application may monitor for system errors on the PCI Bus by sampling SERRQ_N. This signal indicates an address parity error or other serious system error. In addition, the command and status register state is available through the CSR[39:0] bus.

The SERRQ_N signal passed to the user application is a registered version of the system error signal, SERR_IO. This signal is for reporting catastrophic system errors. Typically, the assertion of this signal will result in a non-maskable interrupt being issued to the host. If the user application is a host bridge design, it should monitor the SERRQ_N signal and act accordingly.

Note: The SERR_IO signal is an open drain signal. It is passively deasserted by a pullup after synchronous assertion by a bus agent. This transition may take more than one clock cycle, which creates the possibility of a metastable output on SERRQ_N. The logic in the user application that generates a non-maskable interrupt should be designed with this in mind. Use of a synchronizer is recommended.

Data Parity Errors

The PCI interface checks for parity errors during data phases on the PCI Bus when it is receiving data. As required by the *PCI Local Bus Specification*, the PCI interface reports data parity errors via PERR_IO. A registered version of this signal is available to the user application as PERRQ_N.

If a data parity error is detected, the PCI interface will take no additional action other than signalling the error. However, if the user application may take any steps it deems

necessary, including transfer termination. Figure 15-3 shows the same transaction presented in Figure 15-1, but with a data parity error.

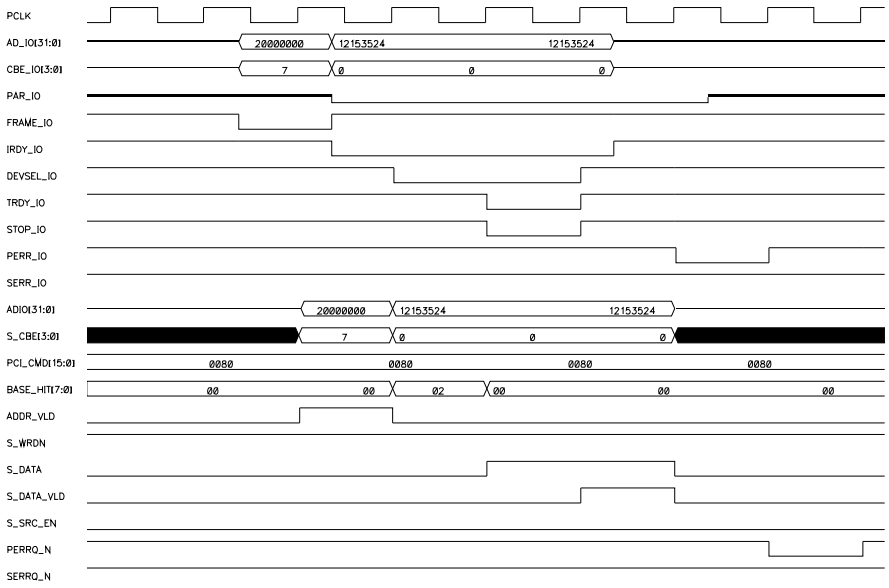


Figure 15-3: **Target Write with Data Parity Error**

The user application may monitor for parity errors on the PCI Bus by sampling PERRQ_N. This signal indicates a data parity error. In addition, the command and status register state is available through the CSR[39:0] bus.

Chapter 16

Design Constraints

This chapter describes how the PCI interface uses timing constraints during processing to ensure that the PCI timing requirements are met when the user design is added. The required PCI performance is guaranteed when the constraints file is used in combination with the appropriate guide file or directed routing constraints.

The following subjects are discussed:

- The function of the constraints in the user constraints file (*.ucf)
- The generation of additional constraints for the user application
- The function of the guide file or directed routing constraints

Caution! Do not modify or remove the constraints in the user constraints file. Customization of this file is allowed, but the modifications must not change the functionality of the supplied constraints. Absolutely do not modify guide files or directed routing constraints. Xilinx guarantees PCI-compliant timing only if the appropriate constraints are used during processing.

Supplied User Constraints

While there are separate constraints files for every part and package combination, the constraints in each perform the same function. These functions are:

- Pinout definition constraints
- Absolute placement constraints
- Timing constraints
- Directed routing constraints

Pinout Definition Constraints

The pinout of the PCI interface is constrained in the user constraints file to match the suggested pin ordering in the *PCI Local Bus Specification* as closely as possible. In addition, the user constraints file prohibits some device pins from being used to reserve them for configuration and boundary scan.

Absolute Placement Constraints

A large number of absolute placement constraints are specified in the user constraints file. These are intended to group critical logic sections together. These constraints are especially important if guide files or directed routing are used.

Timing Constraints

Timing constraints are specified in three distinct steps. First, timename (TNM) attributes are attached to specific instances in the design. Next, timegroups (TIMEGRP) are created from timenames, if necessary. Finally, timespecs (TIMESPEC) are formed by timing specifications between various timenames and timegroups. Each device and package combination has unique timing constraints to ensure full PCI compliance. Do not remove any constraints.

Directed Routing Constraints

Directed routing constraints, when provided, contain routing information for a few highly critical sections of the PCI interface. For this reason, these types of constraints must not be modified. Directed routing constraints are more convenient than guide files because they are included directly in the user constraints file.

Additional User Constraints

Depending on the nature of the customized user design, it may be necessary to modify or add constraints in the user constraints file.

- Change the instance name associated with USER_FFS to match the instance name of the user application.
- If the user application has input and output pads of its own, add a TNM for USER_PADS and specify two additional TIMESPECs for ALL_FFS to USER_PADS and USER_PADS to ALL_FFS.

  Pinout constraints may be added as long as they do not conflict with the pinout established for the PCI interface.
- If the default PERIOD constraint does not adequately constrain the user application, add TNMs, TIMEGRPs, and TIMESPECs as needed to sufficiently constrain the design.

  The PERIOD constraint has the lowest priority, so additional TIMESPECs will "carve" paths out of this default constraint.
- In complex designs, or designs involving multiple clock domains, it may be necessary to entirely remove the PERIOD constraint to effectively process the design.

  In this case, it is important to add at least three additional TIMESPECs to constrain paths within the PCI interface and those paths to and from the portion of the user application that communicates with the PCI interface. These TIMESPECs should have the same value as the PERIOD constraint:

```
TS_UA2PCI = FROM : USER_FFS : TO : PCI_FFS
TS_PCI2UA = FROM : PCI_FFS : TO : USER_FFS
TS_PCI2PCI = FROM : PCI_FFS : TO : PCI_FFS
```

Note: If multiple clock domains are used, the USER_FFS flip-flops should be limited to those that connect to the PCI interface in the PCI clock domain.

Guide Files

Guide files, when provided, contain routing information for a few highly critical sections of the PCI interface. For this reason, these types of constraints must not be modified. For PAR to identify exact matches between the customized user design and the guide file, the

user design must be placed in the supplied wrapper file, which forces net names, instance names, and structure (hierarchy) in the customized user design to match that of the guide file.

XILINX®

Appendix

Protocol Compliance Checklist

The PCI Special Interest Group (PCI-SIG) provides checklists to assist in the design review of ASICs, components, system boards, add-in cards, and systems to verify their compliance with the *PCI Local Bus Specification, Revision 3.0*.

In addition, checklists are used as a requirement to qualify a PCI product for the PCI-SIG Integrator's List by creating a paper trail of testing for PCI compliance. Component, add-in card, and system board vendors may optionally complete the appropriate sections and forward the checklist to their customers if desired.

For the purposes of completing a checklist, Xilinx considers the Initiator/Target for PCI v3.1 interface core the *protocol* portion of a component. This Appendix provides the complete protocol compliance checklist for the PCI v3.0 interface, divided into the following sections:

- Configuration Organization
- Configuration Device Control
- Configuration Device Status
- Configuration Base Addresses
- VGA Devices
- General Component Protocol Checklist (Master)
- General Component Protocol Checklist (Target)
- Component Protocol Checklist for a Master Device
- Component Protocol Checklist for a Target Device

For electrical compliance information, see the respective data sheet for the FPGA family used to generate the physical implementation.

Configuration Organization

Table A-1: **Configuration Organization Checklist**

Item	Description	Pass
CO1.	Does each PCI resource have a configuration space based on the defined 256 byte template, with a predefined 64 byte header and a 192 byte device specific region?	yes_ ✔ no___
CO2.	Do all functions in the device support the Vendor ID, Device ID, Command, Status, Header Type and Class Code fields in the header?	yes_ ✔ no___
CO3.	Is the configuration space available for access at all times?	yes_ ✔ no___
CO4.	Are writes to reserved registers or read only bits completed normally and the data discarded?	yes_ ✔ no___
CO5.	Are reads to reserved or unimplemented registers, or bits, completed normally and a data value of 0 returned?	yes_ ✔ no___
CO6.	Is the vendor ID a number allocated by the PCI-SIG?	yes_ ✔ no___
CO7.	Does the Header Type field have a valid encoding?	yes_ ✔ no___
CO8.	Do multi-byte transactions access the appropriate registers and are the registers in “little endian” order?	yes ✔ no___
CO9.	Are all read only register values within legal ranges?	yes_ ✔ no___
CO10.	Is the class code in compliance with the definition in Appendix D?	yes_ ✔ no___
CO11.	Is the predefined header portion of configuration space accessible as bytes, words, and dwords?	yes_ ✔ no___
CO12.	Is the device a multifunction device?	yes___ no_ ✔
CO13.	If the device is multifunction, are configuration space accesses to unimplemented functions ignored?	yes___ no___ n/a ✔
CO14.	Are Subsystem ID and Subsystem Vendor ID fields are loaded and valid prior to any system software accessing these fields including after boot and resuming from a sleeping state and are not initialized by Expansion ROM code?	yes_ ✔ no___
CO15.	If the function uses extended Capabilities is bit 4 of the status register hardwired to 1 and is the Capabilities List pointer implemented?	yes_ ✔ no___
CO16.	If the function implements Message Signaled Interrupts, is a capability list used to indicate support and does the function not use INTx# pins when MSI is enabled?	yes___ no___ n/a_ ✔

Table A-1: **Configuration Organization Checklist *(Continued)***

Item	Description	Pass
CO17.	If the function supports MSI-X, the MSI-X Capability Structure points to an MSI-X Table structure and an MSI-X Pending Bit Array (PBA) structure. The value in the CAP_ID field (bits 7:0) of the MSI-X Capability Structure is 11h and identifies the function as being MSI-X capable. This field is read only.	yes___ no___ n/a ✔

Indicate either N/A (Not Applicable) or Implemented by checking the appropriate box. Gray areas represent invalid selections. This table should be completed for each function in a multifunction device.

Table A-2: **Function Implementation Checklist**

Location	Name	Required/Optional	N/A	Implemented
00h-01h	Vendor ID	Required		✔
02h-03h	Device ID	Required		✔
04h-05h	Command	Required		✔
06h-07h	Status	Required		✔
08h	Revision ID	Required		✔
09h-0Bh	Class Code	Required		✔
0Ch	Cache Line Size	Required by master devices/functions that can generate Memory Write and Invalidate.	✔	
0Dh	Latency Timer	Required by master devices/functions that can burst more than two data phases.		✔
0Eh	Header Type	If the device is multi-functional, then bit 7 must be set to a 1. The remaining bits are required to have a defined value.		✔
0Fh	Built In Self Test	Optional	✔	
10h-27h	Base Address Registers	One or more required for any address allocation.		✔
28h-2Bh	Cardbus CIS Pointer	Optional		✔
2CH-2Dh	Subsystem Vendor ID	Required		✔
2Eh-2Fh	Subsystem ID	Required		✔
30h-33h	Expansion ROM Base Address Register	Required for devices/functions that have expansion ROM.	✔	
34h	Capabilities Pointer	Optional		✔
35h-3Bh	Reserved	Required		✔

Table A-2: **Function Implementation Checklist *(Continued)***

Location	Name	Required/Optional	N/A	Implemented
3Ch	Interrupt Line	Required by devices/functions that use an interrupt pin.		✔
3Dh	Interrupt Pin	Required by devices/functions that use an interrupt pin.		✔
3Eh	Minimum Grant	Optional		✔
3Fh	Maximum Latency	Optional		✔

Configuration Device Control

Table A-3: **Configuration Device Control Checklist**

Item	Description	Pass
DC1.	When the command register is loaded with a 0000h is the device/function logically disconnected from the PCI bus, with the exception of configuration accesses? (Devices in boot code path are exempt).	yes_✔ no___
DC2.	Is the device/function disabled after the assertion of RST#? (Devices in boot code path are exempt).	yes_✔_ no___

In the following tables for Command and Status Registers, a ✔ (check mark) in the Target or Master columns indicates implementation, and N/A indicates that the bit is not applicable but must return a 0 when read.

Table A-4: **Command and Status Registers Checklist**

Bit	Name	Required/Optional	N/A	Target	Master
0	I/O Space	Required if device/function has registers mapped into I/O space.		✔	N/A
1	Memory Space	Required if device/function responds to memory space accesses.		✔	N/A
2	Bus Master	Required		N/A	✔
3	Special Cycles	Required for devices/functions that can respond to Special Cycles.	✔		N/A
4	Memory Write and Invalidate	Required for devices/functions that generate Memory Write and Invalidate cycles.	✔	N/A	
5	VGA Palette Snoop	Required for VGA or graphical devices/functions that snoop VGA color palette.	✔		N/A
6	Parity Error Response	Required unless exempted per section 3.7.2.		✔	✔
7	Wait Cycle Control	Optional		✔	✔

Table A-4: **Command and Status Registers Checklist *(Continued)***

Bit	Name	Required/Optional	N/A	Target	Master
8	System Error	Required if device/function has SERR# pin.		✔	✔
9	Fast Back-to-Back	Required if Master device/function can support fast back-to-back cycles among different targets.	✔	N/A	
10	Interrupt Disable	Required		✔	✔
11-15	Reserved	Required		✔	✔

Configuration Device Status

Table A-5: **Configuration Device Status Checklist**

Item	Description	Pass
DS1.	Do all implemented read/write bits in the Status reset to 0?	yes_✔ no___
DS2.	Are read/write bits set to a 1 exclusively by the device/function?	yes_✔ no___
DS3.	Are read/write bits reset to a 0 when RST# is asserted?	yes_✔ no___
DS4.	Are read/write bits reset to a 0 by writing a 1 to the bit?	yes_✔ no___

Table A-6: **Device Status Options Checklist**

Bit	Name	Required/Optional	N/A	Target	Master
0-2	Reserved	Required		✔	✔
3	Interrupt Status	Required		✔	✔
4	Capabilities List	Optional		✔	✔
5	66 Mhz Capable	Required for 66Mhz capable devices.		✔	✔
6	Reserved	Required		✔	✔
7	Fast Back-to-Back Capable	Optional	✔		N/A
8	Data Parity Detected	Required		N/A	✔
9-10	DEVSEL# Timing	Required		✔	N/A

Table A-6: **Device Status Options Checklist *(Continued)***

Bit	Name	Required/Optional	N/A	Target	Master
11	Signaled Target Abort	Required for devices/functions that are capable of signaling target abort.		✔	N/A
12	Received Target Abort	Required		N/A	✔
13	Received Master Abort	Required		N/A	✔
14	Signaled System Error	Required for devices/functions that are capable of asserting SERR#.		✔	✔
15	Detected Parity Error	Required unless exempted per section 3.7.2		✔	✔

Configuration Base Addresses

Table A-7: **Configuration Base Address Checklist**

Item	Description	Pass
BA1.	If the device/function uses Expansion ROM, does it implement the Expansion ROM Base Address Register? **Expansion ROM Base Address Register is not implemented.**	yes___ no___ n/a ✔
BA2.	Do all Base Address registers asking for IO space request 256 bytes or less? **Depends on user configuration of interface.**	yes___ no___ n/a_ ✔
BA3.	If the device/function has an Expansion ROM Base Address register, does the memory enable bit in the Command register have precedence over the enable bit in the Expansion ROM base Address register?	yes___ no___ n/a_ ✔
BA4.	Does the device/function use any address space (memory or IO) other than that assigned using Base Address registers? (i.e.; Does the device/function hard-decode any addresses?) Note: If the answer is yes, you must list decoded addresses as explanations at the end of this section.	yes___ no_ ✔
BA5.	Does the device/function decode all 32-bits of IO space? **The upper 28 bits of address are decoded by base address registers.** **The lower 4 bits must be decoded by the user application.**	yes___ no_ ✔
BA6.	If the device/function has an Expansion ROM Base Address register, is the size of the memory space requested 16MB or smaller?	yes___ no___ n/a_ ✔

VGA Devices

Table A-8: **VGA Device Checklist**

Item	Description	Pass
VG1.	Is palette snoop implemented, including bit in Command register?	yes___ no___ n/a_ ✔
VG2.	Is Expansion ROM Base Address register implemented and provide full relocatability of the expansion ROM? (The device must NOT do a hard decode of 0C0000h).	yes___ no___ n/a_ ✔
VG3.	Does the device come up disabled? (Bottom three bits of Command register must be initialized to zero on power-up and RST#).	yes___ no___ n/a ✔
VG4.	Does Class Code field indicate VGA device? (value of 030000h).	yes___ no___ n/a ✔
VG5.	Does the device hard-decode only standard ISA VGA addresses and their aliases? (IO addresses 3B0h through 3BBh, 3C0h through 3DFh, Memory addresses 0A0000h through 0BFFFFh)	yes___ no___ n/a_ ✔

Table A-8: **VGA Device Checklist *(Continued)***

VG6.	Does the device use Base Address Registers to allocate needed space other than standard ISA VGA Addresses? (e.g. for a linear FRAME buffer)	yes___ no___ n/a ✔

General Component Protocol Checklist (Master)

Use the following checklist for general verification of the protocol compliance of the IUT. This checklist applies to all master operations.

Table A-9: **General Component Protocol Checklist (Master)**

Item	Description	Pass
MP1.	All sustained tri-state signals are driven high for one clock before being tri-stated. (2.1)	yes_ ✔ no___
MP2.	IUT always asserts all byte enables during each data phase of a Memory Write and Invalidate cycle. (3.1.1) **Memory Write and Invalidate command not supported.**	yes___ no___ n/a_ ✔
MP3.	IUT always uses Linear Burst Ordering for Memory Write and Invalidate cycles. (3.1.1) **Memory Write and Invalidate command not supported.**	yes___ no___ n/a_ ✔
MP4.	IUT always drives IRDY# when data is valid during a write transaction. (3.2.1)	yes_ ✔ no___
MP5.	IUT only transfers data when both IRDY# and TRDY# are asserted on the same rising clock edge. (3.2.1)	yes_ ✔ no___
MP6.	Once the IUT asserts IRDY# it never changes FRAME# until the current data phase completes. (3.2.1)	yes_ ✔ no___
MP7.	Once the IUT asserts IRDY# it never changes IRDY# until the current data phase completes. (3.2.1)	yes_ ✔ no___
MP8.	IUT never uses reserved burst ordering (AD[1:0] = “01”). (3.2.2)	yes_ ✔ no___
MP9.	IUT never uses reserved burst ordering (AD[1:0] = “11”). (3.2.2)	yes_ ✔ no___
MP10.	IUT always ignores configuration command unless IDSEL is asserted and AD[1:0] are “00”. (3.2.2)	yes_ ✔ no___
MP11.	The IUT's AD lines are driven to stable values during every address and data phase. (3.2.4)	yes_ ✔ no___
MP12.	The IUT's C/BE# output buffers remain enabled from the first clock of the data phase through the end of the transaction. (3.3.1)	yes_ ✔ no___
MP13.	IUT drives C/BE# lines with valid byte enable information during the entire data phase. (3.3.1)	yes_ ✔ no___
MP14.	IUT never deasserts FRAME# unless IRDY# is asserted or will be asserted (3.3.3.1)	yes_ ✔ no___

Table A-9: **General Component Protocol Checklist (Master) *(Continued)***

Item	Description	Pass
MP15.	IUT never deasserts IRDY# until at least one clock after FRAME# is deasserted. (3.3.3.1)	yes ✔ no
MP16.	Once the IUT deasserts FRAME# it never reasserts FRAME# during the same transaction. (3.3.3.1)	yes ✔ no
MP17.	IUT never terminates with master abort once target has asserted DEVSEL#. (3.3.3.1)	yes ✔ no
MP18.	IUT never signals master abort earlier than 5 clocks after FRAME# was first sampled asserted. (3.3.3.1)	yes ✔ no
MP19.	IUT always repeats an access exactly as the original when terminated by retry. (3.3.3.2.2) **The retry process is controlled by logic in the user application.**	yes no n/a ✔
MP20.	IUT never starts cycle unless GNT# is asserted. (3.4.1)	yes ✔ no
MP21.	IUT always tri-states C/BE# and AD within one clock after GNT# negation when bus is idle and FRAME# is negated. (3.4.3)	yes ✔ no
MP22.	IUT always drives C/BE# and AD within eight clocks of GNT# assertion when bus is idle. (3.4.3)	yes ✔ no
MP23.	IUT always asserts IRDY# within eight clocks on all data phases. (3.5.2) **The initial latency is controlled by logic in the user application.**	yes no n/a ✔
MP24.	IUT always begins locked operations with a read transaction. (3.6) **LOCK# function not supported.**	yes no n/a ✔
MP25.	IUT always releases LOCK# when access is terminated by target-abort or master-abort. (3.6) **LOCK# function not supported.**	yes no n/a ✔
MP26.	IUT always deasserts LOCK# for minimum of one idle cycle between consecutive lock operations. (3.6) **LOCK# function not supported.**	yes no n/a ✔
MP27.	IUT always uses Linear Burst Ordering for configuration cycles. (3.7.4)	yes ✓ no
MP28.	IUT always drives PAR within one clock of C/BE# and AD being driven. (3.8.1)	yes ✔ no
MP29.	IUT always drives PAR such that the number of “1”s on AD[31:0],C/BE[3:0], and PAR equals an even number. (3.8.1)	yes ✔ no
MP30.	IUT always drives PERR# (when enabled) active two clocks after data when data parity error is detected. (3.8.2.1)	yes ✔ no
MP31.	IUT always drives PERR# (when enabled) for a minimum of 1 clock for each data phase that a parity error is detected. (3.8.2.1)	yes ✔ no

Table A-9: **General Component Protocol Checklist (Master) *(Continued)***

Item	Description	Pass
MP32.	IUT always holds FRAME# asserted for cycle following the dual address command. (3.10.1)	yes___ no___
	Dual address command not supported.	n/a_ ✔
MP33.	IUT never generates dual address command when upper 32-bits of address are zero. (3.10.1)	yes___ no___
	Dual address command not supported.	n/a_ ✔
MP34.	IUT deasserts REQ# for at least two clocks after having a request terminated with either Retry or Disconnect (3.3.3.2.2)	yes_ ✔ no___

General Component Protocol Checklist (Target)

Use the following checklist for general verification of the protocol compliance of the IUT. This checklist applies to all target operations.

Table A-10: **General Component Protocol Checklist (Target)**

Item	Description	Pass
TP1.	All sustained tri-state signals are driven high for one clock before being tri-stated. (2.1)	yes_ ✔ no___
TP2.	IUT never reports PERR# until it has claimed the cycle and completed a data phase. (2.2.5)	yes_ ✔ no___
TP3.	IUT never aliases reserved commands with other commands. (3.1.1)	yes_ ✔ no___
TP4.	32-bit addressable IUT treats dual address command as reserved. (3.1.1)	yes_ ✔ no___
TP5.	Once IUT has asserted TRDY# it never changes TRDY# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP6.	Once IUT has asserted TRDY# it never changes DEVSEL# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP7.	Once IUT has asserted TRDY# it never changes STOP# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP8.	Once IUT has asserted STOP# it never changes STOP# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP9.	Once IUT has asserted STOP# it never changes TRDY# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP10.	Once IUT has asserted STOP# it never changes DEVSEL# until the data phase completes. (W.1)	yes_ ✔ no___
TP11.	IUT only transfers data when both IRDY# and TRDY# are asserted on the same rising clock edge. (3.2.1)	yes_ ✔ no___
TP12.	IUT always asserts TRDY# when data is valid on a read cycle. (3.2.1)	yes_ ✔ no___

Table A-10: **General Component Protocol Checklist (Target) *(Continued)***

Item	Description	Pass
TP13.	IUT always signals target abort when unable to complete the entire IO access as defined by the byte enables. (3.2.2) **This function is implemented in the user application.**	yes___ no___ n/a_ ✔
TP14.	IUT never responds to reserved encodings. (3.2.2)	yes_ ✔ no___
TP15.	IUT always ignores configuration command unless IDSEL is asserted and AD[1:0] are “00”. (3.2.2)	yes_ ✔ no___
TP16.	IUT always disconnects after the first data phase when reserved burst mode is detected. (3.2.2)	yes_ ✔ no___
TP17.	The IUT's AD lines are driven to stable values during every address and data phase. (3.2.4)	yes_ ✔ no___
TP18.	The IUT's C/BE# output buffers remain enabled from the first clock of the data phase through the end of the transaction. (3.3.1)	yes_ ✔ no___
TP19.	IUT never asserts TRDY# during turnaround cycle on a read. (3.3.1)	yes_ ✔ no___
TP20.	IUT always deasserts TRDY#,STOP#, and DEVSEL# the clock following the completion of the last data phase. (3.3.3.2)	yes_ ✔ no___
TP21.	IUT always signals disconnect when burst crosses resource boundary. (3.3.3.2) **This function is implemented in the user application.**	yes___ no___ n/a_ ✔
TP22.	IUT always deasserts STOP# the cycle immediately following FRAME# being deasserted. (3.3.3.2.1)	yes_ ✔ no___
TP23.	Once the IUT has asserted STOP# it never deasserts STOP# until FRAME# is negated. (3.3.3.2.1)	yes_ ✔ no___
TP24.	IUT always deasserts TRDY# before signaling target-abort. (3.3.3.2.1)	yes_ ✔ no___
TP25.	IUT never deasserts STOP# and continues the transaction. (3.3.3.2.1)	yes_ ✔ no___
TP26.	IUT always completes initial data phase within 16 clocks. (3.5.1.1) **The initial latency is controlled by logic in the user application.**	yes___ no___ n/a_ ✔
TP27.	IUT always locks minimum of 16 bytes. (3.6) **LOCK# function not supported.**	yes___ no___ n/a_ ✔
TP28.	IUT always issues DEVSEL# before any other response. (3.7.1)	yes_ ✔ no___
TP29.	Once IUT has asserted DEVSEL# it never deasserts DEVSEL# until the last data phase has competed except to signal target-abort. (3.7.1)	yes_ ✔ no___

Table A-10: **General Component Protocol Checklist (Target) *(Continued)***

Item	Description	Pass
TP30.	IUT never responds to special cycles. (3.7.2)	yes_✔ no___
TP31.	IUT always drives PAR within one clock of C/BE# and AD being driven. (3.8.1)	yes_✔ no___
TP32.	IUT always drives PAR such that the number of "1"s on AD[31:0],C/BE#[3:0], and PAR equals an even number. (3.8.1)	yes_✔ no___
TP33.	If IUT is accessed during initialization time (time from RST# is deasserted and 2**25 clocks later), IUT responds to the access by (3.5.1.1): Completing the initial data phase within 16 clocks Ignoring the access Claim the access and hold in wait states Claim the access and terminate with Retry **This function is implemented in the user application. One or more of the above options may be implemented.**	yes_✔ no___
TP34.	After terminating a memory write transaction with retry, IUT will be ready to complete at least one data phase of a memory write within 334 clocks for 33 MHz devices and 668 clocks for 66 MHz devices. **This function is implemented in the user application.**	yes___ no___ n/a_✔

Component Protocol Checklist for a Master Device

Test Scenario: 1.1. PCI Device Speed (as indicated by DEVSEL#) Tests

If IUT does not implement memory transactions, mark 1 through 10 N/A. If IUT supports both read and write transactions *do not* mark 1 through 10 N/A.

Table A-11: **PCI Device Speed Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	
9	Master abort bit set after write to slower than subtractive memory slave.	✔	

Table A-11: **PCI Device Speed Checklist #1**

Test	Description	Pass	N/A
10	Master abort bit set after read from slower than subtractive memory slave.	✔	

If IUT does not implement I/O transactions, mark 11 through 20 N/A. Else if IUT supports both read and write transactions *do not* mark 11 through 20 N/A.

Table A-12: **PCI Device Speed Checklist #2**

Test	Description	Pass	N/A
11	Data transfer after write to fast I/O slave.	✔	
12	Data transfer after read from fast I/O slave.	✔	
13	Data transfer after write to medium I/O slave.	✔	
14	Data transfer after read from medium I/O slave.	✔	
15	Data transfer after write to slow I/O slave.	✔	
16	Data transfer after read from slow I/O slave.	✔	
17	Data transfer after write to subtractive I/O slave.	✔	
18	Data transfer after read from subtractive I/O slave.	✔	
19	Master abort bit set after write to slower than subtractive I/O slave.	✔	
20	Master abort bit set after read from slower than subtractive I/O slave.	✔	

If IUT does not implement Configuration transactions, mark 21 through 30 N/A. Else if IUT supports both read and write transactions *do not* mark 21 through 30 N/A.

Table A-13: **PCI Device Speed Checklist #3**

Test	Description	Pass	N/A
21	Data transfer after write to fast config slave.	✔	
22	Data transfer after read from fast config slave.	✔	
23	Data transfer after write to medium config slave.	✔	
24	Data transfer after read from medium config slave.	✔	
25	Data transfer after write to slow config slave.	✔	
26	Data transfer after read from slow config slave.	✔	
27	Data transfer after write to subtractive config slave.	✔	
28	Data transfer after read from subtractive config slave.	✔	
29	Master abort bit set after write to slower than subtractive config slave.	✔	

Table A-13: **PCI Device Speed Checklist #3**

Test	Description	Pass	N/A
30	Master abort bit set after read from slower than subtractive config slave.	✔	

If IUT does not implement Interrupt transactions, mark 31 through 35 N/A.

Table A-14: **PCI Device Speed Checklist #4**

Test	Description	Pass	N/A
31	Data transfer after interrupt from fast memory slave.	✔	
32	Data transfer after interrupt from medium memory slave.	✔	
33	Data transfer after interrupt from slow memory slave.	✔	
34	Data transfer after interrupt from subtractive memory slave.	✔	
35	Master abort bit set for interrupt from slower than subtractive memory slave.	✔	

If IUT does not implement Special transactions, mark 36 and 37 N/A.

Table A-15: **PCI Device Speed Checklist #5**

Test	Description	Pass	N/A
36	Data transfer after Special transaction to slave.	✔	
37	Master abort bit is not set after Special transaction.	✔	

Table A-16: **Explanations for PCI Device Speed**

Explanations

Use this area to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.1 Checklist

Test Scenario: 1.2. PCI Bus Target Abort Cycles

If IUT does not implement memory transactions, mark 1 through 16 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 16 N/A.

Table A-17: **PCI Bus Target Abort Cycles Checklist #1**

Test	Description	Pass	N/A
1	Target Abort bit set after write to fast memory slave.	✔	
2	IUT does not repeat the write transaction.	✔	
3	IUT's Target Abort bit set after read from fast memory slave.	✔	
4	IUT does not repeat the read transaction.	✔	
5	Target Abort bit set after write to medium memory slave.	✔	
6	IUT does not repeat the write transaction.	✔	
7	IUT's Target Abort bit set after read from medium memory slave.	✔	
8	IUT does not repeat the read transaction.	✔	
9	Target Abort bit set after write to slow memory slave.	✔	
10	IUT does not repeat the write transaction.	✔	
11	IUT's Target Abort bit set after read from slow memory slave.	✔	
12	IUT does not repeat the read transaction.	✔	
13	Target Abort bit set after write to subtractive memory slave.	✔	
14	IUT does not repeat the write transaction.	✔	
15	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
16	IUT does not repeat the read transaction.	✔	

If IUT does not implement I/O transactions, mark 17 through 32 N/A. Else if IUT supports both read and write transactions *do not* mark 17 through 32 N/A.

Table A-18: **PCI Bus Target Abort Cycles Checklist #2**

Test	Description	Pass	N/A
17	Target Abort bit set after write to fast I/O slave.	✔	
18	IUT does not repeat the write transaction.	✔	
19	IUT's Target Abort bit set after read from fast I/O slave.	✔	
20	IUT does not repeat the read transaction.	✔	
21	Target Abort bit set after write to medium I/O slave.	✔	
22	IUT does not repeat the write transaction.	✔	
23	IUT's Target Abort bit set after read from medium I/O slave.	✔	

Table A-18: **PCI Bus Target Abort Cycles Checklist #2**

Test	Description	Pass	N/A
24	IUT does not repeat the read transaction.	✔	
25	Target Abort bit set after write to slow I/O slave.	✔	
26	IUT does not repeat the write transaction.	✔	
27	IUT's Target Abort bit set after read from slow I/O slave.	✔	
28	IUT does not repeat the read transaction.	✔	
29	Target Abort bit set after write to subtractive I/O slave.	✔	
30	IUT does not repeat the write transaction.	✔	
31	IUT's Target Abort bit set after read from subtractive I/O slave.	✔	
32	IUT does not repeat the read transaction.	✔	

If IUT does not implement configuration transactions, mark 33 through 48 N/A. Else if IUT supports both read and write transactions *do not* mark 33 through 48 N/A.

Table A-19: **PCI Bus Target Abort Cycles Checklist #3**

Test	Description	Pass	N/A
33	Target Abort bit set after write to fast config slave	✔	
34	IUT does not repeat the write transaction.	✔	
35	IUT's Target Abort bit set after read from fast config slave.	✔	
36	IUT does not repeat the read transaction.	✔	
37	Target Abort bit set after write to medium config slave.	✔	
38	IUT does not repeat the write transaction.	✔	
39	IUT's Target Abort bit set after read from medium config slave.	✔	
40	IUT does not repeat the read transaction.	✔	
41	Target Abort bit set after write to slow config slave.	✔	
42	IUT does not repeat the write transaction.	✔	
43	IUT's Target Abort bit set after read from slow config slave.	✔	
44	IUT does not repeat the read transaction.	✔	
45	Target Abort bit set after write to subtractive config slave.	✔	
46	IUT does not repeat the write transaction.	✔	
47	IUT's Target Abort bit set after read from subtractive config slave.	✔	
48	IUT does not repeat the read transaction.	✔	

If IUT does not implement interrupt transactions, mark 49 through 56 N/A.

Table A-20: **PCI Bus Target Abort Cycles Checklist #4**

Test	Description	Pass	N/A
49	IUT's Target Abort bit set after interrupt acknowledge from fast slave.	✔	
50	IUT does not repeat the interrupt acknowledge transaction.	✔	
51	IUT's Target Abort bit set after interrupt acknowledge from medium slave.	✔	
52	IUT does not repeat the interrupt acknowledge transaction.	✔	
53	IUT's Target Abort bit set after interrupt acknowledge from slow slave.	✔	
54	IUT does not repeat the interrupt acknowledge transaction.	✔	
55	IUT's Target Abort bit set after interrupt acknowledge from subtractive slave.	✔	
56	IUT does not repeat the interrupt acknowledge transaction.	✔	

Table A-21: **Explanations for PCI Target Abort Cycles**

Explanations
The user application is responsible for not repeating terminated transactions.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.2 Checklist

Test Scenario: 1.3. PCI Bus Target Retry Cycles

If IUT does not implement memory transactions, mark 1 through 8 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 8 N/A.

Table A-22: **PCI Bus Target Retry Cycles Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	

Table A-22: **PCI Bus Target Retry Cycles Checklist #1**

Test	Description	Pass	N/A
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions, mark 9 through 16 N/A. Else if IUT supports both read and write transactions *do not* mark 9 through 16 N/A.

Table A-23: **PCI Bus Target Retry Cycles Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions, mark 17 through 24 N/A. Else if IUT supports both read and write transactions *do not* mark 17 through 24 N/A.

Table A-24: **PCI Bus Target Retry Cycles Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config slave.	✔	
18	Data transfer after read from fast config slave.	✔	
19	Data transfer after write to medium config slave.	✔	
20	Data transfer after read from medium config slave.	✔	
21	Data transfer after write to slow config slave.	✔	
22	Data transfer after read from slow config slave.	✔	
23	Data transfer after write to subtractive config slave.	✔	
24	Data transfer after read from subtractive config slave.	✔	

If IUT does not implement interrupt transactions, mark 25 through 28 N/A else *do not* mark 25 through 28 N/A.

Table A-25: **PCI Bus Target Retry Cycles Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after interrupt acknowledge from fast slave.	✔	
26	Data transfer after interrupt acknowledge from medium slave.	✔	
27	Data transfer after interrupt acknowledge from slow slave.	✔	
28	Data transfer after interrupt acknowledge from subtractive slave.	✔	

Table A-26: **Explanations for PCI Bus Target Retry Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.3 Checklist

Test Scenario: 1.4. PCI Bus Single Data Phase Disconnect Cycles

If IUT does not implement memory transactions, mark 1 through 8 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 8 N/A.

Table A-27: **PCI Bus Single Data Phase Disconnect Cycles Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions, mark 9 through 16 N/A. Else if IUT supports both read and write transactions *do not* mark 9 through 16 N/A.

Table A-28: **PCI Bus Single Data Phase Disconnect Cycles Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions, mark 17 through 24 N/A. Else if IUT supports both read and write transactions *do not* mark 17 through 24 N/A.

Table A-29: **PCI Bus Single Data Phase Disconnect Cycles Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config slave.	✔	
18	Data transfer after read from fast config slave.	✔	
19	Data transfer after write to medium config slave.	✔	
20	Data transfer after read from medium config slave.	✔	
21	Data transfer after write to slow config slave.	✔	
22	Data transfer after read from slow config slave.	✔	
23	Data transfer after write to subtractive config slave.	✔	
24	Data transfer after read from subtractive config slave.	✔	

If IUT does not implement interrupt transactions, mark 25 through 28 N/A else *do not* mark 25 through 28 N/A.

Table A-30: **PCI Bus Single Data Phase Disconnect Cycles Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after interrupt acknowledge from fast slave.	✔	
26	Data transfer after interrupt acknowledge from medium slave.	✔	
27	Data transfer after interrupt acknowledge from slow slave.	✔	
28	Data transfer after interrupt acknowledge from subtractive slave.	✔	

Table A-31: **Explanation for PCI Bus Single Data Phase Disconnect Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.4 Checklist

Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles

If IUT does not implement memory transactions, mark 1 through 16 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 16 N/A.

Table A-32: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #1**

Test	Description	Pass	N/A
1	Target Abort bit set after write to fast memory slave.	✔	
2	IUT does not repeat the write transaction.	✔	
3	IUT's Target Abort bit set after read from fast memory slave.	✔	
4	IUT does not repeat the read transaction.	✔	
5	Target Abort bit set after write to medium memory slave.	✔	
6	IUT does not repeat the write transaction.	✔	
7	IUT's Target Abort bit set after read from medium memory slave.	✔	
8	IUT does not repeat the read transaction.	✔	
9	Target Abort bit set after write to slow memory slave.	✔	
10	IUT does not repeat the write transaction.	✔	
11	IUT's Target Abort bit set after read from slow memory slave.	✔	
12	IUT does not repeat the read transaction.	✔	
13	Target Abort bit set after write to subtractive memory slave.	✔	
14	IUT does not repeat the write transaction.	✔	
15	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
16	IUT does not repeat the read transaction.	✔	

If IUT does not implement dual address transactions, mark 17 through 32 N/A. Else if IUT supports both read and write dual address transactions *do not* mark 17 through 32 N/A.

Table A-33: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #2**

Test	Description	Pass	N/A
17	Target Abort bit set after write to fast memory slave.		✔
18	IUT does not repeat the write transaction.		✔
19	IUT's Target Abort bit set after read from fast memory slave.		✔
20	IUT does not repeat the read transaction.		✔
21	Target Abort bit set after write to medium memory slave.		✔
22	IUT does not repeat the write transaction.		✔
23	IUT's Target Abort bit set after read from medium memory slave.		✔
24	IUT does not repeat the read transaction.		✔
25	Target Abort bit set after write to slow memory slave.		✔
26	IUT does not repeat the write transaction.		✔
27	IUT's Target Abort bit set after read from slow memory slave.		✔
28	IUT does not repeat the read transaction.		✔
29	Target Abort bit set after write to subtractive memory slave.		✔
30	IUT does not repeat the write transaction.		✔
31	IUT's Target Abort bit set after read from subtractive memory slave.		✔
32	IUT does not repeat the read transaction.		✔

If IUT does not implement configuration transactions, mark 33 through 48 N/A. Else if IUT supports both read and write transactions *do not* mark 33 through 48 N/A.

Table A-34: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #3**

Test	Description	Pass	N/A
33	Target Abort bit set after write to fast config. slave.	✔	
34	IUT does not repeat the write transaction.	✔	
35	IUT's Target Abort bit set after read from fast config. slave.	✔	
36	IUT does not repeat the read transaction.	✔	
37	Target Abort bit set after write to medium config. slave.	✔	
38	IUT does not repeat the write transaction.	✔	
39	IUT's Target Abort bit set after read from medium config. slave.	✔	
40	IUT does not repeat the read transaction.	✔	
41	Target Abort bit set after write to slow config. slave.	✔	

Table A-34: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #3 *(Continued)***

Test	Description	Pass	N/A
42	IUT does not repeat the write transaction.	✔	
43	IUT's Target Abort bit set after read from slow config. slave.	✔	
44	IUT does not repeat the read transaction.	✔	
45	Target Abort bit set after write to subtractive config. slave.	✔	
46	IUT does not repeat the write transaction.	✔	
47	IUT's Target Abort bit set after read from subtractive config. slave.	✔	
48	IUT does not repeat the read transaction.	✔	

If IUT does not implement memory read multiple transactions, mark 49 through 56 N/A. Else *do not* mark 49 through 56 N/A.

Table A-35: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #4**

Test	Description	Pass	N/A
49	IUT's Target Abort bit set after read from fast memory slave.	✔	
50	IUT does not repeat the read transaction.	✔	
51	IUT's Target Abort bit set after read from medium memory slave.	✔	
52	IUT does not repeat the read transaction.	✔	
53	IUT's Target Abort bit set after read from slow memory slave.	✔	
54	IUT does not repeat the read transaction.	✔	
55	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
56	IUT does not repeat the read transaction.	✔	

If IUT does not implement memory read line transactions, mark 57 through 64 N/A. Else *do not* mark 57 through 64 N/A.

Table A-36: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #5**

Test	Description	Pass	N/A
57	IUT's Target Abort bit set after read from fast memory slave.	✔	
58	IUT does not repeat the read transaction.	✔	
59	IUT's Target Abort bit set after read from medium memory slave.	✔	
60	IUT does not repeat the read transaction.	✔	
61	IUT's Target Abort bit set after read from slow memory slave.	✔	
62	IUT does not repeat the read transaction.	✔	
63	IUT's Target Abort bit set after read from subtractive memory slave.	✔	

Table A-36: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #5**

Test	Description	Pass	N/A
64	IUT does not repeat the read transaction.	✔	

If IUT does not implement memory write and invalidate transactions, mark 65 through 72 N/A.
Else *do not* mark 65 through 72 N/A.

Table A-37: **PCI Bus Multi-Data Phase Target Abort Cycles Checklist #6**

Test	Description	Pass	N/A
65	Target Abort bit set after write to fast memory slave.		✔
66	IUT does not repeat the write transaction.		✔
67	Target Abort bit set after write to medium memory slave.		✔
68	IUT does not repeat the write transaction.		✔
69	Target Abort bit set after write to slow memory slave.		✔
70	IUT does not repeat the write transaction.		✔
71	IUT's Target Abort bit set after read from slow memory slave.		✔
72	IUT does not repeat the write transaction.		✔

Table A-38: **Explanations for PCI Bus Multi-Data Phase Target Abort Cycles**

Explanations
The user application is responsible for not repeating terminated transactions.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.5 Checklist

Test Scenario: 1.6. PCI Bus Multi-Data Phase Retry Cycles

If IUT does not implement memory transactions, mark 1 through 8 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 8 N/A.

Table A-39: **PCI Bus Multi-Data Phase Retry Cycles Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	

Table A-39: **PCI Bus Multi-Data Phase Retry Cycles Checklist #1**

Test	Description	Pass	N/A
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions, mark 9 through 16 N/A. Else if IUT supports both read and write transactions *do not* mark 9 through 16 N/A.

Table A-40: **PCI Bus Multi-Data Phase Retry Cycles Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions, mark 17 through 24 N/A. Else if IUT supports both read and write transactions *do not* mark 17 through 24 N/A.

Table A-41: **PCI Bus Multi-Data Phase Retry Cycles Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config. slave.	✔	
18	Data transfer after read from fast config. slave.	✔	
19	Data transfer after write to medium config. slave.	✔	
20	Data transfer after read from medium config. slave.	✔	
21	Data transfer after write to slow config. slave.	✔	
22	Data transfer after read from slow config. slave.	✔	
23	Data transfer after write to subtractive config. slave.	✔	

Table A-41: **PCI Bus Multi-Data Phase Retry Cycles Checklist #3**

Test	Description	Pass	N/A
24	Data transfer after read from subtractive config. slave.	✔	

If IUT does not implement memory read multiple transactions, mark 25 through 28 N/A else do not mark 25 through 28 N/A.

Table A-42: **PCI Bus Multi-Data Phase Retry Cycles Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after memory read multiple from fast slave.	✔	
26	Data transfer after memory read multiple from medium slave.	✔	
27	Data transfer after memory read multiple from slow slave.	✔	
28	Data transfer after memory read multiple from subtractive slave.	✔	

If IUT does not implement memory read line transactions, mark 29 through 32 N/A. Else do not mark 29 through 32 N/A.

Table A-43: **PCI Bus Multi-Data Phase Retry Cycles Checklist #5**

Test	Description	Pass	N/A
29	Data transfer after memory read line from fast slave.	✔	
30	Data transfer after memory read line from medium slave.	✔	
31	Data transfer after memory read line from slow slave.	✔	
32	Data transfer after memory read line from subtractive slave.	✔	

If IUT does not implement memory write and invalidate transactions, mark 33 through 36 N/A. Else do not mark 33 through 36 N/A.

Table A-44: **PCI Bus Multi-Data Phase Retry Cycles Checklist #6**

Test	Description	Pass	N/A
33	Data transfer after memory write and invalidate to fast slave.		✔
34	Data transfer after memory write and invalidate to medium slave.		✔
35	Data transfer after memory write and invalidate to slow slave.		✔
36	Data transfer after memory write and invalidate to subtractive slave.		✔

Table A-45: **Explanations for PCI Bus Multi-Data Phase Retry Cycles**

Explanations

Table A-45: **Explanations for PCI Bus Multi-Data Phase Retry Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.6 Checklist

Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles

If IUT does not implement multi-data phase memory transactions, mark 1 through 8 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 8 N/A.

Table A-46: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions, mark 9 through 16 N/A. Else if IUT supports both read and write transactions *do not* mark 9 through 16 N/A.

Table A-47: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	

Table A-47: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #2**

Test	Description	Pass	N/A
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions, mark 17 through 24 N/A. Else if IUT supports both read and write transactions *do not* mark 17 through 24 N/A.

Table A-48: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config. slave.	✔	
18	Data transfer after read from fast config. slave.	✔	
19	Data transfer after write to medium config. slave.	✔	
20	Data transfer after read from medium config. slave.	✔	
21	Data transfer after write to slow config. slave.	✔	
22	Data transfer after read from slow config. slave.	✔	
23	Data transfer after write to subtractive config. slave.	✔	
24	Data transfer after read from subtractive config. slave.	✔	

If IUT does not implement memory read multiple transactions, mark 25 through 28 N/A else do not mark 25 through 28 N/A.

Table A-49: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after memory read multiple from fast slave.	✔	
26	Data transfer after memory read multiple from medium slave.	✔	
27	Data transfer after memory read multiple from slow slave.	✔	
28	Data transfer after memory read multiple from subtractive slave.	✔	

If IUT does not implement memory read line transactions, mark 29 through 32 N/A else do not mark 29 through 32 N/A.

Table A-50: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #4**

Test	Description	Pass	N/A
29	Data transfer after memory read line from fast slave.	✔	
30	Data transfer after memory read line from medium slave.	✔	
31	Data transfer after memory read line from slow slave.	✔	
32	Data transfer after memory read line from subtractive slave.	✔	

If IUT does not implement memory write and invalidate transactions, mark 33 through 36 N/A else do not mark 33 through 36 N/A.

Table A-51: **PCI Bus Multi-Data Phase Disconnect Cycles Checklist #5**

Test	Description	Pass	N/A
33	Data transfer after memory write and invalidate to fast slave.		✔
34	Data transfer after memory write and invalidate to medium slave.		✔
35	Data transfer after memory write and invalidate to slow slave.		✔
36	Data transfer after memory write and invalidate to subtractive slave.		✔

Table A-52: **Explanation for PCI Bus Multi-Data Phase Disconnect Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.7 Checklist

Test Scenario: 1.8. Multi-data Phase & TRDY# Cycles

If IUT does not implement multi-data phase memory transactions, mark 1 through 12 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 12 N/A.

Table A-53: **Multi-data Phase & TRDY# Cycles Checklist #1**

Test	Description	Pass	N/A
1	Verify that data is written to primary target when TRDY# is released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
2	Verify that data is read from primary target when TRDY# is released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
3	Verify that data is written to primary target when TRDY# is released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
4	Verify that data is read from primary target when TRDY# is released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	

Table A-53: **Multi-data Phase & TRDY# Cycles Checklist #1 *(Continued)***

Test	Description	Pass	N/A
5	Verify that data is written to primary target when TRDY# is released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
6	Verify that data is read from primary target when TRDY# is released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
7	Verify that data is written to primary target when TRDY# is released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
8	Verify that data is read from primary target when TRDY# is released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
9	Verify that data is written to primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
10	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
11	Verify that data is written to primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	
12	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement dual address transactions, mark 13 through 24 N/A. Else if IUT supports both read and write transactions *do not* mark 13 through 24 N/A.

Table A-54: **Multi-Data Phase & TRDY# Cycles Checklist #2**

Test	Description	Pass	N/A
13	Verify that data is written to primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#		✔
14	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#		✔
15	Verify that data is written to primary target when TRDY# released after 4th rising clock edge and asserted on 5th rising clock edge after FRAME#		✔
16	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 5th rising clock edge after FRAME#		✔
17	Verify that data is written to primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#		✔

Table A-54: **Multi-Data Phase & TRDY# Cycles Checklist #2 *(Continued)***

Test	Description	Pass	N/A
18	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#		✔
19	Verify that data is written to primary target when TRDY# released after 5th rising clock edge and asserted on 7th rising clock edge after FRAME#		✔
20	Verify that data is read from primary target when TRDY# released after 5th rising clock edge and asserted on 7th rising clock edge after FRAME#		✔
21	Verify that data is written to primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#		✔
22	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#		✔
23	Verify that data is written to primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#		✔
24	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#		✔

If IUT does not implement memory read multiple transactions, mark 25 through 30 N/A else do not mark 25 through 30 N/A.

Table A-55: **Multi-Data Phase & TRDY# Cycles Checklist #3**

Test	Description	Pass	N/A
25	Verify that data is read from primary target when TRDY# released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
26	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
27	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
28	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
29	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
30	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement memory read line transactions, mark 31 through 36 N/A else do not mark 31 through 36 N/A.

Table A-56: **Multi-Data Phase & TRDY# Cycles Checklist #4**

Test	Description	Pass	N/A
31	Verify that data is read from primary target when TRDY# released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
32	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
33	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
34	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
35	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
36	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement memory write and invalidate transactions, mark 37 through 42 N/A else do not mark 37 through 42 N/A.

Table A-57: **Multi-Data Phase & TRDY# Cycles Checklist #5**

Test	Description	Pass	N/A
37	Verify that data is written to primary target when TRDY# released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#		✔
38	Verify that data is written to primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#		✔
39	Verify that data is written to primary target when TRDY# released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#		✔
40	Verify that data is written to primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#		✔
41	Verify that data is written to primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#		✔
42	Verify that data is written to primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#		✔

Table A-58: **Explanations for Multi-Data Phase & TRDY# Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.8 Checklist

Test Scenario: 1.9. Bus Data Parity Error Single Cycles

If IUT is exempted from reporting parity errors per the exemptions listed in subsection 3.7.2 of the specification then mark the following N/A. If IUT does not implement memory transactions, mark 1 through 3 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 3 N/A

Table A-59: **Bus Data Parity Error Single Cycles Checklist #1**

Test	Description	Pass	N/A
1	Verify the IUT sets Data Parity Error Detected bit when Primary Target asserts PERR# on IUT Memory Write	✔	
2	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT Memory Read	✔	
3	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Memory read	✔	

If IUT does not implement I/O transactions, mark 4 through 6 N/A. Else if IUT supports both read and write transactions *do not* mark 4 through 6 N/A.

Table A-60: **Bus Data Parity Error Single Cycles Checklist #2**

Test	Description	Pass	N/A
4	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT I/O Write	✔	
5	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT I/O Read	✔	
6	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT I/O read	✔	

If IUT does not implement configuration transactions, mark 7 through 9 N/A. Else if IUT supports both read and write transactions *do not* mark 7 through 9 N/A.

Table A-61: **Bus Data Parity Error Single Cycles Checklist #3**

Test	Description	Pass	N/A
7	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT Config Write	✔	
8	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT Config Read	✔	
9	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Config read	✔	

Table A-62: **Explanations for Bus Data Parity Error Single Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.9 Checklist

Test Scenario: 1.10. Bus Data Parity Error Multi-data Phase Cycles

If IUT is exempted from reporting parity errors per the exemptions listed in subsection 3.7.2 of the specification then mark the following N/A. If IUT does not implement memory transactions, mark 1 through 3 N/A. Else if IUT supports both read and write transactions *do not* mark 1 through 3 N/A.

Table A-63: **Bus Data Parity Error Multi-data Phase Cycles Checklist #1**

Test	Description	Pass	N/A
1	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT multi data phase Memory Write	✔	
2	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT multi data phase Memory Read	✔	
3	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Memory multi data phase read	✔	

If IUT does not implement dual address transactions, mark 4 through 6 N/A. Else if IUT supports both read and write transactions *do not* mark 4 through 6 N/A.

Table A-64: **Bus Data Parity Error Multi-data Phase Cycles Checklist #2**

Test	Description	Pass	N/A
4	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT dual address multi data phase Write		✔
5	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT dual address multi data phase Read		✔
6	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT dual address multi data phase read		✔

If IUT does not implement configuration transactions, mark 7 through 9 N/A. Else if IUT supports both read and write transactions *do not* mark 7 through 9 N/A.

Table A-65: **Bus Data Parity Error Multi-data Phase Cycles Checklist #3**

Test	Description	Pass	N/A
7	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT Config multi-data phase Write	✔	
8	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT Config multi-data phase Read	✔	
9	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Config multi-data phase read	✔	

If IUT does not implement memory read multiple transactions, mark 10 through 11 N/A. Else *do not* mark 10 through 11 N/A.

Table A-66: **Bus Data Parity Error Multi-data Phase Cycles Checklist #4**

Test	Description	Pass	N/A
10	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT mem. rd. multiple data phase.	✔	
11	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT mem. rd. multiple data phase.	✔	

If IUT does not implement memory read line transactions, mark 12 through 13 N/A. Else *do not* mark 12 through 13 N/A.

Table A-67: **Bus Data Parity Error Multi-data Phase Cycles Checklist #5**

Test	Description	Pass	N/A
12	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT mem. rd. line data phase.	✔	
13	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT mem. rd. line data phase.	✔	

If IUT does not implement memory write and invalidate transactions, mark 14 N/A. Else *do not* mark 14 N/A.

Table A-68: **Bus Data Parity Error Multi-data Phase Cycles Checklist #6**

Test	Description	Pass	N/A
14	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT Memory Write and Invalidate data phase.		✔

Table A-69: **Explanations for Bus Data Parity Error Multi-data Phase Cycles**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.10 Checklist

Test Scenario: 1.11. Bus Master Timeout

If IUT does not support dual address cycles, mark 7 and 8 N/A

Table A-70: **Bus Master Timeout Checklist #1**

Test	Description	Pass	N/A
1	Memory write transaction terminates before 4 data phases completed	✔	
2	Memory read transaction terminates before 4 data phases completed	✔	
3	Config write transaction terminates before 4 data phases completed	✔	
4	Config read transaction terminates before 4 data phases completed	✔	
5	Memory read multiple transaction terminates before 4 data phases	✔	
6	Memory read line transaction terminates before 4 data phases	✔	
7	Dual Address write transaction terminates before 4 data phases completed		✔
8	Dual Address read transaction terminates before 4 data phases completed		✔

If IUT does not support cache coherent transactions, mark 9 N/A. Else if IUT supports cache coherent transactions and has implemented the configuration register that specifies cache line length *do not* mark 9 N/A.

Table A-71: **Bus Master Timeout Checklist #2**

Test	Description	Pass	N/A
9	Memory write invalidate terminates on line boundary		✔

Table A-72: **Explanations for Bus Master Timeout Checklist**

Explanations
The default configuration of the interface will timeout one cycle later than required by the specification.
The behavior may be made compliant by changing the core configuration.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.11 Checklist

Test Scenario: 1.12. PCI Bus Master Parking

Verify that the IUT is able to drive PCI bus to stable conditions if it is idle and GNT# is asserted.

Table A-73: **PCI Bus Master Parking Checklist #1**

Test	Description	Pass	N/A
1	IUT drives AD[31::00] to stable values within eight PCI Clocks of GNT#.	✔	
2	IUT drives C/BE#[3::0] to stable values within eight PCI Clocks of GNT#.	✔	
3	IUT drives PAR one clock cycle after IUT drives AD[31:0]	✔	

Verify that the IUT will Tri-state the bus when GNT# is not asserted.

Table A-74: **PCI Bus Master Parking Checklist #2**

Test	Description	Pass	N/A
4	IUT Tri-states AD[31:00] and C/BE[3:0] and PAR when GNT# is released.	✔	

Table A-75: **Explanations for PCI Bus Master Parking**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.12 Checklist

Test Scenario: 1.13. PCI Bus Master Arbitration

Verify that the IUT is able to complete bus transaction when GNT# is deasserted coincident with FRAME# asserted.

Table A-76: **PCI Bus Master Arbitration Checklist**

Test	Description	Pass	N/A
1	IUT completes transaction when deasserting GNT# is coincident with asserting FRAME#.	✔	

Table A-77: **Explanations for PCI Bus Master Arbitration**

Explanations
The interface requires two cycles of GNT# before FRAME# is asserted as the interface uses address stepping.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.13 Checklist

Component Protocol Checklist for a Target Device

Test Scenario: 2.1. Target Reception of an Interrupt Cycle

If IUT does not respond to Interrupt Acknowledge bus transactions, mark 1 and 2 N/A.

Table A-78: **Target Reception of an Interrupt Cycle Checklist**

Test	Description	Pass	N/A
1	IUT generates Interrupts when programmed	✔	
2	IUT clears Interrupts when serviced (may include driver specific actions)	✔	

Test Scenario: 2.2. Target Reception of Special Cycle

If IUT does not implement Special Cycles, mark 1 and 2 N/A.

Table A-79: **Target Reception of Special Cycle Checklist**

Test	Description	Pass	N/A
1	No DEVSEL# Assertion by IUT after Special Cycle		✔
2	IUT receives encoded special cycle		✔

Test Scenario: 2.3. Target Detection of Address and Data Parity Error for Special Cycle

If IUT does not implement Special Cycles mark 2 N/A.

Table A-80: **Target Detection of Address and Data Parity Error for Special Cycle Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR#	✔	
2	IUT reports data parity error via SERR#		✔
3	IUT keeps SERR# active for at least one clock	✔	

Test Scenario: 2.4. Target Reception of I/O Cycles With Legal and Illegal Byte Enables

IF IUT does not support I/O cycles mark 1 through 4 N/A or if IUT claims all 32 bits during an I/O cycle mark 1 and 2 N/A

Table A-81: **Target Reception of I/O Cycles With Legal and Illegal Byte Enables Checklist**

Test	Description	Pass	N/A
1	IUT asserts TRDY# following 2nd rising edge from FRAME on all legal BE		✔
2	IUT terminates with target abort for each illegal BE		✔

Table A-81: **Target Reception of I/O Cycles With Legal and Illegal Byte Enables Checklist**

3	IUT asserts STOP#		✔
4	IUT deasserts STOP# after FRAME# deassertion		✔

Test Scenario: 2.5. Target Ignores Reserved Commands

Table A-82: **Target Ignores Reserved Commands Checklist**

Test	Description	Pass	N/A
1	IUT does not respond to reserved commands	✔	
2	Initiator detects master abort for each transfer	✔	
3	IUT does not respond to 64bit cycle (dual address)	✔	

Test Scenario: 2.6. Target Receives Configuration Cycles

If IUT does not support configuration type 1 mark 3 NA.

Table A-83: **Target Receives Configuration Cycles**

Test	Description	Pass	N/A
1	IUT responds to all configuration cycles type 0 read/write cycles appropriately	✔	
2	IUT does not respond to configuration cycles type 0 with IDSEL inactive	✔	
3	IUT responds to all configuration cycles type 1 read/write cycles appropriately		✔
4	IUT responds to all configuration cycles type 0 read/write cycles appropriately	✔	
5	IUT does not respond (master abort) on illegal configuration cycle types	✔	

Test Scenario: 2.7. Target Receives I/O Cycles With Address and Data Parity Errors

If IUT does not support I/O cycles mark 1 through 2 N/A.

Table A-84: **Target Receives I/O Cycles With Address and Data Parity Errors Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR# during I/O read/write cycles	✔	
2	IUT reports data parity error via PERR# during I/O write cycles	✔	

Test Scenario: 2.8. Target Gets Config Cycles With Address and Data Parity Errors

Table A-85: **Target Gets Config Cycles With Address and Data Parity Errors Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR# during configuration read/write cycles	✔	
2	IUT reports data parity error via PERR# during configuration write cycles	✔	

Test Scenario: 2.9. Target Receives Memory Cycles

If IUT does not interface to a memory subsystem mark all N/A. If IUT does not interface to main system memory or memory is not cacheable mark 2 through 4 N/A.

Table A-86: **Target Receives Memory Cycles Checklist**

Test	Description	Pass	N/A
1	IUT completes single memory read and write cycles appropriately	✔	
2	IUT completes memory read line cycles appropriately	✔	
3	IUT completes memory read multiple cycles appropriately	✔	
4	IUT completes memory write and invalidate cycles appropriately	✔	
5	IUT completes one cycle and disconnects on reserved memory operations	✔	
6	IUT disconnects on burst transactions that cross its address boundary	✔	

Test Scenario: 2.10. Target Gets Memory Cycles With Address and Data Parity Errors

If IUT does not interface to a memory subsystem mark 1 to 2 N/A.

Table A-87: **Target Gets Memory Cycles With Address and Data Parity Errors Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR# during all memory read and write cycles	✔	
2	IUT reports data parity error via PERR# during all memory write cycles	✔	

Test Scenario: 2.11. Target Gets Fast Back to Back Cycles

If IUT does not implement the *fast back-to-back* bit, mark 3 and 4 N/A.

Table A-88: **Target Gets Fast Back to Back Cycles Checklist**

Test	Description	Pass	N/A
1	IUT responds to back to back memory writes appropriately	✔	
2	IUT responds to memory write followed by memory read appropriately	✔	
3	IUT responds to back to back memory writes with 2nd write selecting IUT		✔
4	IUT responds to memory write followed by memory read with read selecting IUT		✔

Table A-89: **Explanations for Test Scenarios 2.1 - 2.11**

Explanations
Test 2.4: The user application is responsible for terminating the transfer.
Test 2.6: The implementation under test does not support configuration bursts as a target.
Test 2.9: The user application is responsible for terminating the transfer.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.