pci_c MegaCore Function User Guide

Version 1.1
June 1999

pci_c MegaCore Function User Guide A-UG-PCIC-01.1
June 1999 P25-04562-00

Altera, BitBlaster, ByteBlaster, ByteBlasterMV, FLEX, FLEX 10K, MegaWizard, MAX, MAX+PLUS, MAX+PLUS II, MegaCore, OpenCore, and specific device designations are trademarks and/or service marks of Altera Corporation in the United States and/or other countries. Product elements and mnemonics used by Altera Corporation are protected by copyright and/or trademark laws.

Altera Corporation acknowledges the trademarks of other organizations for their respective products or services mentioned in this document.

Altera reserves the right to make changes, without notice, in the devices or the device specifications identified in this document. Altera advises its customers to obtain the latest version of device specifications to verify, before placing orders, that the information being relied upon by the customer is current. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty. Testing and other quality control techniques are used to the extent Altera deems such testing necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed. The megafunctions described in this catalog are not designed nor tested by Altera, and Altera does not warrant their performance or fitness for a particular purpose, or non-infringement of any patent, copyright, or other intellectual property rights. In the absence of written agreement to the contrary, Altera assumes no liability for Altera applications assistance, customer's product design, or infringement of patents or copyrights of third parties by or arising from use of semiconductor devices described herein. Nor does Altera warrant non-infringement of any patent, copyright, or other intellectual property right covering or relating to any combination, machine, or process in which such semiconductor devices might be or are used.

Altera's products are not authorized for use as critical components in life support devices or systems without the express written approval of the president of Altera Corporation. As used herein:

1. Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Products mentioned in this document may be covered by one or more of the following U.S. patents: 5,821,787: 5,821,771; 5,815,726; 5,815,024; 5,812,479; 5,812,450; 5,809,281; 5,805,516; 5,802,540; 5,801,541; 5,796,267; 5,793,246; 5,790,469; 5,787,009; 5,771,264; 5,768,562; 5,768,372; 5,767,734; 5,764,583; 5,764,569; 5,764,080; 5,764,079; 5,761,099; 5,760,624; 5,757,207; 5,757,070; 5,744,991; 5,744,383; 5,740,110; 5,732,020; 5,729,495; 5,717,901; 5,705,939; 5,699,020; 5,699,312; 5,696,455; 5,693,540; 5,694,058; 5,691,653; 5,689,195; 5,668,771; 5,680,061; 5,672,985; 5,670,895; 5,659,717; 5,650,734; 5,649,163; 5,642,262; 5,642,082; 5,633,830; 5,631,576; 5,621,312; 5,614,840; 5,612,642; 5,608,337; 5,606,276; 5,606,266; 5,604,453; 5,598,109; 5,598,108; 5,592,106; 5,592,102; 5,590,305; 5,583,749; 5,581,501; 5,574,893; 5,572,717; 5,572,148; 5,572,067; 5,570,040; 5,567,177; 5,565,793; 5,563,592; 5,561,757; 5,557,217; 5,555,214; 5,550,842; 5,550,782; 5,548,552; 5,548,228; 5,543,732; 5,543,730; 5,541,530; 5,537,295; 5,537,057; 5,525,917; 5,525,827; 5,523,706; 5,523,247; 5,517,186; 5,498,975; 5,495,182; 5,493,526; 5,493,519; 5,490,266; 5,488,586; 5,487,143; 5,486,775; 5,485,103; 5,485,102; 5,483,178; 5,481,486; 5,477,474; 5,473,266; 5,463,328, 5,444,394; 5,438,295; 5,436,575; 5,436,574; 5,434,514; 5,432,467; 5,414,312; 5,399,922; 5,384,499; 5,376,844; 5,375,086; 5,371,422; 5,369,314; 5,359,243; 5,359,242; 5,353,248; 5,352,940; 5,309,046; 5,350,954; 5,349,255; 5,341,308; 5,341,048; 5,341,044; 5,329,487; 5,317,212; 5,317,210; 5,315,172; 5,301,416; 5,294,975; 5,285,153; 5,280,203; 5,274,581; 5,272,368; 5,268,598; 5,266,037; 5,260,611; 5,260,610; 5,258,668; 5,247,478; 5,247,477; 5,243,233; 5,241,224; 5,237,219; 5,220,533; 5,220,214; 5,200,920; 5,187,392; 5,166,604; 5,162,680; 5,144,167; 5,138,576; 5,128,565; 5,121,006; 5,111,423; 5,097,208; 5,091,661; 5,066,873; 5,045,772; 4,969,121; 4,930,107; 4,930,098; 4,930,097; 4,912,342; 4,903,223; 4,899,070; 4,899,067; 4,871,930; 4,864,161; 4,831,573; 4,785,423; 4,774,421; 4,713,792; 4,677,318; 4,617,479; 4,609,986; 4,020,469; and certain foreign patents.

I.S. EN ISO 9001

Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights.

Copyright © 1999 Altera Corporation. All rights reserved.

Printed on Recycled Paper.

About this User Guide

June 1999

User Guide Contents

This user guide should be used in conjunction with the Altera® pci_c function version 1.1. This user guide describes the specifications of the pci_c function and how to use it in your design. The information in this user guide is current as of the printing date, but megafunction specifications are subject to change. For the most current information, refer to the Altera world-wide web site at **http://www.altera.com**.

For additional details on the functions, including availability, pricing, and delivery terms, contact your local Altera sales representative.

How to Contact Altera

For additional information about Altera products, consult the sources shown in Table 1.

Table 1. Contact Information

Information Type	Access	U.S. & Canada	All Other Locations
Literature	Altera Express	(800) 5-ALTERA	(408) 544-7850
	Altera Literature Services	(888) 3-ALTERA lit_req@altera.com	(888) 3-ALTERA lit_req@altera.com
Non-Technical Customer Service	Telephone Hotline	(800) SOS-EPLD	(408) 544-7000
	Fax	(408) 544-8186	(408) 544-7606
Technical Support	Telephone Hotline (6:00 a.m. to 6:00 p.m. Pacific Time)	(800) 800-EPLD	(408) 544-7000
	Fax	(408) 544-6401	(408) 544-6401
	Electronic Mail	sos@altera.com	sos@altera.com
	FTP Site	ftp.altera.com	ftp.altera.com
General Product Information	Telephone	(408) 544-7104	(408) 544-7104
	World-Wide Web	http://www.altera.com	http://www.altera.com

Typographic Conventions

The *PCI MegaCore Function User Guide* uses the typographic conventions shown in Table 2.

Table 2. PCI MegaCore Function User Guide Conventions

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names and dialog box titles are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: $\mathbf{f_{MAX}}$, **\maxplus2** directory, **d:** drive, **chiptrip.gdf** file.
Bold italic type	Book titles are shown in bold italic type with initial capital letters. Example: ***1999 Data Book***.
Italic Type with Initial Capital Letters	Document titles, checkbox options, and options in dialog boxes are shown in italic type with initial capital letters. Examples: *AN 75 (High-Speed Board Design)*, the *Check Outputs* option, the *Directories* box in the **Open** dialog box.
Italic type	Internal timing parameters and variables are shown in italic type. Examples: t_{PIA}, $n + 1$. Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: *<file name>*, *<project name>*.**pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of MAX+PLUS II Help topics are shown in quotation marks. Example: "Configuring a FLEX 10K or FLEX 8000 Device with the BitBlaster™ Download Cable."
`Courier type`	Reserved signal and port names are shown in uppercase Courier type. Examples: `DATA1`, `TDI`, `INPUT`. User-defined signal and port names are shown in lowercase Courier type. Examples: `my_data`, `ram_input`. Anything that must be typed exactly as it appears is shown in Courier type. For example: `c:\max2work\tutorial\chiptrip.gdf`. Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword `SUBDESIGN`), as well as logic function names (e.g., `TRI`) are shown in Courier.
1., 2., 3., and a., b., c.,...	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■	Bullets are used in a list of items when the sequence of the items is not important.
✓	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
↵	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.

Contents

June 1999, ver. 1.1

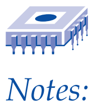
Introduction 1
PCI MegaCore Functions 3
OpenCore Feature 4
Altera Devices 5
Software Tools 5
Verification 6
References 7

Getting Started 9
Before You Begin 11
Walk-Through Overview 14
Using Third-Party EDA Tools 19

MegaCore Overview 23
Features 25
General Description 26
Compliance Summary 29
PCI Bus Signals 30
Parameters 43
Functional Description 46

Specifications 53
PCI Bus Commands 55
Configuration Registers 56
Target Mode Operation 70
Master Mode Operation 108
64-Bit Addressing, Dual Address Cycle (DAC) 143

PCI SIG Protocol Checklists 149
Checklists 149
PCI SIG Test Scenarios 156



Notes:

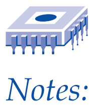
ALTERA®

Introduction

Contents

June 1999

PCI MegaCore Functions 3
OpenCore Feature 4
Altera Devices 5
Software Tools 5
Verification 6
References 7



Notes:

Introduction

June 1999, ver. 1.1

As programmable logic device (PLD) densities grow to over 1 million gates, design flows must be as efficient and productive as possible. Altera provides ready-made, pre-tested, and optimized megafunctions that let you rapidly implement the functions you need, instead of building them from the ground up. Altera® MegaCore™ functions, which are reusable blocks of pre-designed intellectual property, improve your productivity by allowing you to concentrate on adding proprietary value to your design. When you use MegaCore functions, you can focus on your high-level design and spend more time and energy on improving and differentiating your product.

Altera PCI solutions include PCI MegaCore functions developed and supported by Altera. Altera's FLEX® devices easily implement PCI applications, while leaving ample room for your custom logic. The devices are supported by Altera's MAX+PLUS® II development system, which allows you to perform a complete design cycle including design entry, synthesis, place-and-route, simulation, timing analysis, and device programming. Altera's PCI MegaCore functions are hardware-tested using the HP PCI Analyzer and Exerciser product series. Combined with Altera's FLEX devices, Altera software, and extensive hardware testing, Altera PCI MegaCore functions provide you with a complete design solution.

PCI MegaCore Functions

The PCI MegaCore functions are developed and supported by Altera. Four PCI MegaCore functions are currently offered (see Table 1). You can use the OpenCore™ feature in the MAX+PLUS II software to test-drive PCI and other MegaCore functions before you decide to license the function. This user guide discusses the `pci_c` MegaCore function.

Table 1. Altera PCI MegaCore Functions

Function	Description
`pci_a`	32-bit master/target interface function with direct memory access (DMA)
`pcit1`	32-bit target interface function
`pci_b`	32-bit customizable master/target interface function
`pci_c`	64-bit customizable master/target interface function

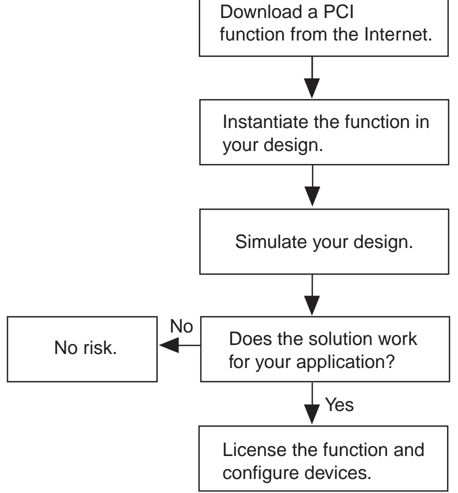
For more information, refer to the following documents:

- *PCI Master/Target MegaCore Function with DMA Data Sheet*
- *pcit1 PCI Target MegaCore Function Data Sheet*
- *pci_b PCI Master/Target MegaCore Function Data Sheet*
- *PCI MegaCore Function User Guide*

OpenCore Feature

Altera's exclusive OpenCore feature allows you to evaluate MegaCore functions before deciding to license them. You can instantiate a MegaCore function in your design, compile and simulate the design, and then verify the MegaCore function's size and performance. This evaluation provides first-hand functional, timing, and other technical data that allows you to make an informed decision on whether to license the MegaCore function. Once you license a MegaCore function, you can use the MAX+PLUS II software to generate programming files, as well as EDIF, VHDL, or Verilog HDL output netlist files for simulation in third-party EDA tools. Figure 1 shows a typical design flow using MegaCore functions and the OpenCore feature. All MegaCore functions can be evaluated risk-free by downloading them from the Altera web site at **http://www.altera.com**.

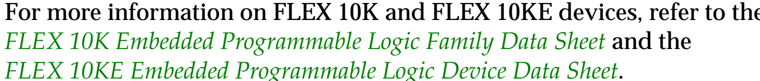
Figure 1. OpenCore Design Flow



Altera Devices

The PCI MegaCore functions have been optimized and targeted for Altera PCI-compliant FLEX devices. The `pci_c` MegaCore function has been optimized and targeted specifically for the new 2.5-V FLEX 10KE devices. FLEX 10KE devices deliver the flexibility of traditional programmable logic with the efficiency and density of gate arrays with embedded memory.

FLEX 10KE devices offer enhanced performance from the 5.0-V FLEX 10K family. Designed for compliance with the 3.3-V, 66 MHz PCI specification, FLEX 10KE devices offer 100-MHz system speed and 150-MHz first-in first-out (FIFO) buffers in devices with densities from 30,000 to 200,000 gates. FLEX 10KE devices feature dual-port embedded array blocks (EABs), which are 4,096 Kbits of RAM that can be configured as 256 × 16, 512 × 8, 1,024 × 4, or 2,048 × 2 blocks. Additionally, the FLEX 10KE devices offer all the features of programmable logic: ease-of-use, fast and predictable performance, register-rich architecture, and in-circuit reconfigurability (ICR). Together, these features enable FLEX 10KE devices to achieve the fastest high-density performance in the programmable logic market.



For more information on FLEX 10K and FLEX 10KE devices, refer to the *FLEX 10K Embedded Programmable Logic Family Data Sheet* and the *FLEX 10KE Embedded Programmable Logic Device Data Sheet*.

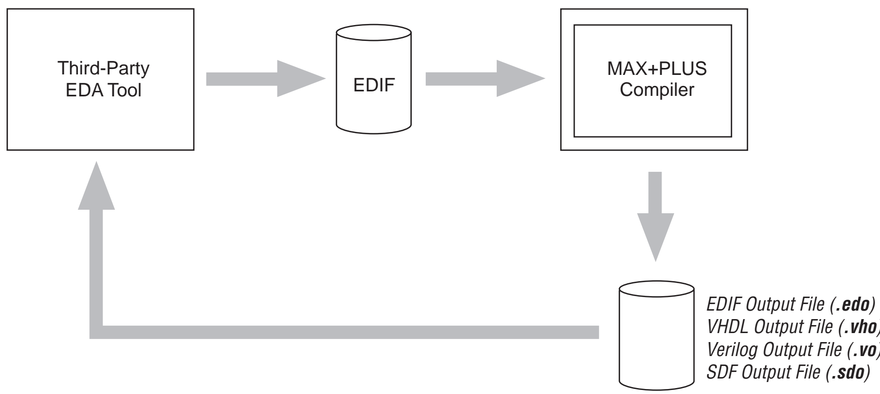
Software Tools

Long recognized as the best development system in the programmable logic industry, the MAX+PLUS II software continues to offer unmatched flexibility and performance. The MAX+PLUS II software offers a completely integrated development flow and an intuitive, Windows-based graphical user interface, making it easy to learn and use. The software lets you quickly implement and test changes in your design, program Altera PLDs at your desktop, and eliminates the long lead times typically associated with gate arrays.

The MAX+PLUS II software offers a seamless development flow, allowing you to enter, compile, and simulate your design and program devices using a single, integrated tool, regardless of the Altera device you choose. The MAX+PLUS II software supports industry-standard VHDL and Verilog HDL design descriptions, as well as EDIF netlists generated by third-party EDA schematic and synthesis tools.

As a standard feature, the MAX+PLUS II software interfaces with all major EDA design tools, including tools for ASIC designers. Once a design is captured and simulated using the tool of your choice, you can transfer your EDIF file directly into the MAX+PLUS II software. After synthesis and fitting, you can transfer your file back into your tool of choice for simulation. The MAX+PLUS II system outputs the full-timing VHDL, Verilog HDL, Standard Delay Format (SDF), and EDIF netlists that can be used for post-route device- and system-level simulation. Figure 2 shows the typical design flow when using the MAX+PLUS II software with other EDA tools.

Figure 2. MAX+PLUS II/EDA Tool Design Flow



To simplify the design flow between the MAX+PLUS II software and other EDA tools, Altera has developed the MAX+PLUS II Altera Commitment to Cooperative Engineering Solutions (ACCESS℠) Key Guidelines. These guidelines provide complete instructions on how to create, compile, and simulate your design with tools from leading EDA vendors. These guidelines are available on the MAX+PLUS II installation CD-ROM and on the Altera web site at **http://www.altera.com**.

Verification

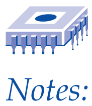
Altera has simulated and hardware tested the PCI MegaCore functions extensively in real systems and against multiple PCI bridges. Altera tested numerous vectors for different PCI transactions to analyze the PCI traffic and check for protocol violations. Altera's aggressive hardware testing policy produces PCI functions that are far more robust than could be achieved from simulation alone.

For information on the equipment used for hardware testing, please see "Compliance Summary" on page 29 in the Overview section of this user guide.

References

Reference documents for the pci_c function include:

- *PCI Local Bus Specification, Revision 2.2*. PCI SIG. Portland, Oregon: PCI Special Interest Group, December 1998.
- *PCI Compliance Checklist, Revision 2.1*. PCI SIG. Portland, Oregon.



Notes:

Getting Started

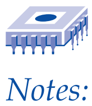
Contents

June 1999

Before You Begin 11
- Obtaining MegaCore Functions 11
- Installing the MegaCore Files 12
- MegaCore Directory Structure 13

Walk-Through Overview 14
- Design Entry 15
- Functional Compilation/Simulation 16
- Run the make_acf Utility 16
- Timing Compilation & Analysis 18
- Configuring a Device 18

Using Third-Party EDA Tools 19
- VHDL & Verilog HDL Functional Models 20
- Synthesis Compilation & Post-Routing Simulation 21



Notes:

Getting Started

June 1999, ver. 1.1

Altera PCI MegaCore™ functions provide solutions for integrating 32- and 64-bit PCI peripheral devices, including network adapters, graphic accelerator boards, and embedded control modules. The functions are optimized for Altera® FLEX® devices, greatly enhancing your productivity by allowing you to focus efforts on the custom logic surrounding the PCI interface. The PCI MegaCore functions are fully tested to meet the requirements of the PCI Special Interest Group (SIG) *PCI Local Bus Specification, Revision 2.2* and *Compliance Checklist, Revision 2.1*.

This section describes how to obtain Altera PCI MegaCore functions, explains how to install them on your PC or workstation, and walks you through the process of implementing the function in a design. You can test-drive MegaCore functions using Altera's OpenCore™ feature to simulate the functions within your custom logic. When you are ready to license a function, contact your local Altera sales representative.

Before You Begin

Before you can start using Altera PCI MegaCore functions, you must obtain the MegaCore files and install them on your PC or workstation. The following instructions describe this process and explain the directory structure for the functions.

Obtaining MegaCore Functions

If you have Internet access, you can download MegaCore functions from Altera's web site at **http://www.altera.com**. Follow the instructions below to obtain the MegaCore functions via the Internet. If you do not have Internet access, you can obtain the MegaCore functions from your local Altera representative.

1. Run your web browser (e.g., Netscape Navigator or Microsoft Internet Explorer).

2. Open the URL **http://www.altera.com**.

3. Click the Altera Megafunctions link.

4. Click the link for the Altera PCI MegaCore function you wish to download.

5. Follow the on-line instructions to download the function and save it to your hard disk.

Installing the MegaCore Files

Depending on your platform, use the following instructions:

Windows NT 3.51

For Windows NT 3.51, follow the instructions below:

1. Open the Program Manager.

2. Click **Run** (File menu).

3. Type <*path name*>\<*filename*>`.exe`, where <*path name*> is the location of the downloaded MegaCore function and <*filename*> is the filename of the function.

4. Click **OK**. The **MegaCore Installer** dialog box appears. Follow the on-line instructions to finish installation.

Windows 95/98 & Windows NT 4.0

For Windows 95/98 and Windows NT 4.0, follow the instructions below:

1. Click **Run** (Start menu).

2. Type <*path name*>\<*filename*>`.exe`, where <*path name*> is the location of the downloaded MegaCore function and <*filename*> is the filename of the function.

3. Click **OK**. The **MegaCore Installer** dialog box appears. Follow the on-line instructions to finish installation.

UNIX

At a UNIX command prompt, change to the directory in which you saved the downloaded MegaCore function and type the following commands:

```
uncompress <filename>.tar.Z ↵
tar xvf <filename>.tar ↵
```

MegaCore Directory Structure

Altera PCI MegaCore function files are organized into several directories; the top-level directory is **\megacore** (see Table 1).

The MegaCore directory structure may contain several MegaCore products. Additionally, Altera updates MegaCore files from time to time. Therefore, Altera recommends that you do not save your project-specific files in the MegaCore directory structure.

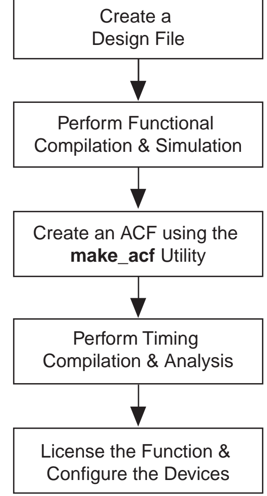
Table 1. PCI MegaCore Directories

Directory	Description
\bin	Contains the **make_acf** utility that generates a MAX+PLUS II Assignment & Configuration File (**.acf**) for your custom design hierarchy. The generated ACF contains all necessary assignments to ensure that all PCI timing requirements are met.
\lib	Contains encrypted lower-level design files. After installing the MegaCore function, you should set a user library in the MAX+PLUS II software that points to this directory. This library allows you to access all the necessary MegaCore files.
\pci_c	Contains the MegaCore function files.
\pci_c\acf	Contains ACFs for targeted Altera FLEX devices. These ACFs contain all necessary assignments to meet PCI timing requirements. By using the **make_acf** utility, you can annotate the assignments in one of these ACFs for your project.
\pci_c\doc	Contains documentation for the function.
\pci_c\examples	The **\examples** directory has subdirectories containing examples for FLEX device/package combinations. Each subdirectory contains a Graphic Design File (**.gdf**) and an ACF. For more information, refer to the **readme** file in the **\examples** directory. The **\examples** directory also contains the following subdirectory: ■ **\sim_top**, which contains a GDF and an ACF that can be used to perform functional compilation and simulation of the PCI MegaCore function.
\pci_c\sim\scf	Contains the Simulator Channel Files (**.scf**) for different PCI protocol transactions that can be used to verify the functionality of the Altera PCI MegaCore function.

Walk-Through Overview

This section describes the PCI design flow using an Altera PCI MegaCore function and the MAX+PLUS II development system (see Figure 1).

Figure 1. Example PCI Design Flow



The following instructions assume that:

- You are using the `pci_c` MegaCore function.
- All files are located in the default directory, **c:\megacore**. If the files are installed in a different directory on your system, substitute the appropriate path name.
- You are using a PC; UNIX users should alter the steps as appropriate.
- You are familiar with the MAX+PLUS II software.
- MAX+PLUS II version 9.22 or higher is installed in the default location (i.e., **c:\maxplus2**).
- You are using the OpenCore feature to test-drive the function or you have licensed the function.

You can use Altera's OpenCore feature to compile and simulate the PCI MegaCore functions, allowing you to evaluate the functions before deciding to license them. However, you must obtain a license from Altera before you can generate programming files or EDIF, VHDL, or Verilog HDL netlist files for simulation in third-party EDA tools.

The sample design process uses the following steps:

1. Create a GDF that instantiates the PCI MegaCore function.

2. Perform functional compilation and simulation to evaluate and verify the functionality.

3. Run the **make_acf** utility to create an ACF that contains the necessary assignments for meeting the targeted device's PCI timing requirements.

4. Perform timing compilation and analysis to verify that the PCI timing specifications are met.

5. If you have licensed the MegaCore function, configure a targeted Altera FLEX device with the completed design.

Design Entry

The following steps explain how to create a GDF that instantiates the `pci_c` MegaCore function.

☞ Refer to MAX+PLUS II Help for detailed instructions on how to use the Graphic Editor.

1. Run the MAX+PLUS II software.

2. Specify user libraries for the `pci_c` function. Choose **User Libraries** (Options menu) and specify the directory **c:\megacore\lib**.

3. Create a directory to hold your design file, e.g., **c:\altr_app**.

4. Create a new GDF named **pci_top.gdf** and save it to your new directory (e.g., **c:\altr_app\pci_top.gdf**).

5. Choose **Project Set Project to Current File** (File menu) and specify the **pci_top.gdf** file as the current project.

6. Enter the schematic shown in the **pci_top.gdf** file in the **\examples\sim_top** directory. You may skip this step by copying the schematic in the **pci_top.gdf** file into your **pci_top.gdf** file in your working directory.

After you have entered your design, you are ready to perform functional simulation to evaluate and verify the functionality.

Functional Compilation/Simulation

The following steps explain how to functionally compile and simulate your design.

1. In the MAX+PLUS II Compiler, turn on **Functional SNF Extractor** (Processing menu).

2. Click **Start** to compile your design.

3. In the MAX+PLUS II Simulator, choose **Inputs/Outputs** (File Menu), specify **c:\megacore\pci_c\sim\scf**<*target or master transactions*>**.scf** in the *Input* box, and choose **OK**.

4. Click **Start** to simulate your design.

5. Click **Open SCF** to view the simulation file. The different simulation files show the behavior of the PCI and local-side signals for different types of transactions.

After you have verified that your design is functionally correct, you are ready to synthesize and place-and-route your design. However, you still need to generate an ACF to ensure that all of the PCI signals in your design meet the PCI timing specifications.

Run the make_acf Utility

The **make_acf** utility, located in the **c:\megacore\bin** directory, is used to generate an ACF that contains the placement and configuration assignments to meet the PCI timing specifications. For more information on the **make_acf** utility, refer to the documentation in the **c:\megacore\bin** directory.

For the **make_acf** utility to operate correctly, you must use directory names and filenames that are eight (8) characters or less.

In the previous section, the `NUMBER_OF_BARS` parameter is set to a decimal value of 6 because `BAR0` through `BAR5` parameter settings are based upon the functional simulations in the **\sim\scf** directory. This setting allows you to evaluate the functionality of the `pci_c` MegaCore function. The number of base address registers (BARs) that are instantiated and the size of the memory for each BAR instantiated affects the amount of logic that is generated for your design. If the `NUMBER_OF_BARS` parameter is set to a value less than 6, the logic for the unused BARs will not be generated.

Generate the file pci_top.acf by performing the following steps:

1. Open **pci_top.gdf**. Set the following parameters: `NUMBER_OF_BARS = 1`, `BAR0 = "H"FFF00000""`, and `TARGET_DEVICE = "EPF10K200EFC672"`. Double-click on the **Parameters Field** of the `pci_c` symbol to open the **Edit Ports/Parameters** dialog box.

When changing a parameter value, only change the number, i.e., leave the hexadecimal indicator `H` and quotation marks. If you delete these characters, you will receive a compilation error. Additionally, when setting register values, the MAX+PLUS II software may issue several warning messages indicating that one or more registers are stuck at ground. These warning messages can be ignored.

2. Run the **make_acf** utility by typing the following command at a DOS command prompt:

   `c:\megacore\bin\make_acf` ↵

3. You are prompted with several questions. Type the following after each question. (The bold text is the prompt text.)

   Enter the hierarchical name for the PCI MegaCore:

   `|pci_c:`*`YY`* ↵

   where *YY* is the instance name for the MegaCore function. In a GDF, it is the number in the lower left-hand corner of the PCI MegaCore symbol.

   Enter the chip name:

   `pci_top` ↵

   Type the path and name of the output acf file:

   `c:\altr_app\pci_top.acf` ↵

   Type the path and name of the input acf file:

   `c:\megacore\pci_c\acf\200EF672.acf` ↵

For a listing of the supported Altera device ACFs, refer to the readme file in **\megacore\pci_c\doc**.

4. After you have generated your ACF, you are ready to perform timing compilation to synthesize and place and route your design.

Timing Compilation & Analysis

The following steps explain how to perform timing compilation and analysis.

1. Choose **Project Set Project to Current File** (File menu).

2. In the Compiler, turn off the **Functional SNF Extractor** command (Processing menu).

3. Click **Start** to begin compilation.

4. After a successful compilation, open the Timing Analyzer. There are three forms of timing analysis you can perform on your design:

   - In the Timing Analyzer, choose **Registered Performance** (Analysis menu). The Registered Performance Display calculates the maximum clock frequency and identifies the longest delay paths between registers.
   - In the Timing Analyzer, choose **Delay Matrix** (Analysis menu). The Delay Matrix Display calculates combinatorial delays, e.g., t_{CO} and t_{PD}.
   - In the Timing Analyzer, choose **Setup/Hold Matrix** (Analysis menu). The Setup/Hold Matrix Display calculates the setup and hold times of the registers.

You are now ready to configure your targeted Altera FLEX device.

Configuring a Device

After you have compiled and analyzed your design, you are ready to configure your targeted Altera FLEX device. If you are evaluating the PCI MegaCore function with the OpenCore feature, you must license the PCI MegaCore function before you can generate configuration files. Altera provides three types of hardware to configure FLEX devices:

- The Altera Stand-Alone Programmer (ASAP2) includes an LP6 Logic Programmer card and a Master Programming Unit (MPU). You should use a PLMJ1213 programming adapter with the MPU to program a serial configuration device, which loads the configuration data to the FLEX device during power-up. A Programmer Object File (**.pof**) is used to program the configuration device. The Altera Stand-Alone Programmer is typically used in the production stage of the design flow.
- The BitBlaster™ serial download cable is a hardware interface to a standard PC or UNIX workstation RS-232 port. An SRAM Object File (**.sof**) is used to configure the FLEX device. The BitBlaster cable is typically used in the prototyping stage of the design flow.
- The ByteBlaster™ and ByteBlasterMV™ parallel port download cables provide a hardware interface to a standard parallel port. (The ByteBlaster cable is obsolete and is replaced by the ByteBlasterMV cable.) The SOF is used to configure the FLEX device. The ByteBlaster and ByteBlasterMV cables are typically used in the prototyping stage.

For more information, refer to the *BitBlaster Serial Download Cable Data Sheet*, *ByteBlaster Parallel Port Download Cable Data Sheet*, and *ByteBlasterMV Parallel Port Download Cable Data Sheet*.

Perform the following steps to set up the MAX+PLUS II configuration interface. For more information, refer to MAX+PLUS II Help.

1. Open the Programmer.
2. Choose **Hardware Setup** (Options menu).
3. In the **Hardware Setup** dialog box, select your programming hardware in the *Hardware Type* box and click **OK**.
4. Choose **Select Programming File** (File menu) and select your programming filename.
5. Click **Program** to program a serial configuration device, or click **Configure** if you are using the BitBlaster, ByteBlaster, or ByteBlasterMV cables.

Using Third-Party EDA Tools

As a standard feature, Altera's MAX+PLUS II software works seamlessly with tools from all EDA vendors, including Cadence, Exemplar Logic, Mentor Graphics, Synopsys, Synplicity, and Viewlogic. After you have licensed the MegaCore function, you can generate EDIF, VHDL, Verilog HDL, and Standard Delay output files from the MAX+PLUS II software and use them with your existing EDA tools to perform functional modeling and post-route simulation of your design.

To simplify the design flow between the MAX+PLUS II software and other EDA tools, Altera has developed the MAX+PLUS II Altera Commitment to Cooperative Engineering Solutions (ACCESS^{SM}) Key Guidelines. These guidelines provide complete instructions on how to create, compile, and simulate your design with tools from leading EDA vendors. The MAX+PLUS II ACCESS Key Guidelines are part of Altera's ongoing efforts to give you state-of-the-art tools that fit into your design flow, and to enhance your productivity for even the highest-density devices. The MAX+PLUS II ACCESS Key Guidelines are available on the Altera web site (**http://www.altera.com**) and the MAX+PLUS II CD-ROM.

The following sections describe how to generate a VHDL or Verilog HDL functional model, and describe the design flow to compile and simulate your custom Altera PCI MegaCore design with a third-party EDA tool. Refer to Figure 2 on page 6, which shows the design flow for interfacing your third-party EDA tool with the MAX+PLUS II software.

VHDL & Verilog HDL Functional Models

To generate a VHDL or Verilog HDL functional model, perform the following steps:

1. In the MAX+PLUS II software, open a **pci_top.gdf** file located in any of the FLEX device/package example subdirectories in the **\megacore\pci_c\examples** directory.

2. In the Compiler, ensure that the **Functional SNF Extractor** command (Processing menu) is turned off.

3. Turn on the **Verilog Netlist Writer** or **VHDL Netlist Writer** command (Interfaces menu), depending on the type of output file you want to use in your third-party simulator.

4. Choose **Verilog Netlist Writer Settings** (Interface menu) if you turned on **Verilog Netlist Writer**.

5. In the **Verilog Netlist Writer Settings** dialog box, select either *SDF Output File [.sdo] Ver 2.1* or *SDF Output File [.sdo] Ver.1.0* and click **OK**. Selecting one of these options causes the MAX+PLUS II software to generate the files **pci_top.vo**, **pci_top.sdo**, and **alt_max2.vo**. **pci_top.vo** is the functional model of your PCI MegaCore design. The **pci_top.sdo** file contains the timing information. The **alt_max2.vo** file contains the functional models of any Altera macrofunctions or primitives.

6. Choose **VHDL Netlist Writer Settings** (Interface menu) if you turned on **VHDL Netlist Writer**.

7. In the **VHDL Netlist Writer Settings** dialog box, select either *SDF Output File [.sdo] Ver 2.1 (VITAL)* or *SDF Output File [.sdo] Ver. 1.0* and click **OK**. Choosing one of these options causes the MAX+PLUS II software to generate the files **pci_top.vho** and **pci_top.sdo**. The **pci_top.vho** file is the functional model of your PCI MegaCore design. The **pci_top.sdo** file contains the timing information.

8. Compile the **pci_top.vo** or **pci_top.vho** output files in your third-party simulator to perform functional simulation using Verilog HDL or VHDL.

Synthesis Compilation & Post-Routing Simulation

To synthesize your design in a third-party EDA tool and perform post-route simulation, perform the following steps:

1. Create your custom design instantiating a PCI MegaCore function.

2. Synthesize the design using your third-party EDA tool. Your EDA tool should treat the PCI MegaCore instantiation as a black box by either setting attributes or ignoring the instantiation.

For more information on setting compiler options in your third-party EDA tool, refer to the MAX+PLUS II ACCESS Key Guidelines.

3. After compilation, generate a hierarchical EDIF netlist file in your third-party EDA tool.

4. Open your EDIF file in the MAX+PLUS II software.

5. Run the **make_acf** utility to generate an ACF for your targeted FLEX device. Refer to "Run the make_acf Utility" on page 16 for more information.

6. Set your EDIF file as the current project in the MAX+PLUS II software.

7. Choose **EDIF Netlist Reader Settings** (Interfaces menu).

8. In the **EDIF Netlist Reader Settings** dialog box, select the vendor for your EDIF netlist file in the *Vendor* drop-down list box and click **OK**.

9. Make logic option and/or place-and-route assignments for your custom logic using the commands in the Assign menu.

10. In the MAX+PLUS II Compiler, make sure **Functional SNF Extractor** (Processing menu) is turned off.

11. Turn on the **Verilog Netlist Writer** or **VHDL Netlist Writer** command (Interfaces menu), depending on the type of output file you want to use in your third-party simulator. Set the netlist writer settings as described in step 5 in "VHDL & Verilog HDL Functional Modeling."

12. Compile your design. The MAX+PLUS II Compiler synthesizes and performs place-and-route on your design, and generates output and programming files.

13. Import your MAX+PLUS II-generated output files (**.edo**, **.vho**, **.vo**, or .**sdo**) into your third-party EDA tool for post-route, device-level, and system-level simulation.

MegaCore Overview

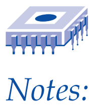
Contents

June 1999

Features 25
General Description 26
Compliance Summary 29
PCI Bus Signals 30
Target Local-Side Signals 36
Master Local-Side Signals 40
Parameters 43
Functional Description 46
Target Device Signals & Signal Assertion 48
Master Device Signals & Signal Assertion 50

3

MegaCore Overview



Notes:

MegaCore Overview

June 1999, ver. 1.1

Features...

This section describes the features of the `pci_c` MegaCore™ function. The `pci_c` function is a parameterized MegaCore function implementing a 64-bit peripheral component interconnect (PCI) master/target interface.

- A flexible general-purpose interface that can be customized for specific peripheral requirements
- Dramatically shortens design cycles
- Fully compliant with the PCI Special Interest Group (PCI SIG) ***PCI Local Bus Specification, Revision 2.2*** timing and functional requirements
- Extensively verified using industry-proven Phoenix Technology test bench
- Extensively hardware tested using the following hardware and software (see "Compliance Summary" on page 29 for details)
  - HP E2928A PCI Bus Analyzer and Exerciser
  - HP E2920 Computer Verification Tools, PCI series
  - Altera's intellectual property (IP) development board
- Optimized for the FLEX® 10K architecture
- 66-MHz compliant with FLEX 10KE-1 devices
- No-risk OpenCore™ feature allows designers to instantiate and simulate designs in the MAX+PLUS® II software prior to purchase
- PCI master features:
  - Infinite cycles zero-wait state PCI read/write operation (up to 528 Mbytes per second)
  - Initiates most PCI commands including: configuration read/write, memory read/write, I/O read/write, memory read multiple (MRM), memory read line (MRL), memory write and invalidate (MWI)
  - Initiates 64-bit addressing, using Dual-Address Cycle (DAC)
  - Initiates 64-bit memory transactions
  - Initiates 32-bit memory, I/O and configuration transactions
  - Dynamically negotiates 64-bit transactions and automatically multiplexes data on the local 64-bit data bus.
  - PCI bus parking
  - Functions in host bridge applications by allowing master to power up enabled and initiating configuration cycles to its own target

...and More Features

- PCI target features:
  - Type zero configuration space
  - Capabilities list pointer support
  - Parity error detection
  - Up to six base address registers (BARs) with adjustable memory size and type
  - Expansion ROM BAR support
  - Zero-wait state PCI read/write (up to 528 Mbytes per second)
  - Most PCI bus commands are supported; configuration read/write, memory read/write, I/O read/write, MRM, MRL, and MWI
  - Local side can request a target abort, retry, or disconnect
  - 64-bit addressing capable
  - Automatically responds to 32- or 64-bit transactions
  - Local-side interrupt request
- Configuration registers:
  - Parameterized registers: device ID, vendor ID, class code, revision ID, BAR0 through BAR5, subsystem ID, subsystem vendor ID, maximum latency, minimum grant, capabilities list pointer, expansion ROM BAR
  - Non-parameterized registers: command, status, header type, latency timer, cache line size, interrupt pin, interrupt line

General Description

The `pci_c` MegaCore function (ordering code: PLSM-PCI/C) is a hardware-tested, high-performance, flexible implementation of the 64-bit PCI master/target interface. This function handles the complex PCI protocol and stringent timing requirements internally, and its backend interface is designed for easy integration. Therefore, designers can focus their engineering efforts on value-added custom development, significantly reducing time-to-market.

Optimized for Altera® FLEX 10K devices, the `pci_c` function supports configuration, I/O, and memory transactions. With the high density of FLEX devices, designers have ample resources for custom local logic after implementing the PCI interface. The high performance of FLEX devices also enables the `pci_c` function to support unlimited cycles of zero-wait-state memory-burst transactions. The `pci_c` function can run at either 33 MHz or 66 MHz PCI bus clock speeds, thus achieving 264 Mbytes per second throughput in a 64-bit, 33 MHz PCI bus system, or 528 Mbytes per second throughput in a 64-bit, 66 MHz PCI bus system.

In the `pci_c` function, the master and target interface can operate independently, allowing maximum throughput and efficient usage of the PCI bus. For instance, while the target interface is accepting zero-wait state burst write data, the local logic may simultaneously request PCI bus mastership, thus minimizing latency.

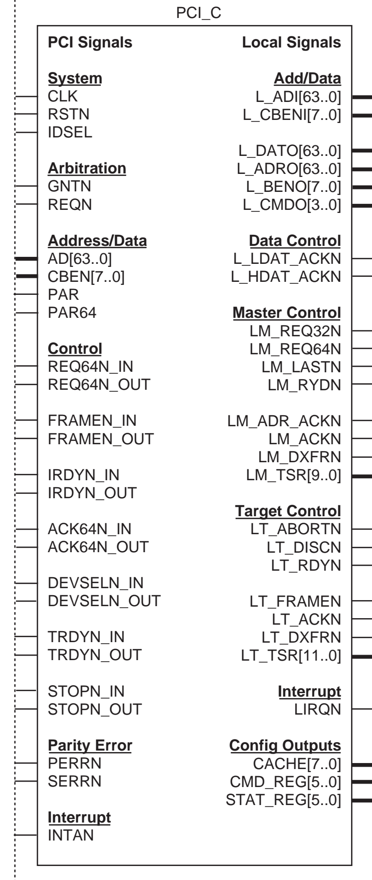
To ensure timing and protocol compliance, the `pci_c` function has been vigorously hardware tested. See "Compliance Summary" on page 29 for more information on the hardware tests performed.

As a parameterized function, `pci_c` has configuration registers that can be modified upon instantiation. These features provide scalability, adaptability, and efficient silicon implementation. As a result, the same MegaCore functions can be used in multiple PCI projects with different requirements. For example, the `pci_c` function offers up to six base address registers (BARs) for multiple local-side devices. However, some applications require only one contiguous memory range. PCI designers can choose to instantiate only one BAR, which reduces logic cell consumption. After designers define the parameter values, the MAX+PLUS II software automatically and efficiently modifies the design and implements the logic.

This user guide should be used in conjunction with the latest PCI specification, published by the PCI Special Interest Group (SIG). Users should be fairly familiar with the PCI standard before using this function. Figure 1 shows the symbol for the `pci_c` function.

Figure 1. pci_c Symbol

BAR0="H"FFF00000""
BAR1="H"FFF00000""
BAR2="H"FFF00000""
BAR3="H"FFF00000""
BAR4="H"FFF00000""
BAR5="H"FFF00000""
CAP_LIST_ENA="NO"
CAP_PTR=H"40"
CLASS_CODE=H"FF0000"
DEVICE_ID=H"0004"
EXP_ROM_BAR="H"FF000000""
EXP_ROM_ENA="NO"
HOST_BRIDGE_ENA="NO"
INTERNAL_ARBITER="NO"
MAX_LATENCY=H"00"
MIN_GRANT=H"00"
NUMBER_OF_BARS=1
PCI_66MHZ_CPABLE="YES"
REVISION_ID=H"01"
SUBSYSTEM_ID=H"0000"
SUBSYSTEM_VENDOR_ID=H"0000"
TARGET_DEVICE="EPF10K50EFC484"
VENDOR_ID=H"1172"



Compliance Summary

The `pci_c` function is compliant with the requirements specified in the PCI SIG *PCI Local Bus Specification, Revision 2.2* and *Compliance Checklist, Revision 2.1*. The function is shipped with sample MAX+PLUS II Simulator Channel Files (**.scf**), which can be used to validate the functions in the MAX+PLUS II software. Consult the **readme** files provided in the **\sim** directory for a complete list and description of the included simulations.

To ensure PCI compliance, Altera has performed extensive validation of the `pci_c` function. Validation includes both simulation and hardware testing. The following simulations are covered by the validation suite for `pci_c`:

- PCI SIG checklist simulations
- Applicable operating rules in PCI specification appendix C, including:
  - Basic protocol
  - Signal stability
  - Master and target signals
  - Data phases
  - Arbitration
  - Latency
  - Exclusive access
  - Device selection
  - Parity
- Local-side interface functionality
- Corner cases of the PCI and local-side interface, such as random wait state insertion.

In addition to simulation, Altera has performed extensive hardware testing on the `pci_c` function to ensure robustness and PCI compliance. The test platforms included the HP E2928A PCI Bus Exerciser and Analyzer, the Altera IP development board with a FLEX 10KE device configured with the MegaCore function, and PCI bus agents such as the host bridge, Ethernet network adapter, and video card. The hardware testing ensures that the `pci_c` function operates flawlessly under the most stringent conditions.

During hardware testing with the HP E2928A PCI Bus Exerciser and Analyzer, various tests are performed to guarantee robustness and strict compliance. These tests include:

- Memory read/write
- I/O read/write tests
- Configuration read/write tests

The tests were made to generate random transaction type and parameters at the PCI and local sides. The HP E2928A PCI Bus Exerciser and Analyzer simulates random behavior on the PCI bus by randomizing transactions with variable parameters such as:

- Bus commands
- Burst length
- Data types
- Wait states
- Terminations
- Error conditions

The local side also emulates the variety of conditions where `pci_c` is being used by randomizing the wait states and terminations. During the tests, the HP E2928A PCI Bus Exerciser and Analyzer also acts as a PCI protocol and data integrity checker as well as a logic analyzer to aid in debugging. This testing ensures that the `pci_c` function operates under the most stringent conditions in your system. For more information on the HP E2928A PCI Bus Exerciser and Analyzer, see the Hewlett Packard web site at **http://www.hp.com**.

PCI Bus Signals

The following PCI signals are used by the `pci_c` function:

- *Input*—Standard input-only signal.
- *Output*—Standard output-only signal.
- *Bidirectional*—Tri-state input/output signal.
- *Sustained tri-state (STS)*—Signal that is driven by one agent at a time (e.g., device or host operating on the PCI bus). An agent that drives a sustained tri-state pin low must actively drive it high for one clock cycle before tri-stating it. Another agent cannot drive a sustained tri-state signal any sooner than one clock cycle after it is released by the previous agent.
- *Open-drain*—Signal that is wire-ORed with other agents. The signaling agent asserts the open-drain signal, and a weak pull-up resistor deasserts the open-drain signal. The pull-up resistor may require two or three PCI bus clock cycles to restore the open-drain signal to its inactive state.

Table 1 summarizes the PCI bus signals that provide the interface between the pci_c function and the PCI bus.

Table 1. PCI Interface Signals (Part 1 of 3)

Name	Type	Polarity	Description
clk	Input	–	Clock. The clk input provides the reference signal for all other PCI interface signals, except rstn and intan.
rstn	Input	Low	Reset. The rstn input initializes the FLEX 10K PCI interface circuitry, and can be asserted asynchronously to the PCI bus clk edge. When active, the PCI output signals are tri-stated and the open-drain signals, such as serrn, float.
gntn	Input	Low	Grant. The gntn input indicates to the pci_c master device that it has control of the PCI bus. Every master device has a pair of arbitration lines (gntn and reqn) that connect directly to the arbiter.
reqn	Output	Low	Request. The reqn output indicates to the arbiter that the pci_c master wants to gain control of the PCI bus to perform a transaction.
ad[63..0]	Tri-State	–	Address/data bus. The ad[63..0] bus is a time-multiplexed address/data bus; each bus transaction consists of an address phase followed by one or more data phases. The data phases occur when irdyn and trdyn are both asserted.In the case of a 32-bit data phase, only ad[31..0] bus holds valid data.
cben[7..0]	Tri-State	Low	Command/byte enable. The cben[7..0] bus is a time-multiplexed command/byte enable bus. During the address phase, this bus indicates the command; during the data phase, this bus indicates byte enables.
par	Tri-State	–	Parity. The par signal is even parity across the 32 least significant address/data bits and four least significant command/byte enable bits. In other words, the number of 1s on ad[31..0], cben[3..0], and par equal an even number. The parity of a data phase is presented on the bus on the clock following the data phase.
par64	Tri-State	–	Parity 64. The par64 signal is even parity across the 32 most significant address/data bits and the four most significant command/byte enable bits. In other words, the number of 1s on ad[63..32], cben[7..4], and par64 equal an even number. The parity of a data phase is presented on the bus on the clock following the data phase.
idsel	Input	High	Initialization device select. The idsel input is a chip select for configuration transactions.

Table 1. PCI Interface Signals (Part 2 of 3)			
Name	**Type**	**Polarity**	**Description**
`framen` *(1)*	STS	Low	Frame. The `framen` signal is an output from the current bus master that indicates the beginning and duration of a bus operation. When `framen` is initially asserted, the address and command signals are present on the `ad[63..0]` and `cben[7..0]` buses. The `framen` signal remains asserted during the data operation and is deasserted to identify the end of a transaction.
`req64n` *(1)*	STS	Low	Request 64-bit transfer. The `req64n` signal is an output from the current bus master and indicates that the master is requesting a 64-bit transaction. `req64n` has the same timing as `framen`.
`irdyn` *(1)*	STS	Low	Initiator ready. The `irdyn` signal is an output from a bus master to its target and indicates that the bus master can complete the current data transaction. In a write transaction, `irdyn` indicates that valid data is on the `ad[63..0]` bus. In a read transaction, `irdyn` indicates that the master is ready to accept the data on the `ad[63..0]` bus.
`devseln` *(1)*	STS	Low	Device select. Target asserts `devseln` to indicate that the target has decoded its own address and accepts the transaction.
`ack64n` *(1)*	STS	Low	Acknowledge 64-bit transfer. The target asserts `ack64n` to indicate that the target can transfer data using 64 bits. The `ack64n` has the same timing as `devseln`.
`trdyn` *(1)*	STS	Low	Target ready. The `trdyn` signal is a target output, indicating that the target can complete the current data transaction. In a read operation, `trdyn` indicates that the target is providing data on the `ad[63..0]` bus. In a write operation, `trdyn` indicates that the target is ready to accept data on the `ad[63..0]` bus.
`stopn` *(1)*	STS	Low	Stop. The `stopn` signal is a target device request that indicates to the bus master to terminate the current transaction. The `stopn` signal is used in conjunction with `trdyn` and `devseln` to indicate the type of termination initiated by the target.

Table 1. PCI Interface Signals (Part 3 of 3)			
Name	**Type**	**Polarity**	**Description**
perrn	STS	Low	Parity error. The perrn signal indicates a data parity error. The perrn signal is asserted one clock following the par and par64 signals or two clocks following a data phase with a parity error. The pci_c function asserts the perrn signal if it detects a parity error on the par or par64 signals and the perrn bit (bit 6) in the command register is set. The par64 signal is only evaluated during 64-bit transactions.
serrn	Open-Drain	Low	System error. The serrn signal indicates system error and address parity error. The pci_c function asserts serrn if a parity error is detected during an address phase and the serrn enable bit (bit 8) in the command register is set.
intan	Open-Drain	Low	Interrupt A. The intan signal is an active-low interrupt to the host and must be used for any single-function device requiring an interrupt capability.

Note:
(1) In the MegaCore function symbol, the signals are separated into two components: input and output. For example, framen has the input framen_in and the output framen_out. This separation of signals allows the use of devices that do not meet set-up times to implement a PCI interface. By driving the input part of one or more of these signals to a dedicated input pin and the output part to a regular I/O pin, allows devices that cannot meet set-up times to meet them. For more information on these devices, see the **readme** file provided with the MegaCore function.

Local Address, Data, Command & Byte Enable Signals

Table 2 summarizes the pci_c local interface signals for address, data, command, and byte enable signals.

Table 2. pci_c Local Address, Data, Command & Byte Enable Signals (Part 1 of 2)

Name	Type	Polarity	Description
l_adi[63..0]	Input	–	Local address/data input. This bus is a local-side time multiplexed address/data bus. During master transactions, the local side must provide the address on l_adi[63..0] when lm_adr_ackn is asserted. For 32-bit addressing, only the l_adi[31..0] signals are valid during the address phase. The l_adi[63..0] bus is driven active by the local-side device during PCI bus-initiated target read transactions or local-side initiated master write transactions. During 64-bit target read, 32-bit target read, or 64-bit master write transactions, data is transferred from the local side to the pci_c function using the entire bus. During 32-bit local-side initiated master write transactions, only the l_adi[31..0] bus is used to transfer data from the local side to the pci_c function.
l_cbeni[7..0]	Input	–	Local command/byte enable input. This bus is a local-side time multiplexed command/byte enable bus. During master transactions, the local side must provide the command on l_cbeni[3..0] when lm_adr_ackn is asserted. For 64-bit addressing, the local side must provide the dual address cycle (DAC) command (B"1101") on l_cbeni[3..0] and the transaction command on l_cbeni[7..4] when lm_adr_ackn is asserted. The local side drives the command with the same encoding as specified in the ***PCI Local Bus Specification, Revision 2.2***. The l_cbeni[7..0] bus is driven by the local-side device during master transactions. The local-master device drives byte enables on this bus during master transactions. The local-master device must provide the byte-enable value on l_cbeni[7..0] during the next clock after lm_adr_ackn is asserted. The pci_c function drives the byte-enable value from the local side to the PCI side and maintains the same byte-enable value for the entire transaction.

Table 2. pci_c Local Address, Data, Command & Byte Enable Signals (Part 2 of 2)

Name	Type	Polarity	Description
l_adro[63..0]	Output	–	Local address output. The l_adro[31..0] bus is driven by the pci_c function during target-read or write transactions. The PCI transaction address is valid on the local side until the pci_c target is in turn-around phase on the PCI bus. The l_adro[63..32] bus is driven with a valid address during a 64-bit addressing transaction, indicated when the lt_tsr[11] status signal is set. For more information on the local target status signals, please refer to Table 4.
l_dato[63..0]	Output	–	Local data output. The l_dato[63..0] bus is driven active by the pci_c function during PCI bus-initiated target write transaction or local side-initiated master read transaction. During 64-bit target write transactions and master read transactions, the pci_c function transfers data on the entire l_dato[63..0] bus. During 32-bit master read transactions, the pci_c function transfers data only on the l_dato[31..0]. During 32-bit target write transaction, the pci_c function drives the same data on both the l_dato[31..0] and l_dato[63..32] buses and, depending on the transaction address, the pci_c function will either assert l_ldat_ackn or l_hdat_ackn to indicate whether the low or high DWORD is valid.
l_beno[7..0]	Output	–	Local byte enable output. The l_beno[7..0] bus is driven by the pci_c function during target transactions. This bus holds the byte enable value during data transfers.
l_cmdo[3..0]	Output	–	Local command output. The l_cmdo[3..0] bus is driven by the pci_c function during target transactions. It has the bus command and the same timing as the l_adro[31..0] bus. The command is encoded as presented on the PCI bus.
l_ldat_ackn	Output	Low	Local low data acknowledge. The l_ldat_ackn output is used during target write and master read transactions. When asserted, it indicates that the next data transfer is on the least significant DWORD of the l_dato[63..0] bus. In other words, when l_ldat_ackn is asserted, valid data is presented on the l_dato[31..0] bus. The signals lm_ackn or lt_ackn must be used to qualify valid data.
l_hdat_ackn	Output	Low	Local high data acknowledge. The l_hdat_ackn output is used during target write and master read transactions. When asserted, it indicates that the next data transfer is on the most significant DWORD of l_dato[63..0] bus. In other words, when l_hdat_ackn is asserted, valid data is presented on the l_dato[63..32]. The signals lm_ackn or lt_ackn must be used to qualify valid data.

Target Local-Side Signals

Table 3 summarizes the target interface signals that provide the interface between the MegaCore function to the local-side peripheral device(s) during target transactions.

Table 3. Target Signals Connecting to the Local Side (Part 1 of 2)

Name	Type	Polarity	Description
lt_abortn	Input	Low	Local target abort request. The local side should assert this signal requesting the pci_c function to issue a target abort to the PCI master. The local side should request an abort when it has encountered a fatal error and cannot complete the current transaction.
lt_discn	Input	Low	Local target disconnect request. The lt_discn input requests the pci_c function to issue a retry or a disconnect. The pci_c function issues a retry or disconnect depending on when the signal is asserted during a transaction. The PCI bus specification requires that a PCI target issues a disconnect whenever the transaction exceeds its memory space. When using the pci_c function, the local side is responsible for asserting lt_discn if the transaction crosses the pci_c memory space.
lt_rdyn	Input	Low	Local target ready. The local side asserts lt_rdyn to indicate a valid data input during target read, or ready to accept data input during a target write. During a target read, lt_rdyn de-assertion suspends the current transfer, i.e., a wait state is inserted by the local side. During a target write, an inactive lt_rdyn signal directs the pci_c function to insert wait states on the PCI bus. The only time the function inserts wait states during a burst is when lt_rdyn inserts wait states on the local side. lt_rdyn is sampled one clock before actual data is transferred on the local side.
lt_framen	Output	Low	Local target frame request. The lt_framen output is asserted while the pci_c function is requesting access to the local side. It is asserted one clock before the function asserts devseln and it is released after the last data phase of the transaction is transferred to/from the local side.

Table 3. Target Signals Connecting to the Local Side (Part 2 of 2)

Name	Type	Polarity	Description
lt_ackn	Output	Low	Local target acknowledge. The pci_c function asserts lt_ackn to indicate valid data output during a target write, or ready to accept data during a target read. During a target read, an inactive lt_ackn indicates that the function is not ready to accept data and local logic should hold off the bursting operation. During a target write, lt_ackn de-assertion suspends the current transfer, i.e., a wait state is inserted by the PCI master. The lt_ackn signal is only inactive during a burst when the PCI bus master inserts wait states.
lt_dxfrn	Output	Low	Local target data transfer. The pci_c function asserts the lt_dxfrn signal when a data transfer on the local side is successful during a target transaction.
lt_tsr[11..0]	Output	–	Local target transaction status register. The lt_tsr[11..0] bus carries several signals which can be monitored for the transaction status. See Table 4.
lirqn	Input	Low	Local interrupt request. The local-side peripheral device asserts lirqn to signal a PCI bus interrupt. Asserting this signal forces the pci_c function to assert the intan signal for as long as the lirqn signal is asserted.
cache[7..0]	Output	–	Cache registers output. The cache[7..0] bus is the same as the configuration space cache register. The local-side logic uses this signal to provide support for cache commands.
cmd_reg[5..0]	Output	–	Command register output. The cmd_reg[5..0] bus drives the important signals of the configuration space command register to the local side. See Table 5.
stat_reg[5..0]	Output	–	Status register output. The stat_reg[5..0] bus drives the important signals of the configuration space status register to the local side. See Table 6.

Table 4 shows definitions for the local target transaction status register outputs.

Table 4. Local Target Transaction Status Register Bit Definition

Bit Number	Bit Name	Description
5..0	bar_hit[5..0]	Base address register hit. Asserting bar_hit[5..0] indicates that the PCI address matches that of a base address register and pci_c has claimed the transaction. Each bit in the bar_hit[5..0] bus is used for the corresponding base address register, therefore, bar_hit[0] is used for BAR0. bar_hit[5..0] have the same timing as the lt_framen signal. When a 64-bit base address register is used, both bar_hit[0] and bar_hit[1] are asserted to indicate that pci_c has claimed the transaction.
6	exp_rom_hit	Expansion ROM register hit. pci_c asserts this signal when the transaction address matches the address in the expansion ROM BAR.
7	trans64	64-bit target transaction. pci_c asserts this signal when the current transaction is 64 bits. If a transaction is active and this signal is low, the current transaction is 32 bits.
8	targ_access	Target access. pci_c asserts this signal when PCI target access to the pci_c target is active.
9	burst_trans	Burst transaction. When asserted, this signal indicates that the current target transaction is a burst. This signal is detected if both framen and irdyn signals are asserted at the same time during the first data phase.
10	pxfr	PCI transfer. This signal is asserted to indicate that there was a successful data transfer on the PCI side during the previous clock cycle.
11	dac	Dual address cycle. When asserted, this signal indicates that the current transaction is using a dual address cycle on the pci_c target.

Table 5 shows definitions for configuration output bus bits.

Table 5. Configuration Output Bus Bit Definition

Bit Number	Bit Name	Description
0	io_ena	I/O accesses enable. Bit 0 of command register.
1	mem_ema	Memory access enable. Bit 1 of command register
2	mstr_ena	Master enable. Bit 2 of command register.
3	mwi_ena	Memory write and invalidate enable. Bit 4 of command register.
4	perr_ena	Parity error response enable. Command register bit 6.
5	serr_ena	System error response enable. Command register bit 8.

Table 6 shows definitions for local target transaction status register bits.

Table 6. Local Target Transaction Status Register Bit Definition

Bit Number	Bit Name	Description
0	perr_rep	Parity error reported, Status register bit 8.
1	tabort_sig	Target abort signaled. Status register bit 11.
2	tabort_rcvd	Target abort received. Status register bit 12.
3	mabort_rcvd	Master abort received. Status register bit 13.
4	serr_sig	Signaled system error. Status register bit 14.
5	perr_det	Parity error detected. Status register bit 15.

Master Local-Side Signals

Table 7 summarizes the pci_c master interface signals that provide the interface between the pci_c MegaCore function and the local-side peripheral device(s) during master transactions.

Table 7. pci_c Master Signals Interfacing to the Local Side (Part 1 of 2)

Name	Type	Polarity	Description
lm_req32n	Input	Low	Local master request 32-bit data transaction. The local side asserts this signal to request ownership of the PCI bus for a 32-bit master transaction. To request a master transaction, it is sufficient for the local-side device to assert lm_req32n for one clock cycle. When requesting a 32-bit transaction, only l_dati[31..0] for a master write transaction or l_dato[31..0] for a master read transaction, are valid.
lm_req64n	Input	Low	Local master request 64-bit data transaction. The local side asserts this signal to request ownership of the PCI bus for a 64-bit master transaction. To request a master transaction, it is sufficient for the local side device to assert lm_req64n for one clock. When requesting a 64-bit data transaction, pci_c requests a 64-bit PCI transaction. When the target does not assert its ack64n signal, the transaction will be 32 bits. In a 64-bit master write transaction where the target does not assert its ack64n signal, pci_c automatically accepts 64-bit data on the local side and multiplexes the data appropriately to 32 bits on the PCI side. When the local side requests 64-bit PCI transactions, it must ensure that the address is at a quad WORD boundary.
lm_lastn	Input	Low	Local master last. This signal is driven by the local side to request that the pci_c MegaCore master interface ends the current transaction. When the local side asserts this signal, the pci_c MegaCore master interface deasserts framen as soon as possible, and asserts irdyn to indicate that the last data phase has begun. The local side can assert this signal for one clock any time during the master transaction.

Table 7. pci_c Master Signals Interfacing to the Local Side (Part 2 of 2)

Name	Type	Polarity	Description
lm_rdyn	Input	Low	Local master ready. The local side asserts the lm_rdyn signal to indicate a valid data input during a master write, or ready to accept data during a master read. During a master write, the lm_rdyn signal de-assertion suspends the current transfer, i.e., wait state is inserted by the local side. During a master read, an inactive lm_rdyn signal directs pci_c to insert wait states on the PCI bus. The only time pci_c inserts wait states during a burst is when the lm_rdyn signal inserts wait states on the local side. The lm_rdyn signal is sampled one clock before actual data is transferred on the local side.
lm_adr_ackn	Output	Low	Local master address acknowledge. pci_c asserts the lm_adr_ackn signal to the local side to acknowledge the requested master transaction. During the same clock cycle when lm_adr_ackn is asserted low, the local side must provide the transaction address on the l_adi[31..0] bus and the transaction command on the l_cmdi[3..0] bus. The local side cannot delay pci_c by registering the address on the l_adi[31..0] bus.
lm_ackn	Output	Low	Local master acknowledge. pci_c asserts the lm_ackn signal to indicate valid data output during a master read, or ready to accept data during a master write. During a master write, an inactive lm_ackn signal indicates that pci_c is not ready to accept data, and local logic should hold off the bursting operation. During a master read, the lm_ackn signal de-assertion suspends the current transfer, i.e., a wait state is inserted by the PCI target. The only time the lm_ackn signal goes inactive during a burst is when the PCI bus target inserts wait states.
lm_dxfrn	Output	Low	Local master data transfer. pci_c asserts this signal when a data transfer on the local side is successful during a master transaction.
lm_tsr[9..0]	Output	–	Local master transaction status register bus. These signals inform the local interface the progress of the transaction. See Table 8 for a detailed description of the bits in this bus.

Table 8 shows definitions for the local master transaction status register outputs.

Table 8. pci_c Local Master Transaction Status Register Bit Definition

Bit Number	Bit Name	Description
0	`req`	Request. This signal indicates that the `pci_c` function is requesting mastership of the PCI bus, i.e., it is asserting its `reqn` signal.
1	`gnt`	Grant. This signal is active after the `pci_c` function has detected that `gntn` is asserted.
2	`adr_phase`	Address phase. This signal is active during a PCI address phase where `pci_c` is the bus master.
3	`dat_xfr`	Data transfer. This signal is active while the `pci_c` function is in data transfer mode. The signal is active after the address phase and remains active until the turn-around state begins.
4	`lat_exp`	Latency timer expired. This signal indicates that `pci_c` terminated the master transaction because the latency timer counter expired.
5	`retry`	Retry detected. This signal indicates that the `pci_c` function terminated the master transaction because the target issued a retry. Per the PCI specification, a transaction that ended in a retry must be retried at a later time.
6	`disc_wod`	Disconnect without data detected. This signal indicates that the `pci_c` signal terminated the master transaction because the target issued a disconnect without data.
7	`disc_wd`	Disconnect with data detected. This signal indicates that `pci_c` terminated the master transaction because the target issued a disconnect with data.
8	`dat_phase`	Data phase. This signal indicates that a successful data transfer has occurred on the PCI side in the prior clock cycle. This signal can be used by the local side to keep track of how much data was actually transferred on the PCI side.
9	`trans64`	64-bit transaction. This signal indicates that the target claiming the transaction has asserted its `ack64n` signal.

Parameters

The pci_c MegaCore configuration parameters set the PCI bus configuration registers, and the TARGET_DEVICE parameter optimizes the logic resources used in your target FLEX device. All configuration parameters except for NUMBER_OF_BARS and BAR0 through BAR5 set read-only PCI configuration registers; these registers are device identification registers. See “Configuration Registers” on page 56 for more information on these registers. Table 9 describes the PCI MegaCore function parameters.

Table 9. pci_c MegaCore Function Parameters (Part 1 of 3)

Name	Format	Default Value	Description
BAR0 *(1)*	Hexadecimal	H"FFF00000"	Base address register zero. When a 64-bit base address register is used, BAR0 contains the lower 32-bit address. For more information, refer to “Base Address Registers” on page 63.
BAR1 *(1)*	Hexadecimal	H"FFF00000"	Base address register one. When a 64-bit base address register is used, BAR1 contains the upper 32-bit address. For more information, refer to “Base Address Registers” on page 63.
BAR2 *(1)*	Hexadecimal	H"FFF00000"	Base address register two.
BAR3 *(1)*	Hexadecimal	H"FFF00000"	Base address register three.
BAR4 *(1)*	Hexadecimal	H"FFF00000"	Base address register four.
BAR5 *(1)*	Hexadecimal	H"FFF00000"	Base address register five.
CAP_LIST_ENA	String	"NO"	Capabilities list enable. When set to "YES", this parameter enables the capabilities list pointer register and sets bit 4 of the status register.
CAP_PTR	Hexadecimal	H"40"	Capabilities list pointer register. This is 8-bit value sets the capabilities list pointer register.
CLASS_CODE	Hexadecimal	H"FF0000"	Class code register. This parameter is a 24-bit hexadecimal value that sets the class code register in the pci_c configuration space. The value entered for this parameter must be a valid PCI SIG-assigned class code register value.
DEVICE_ID	Hexadecimal	H"0004"	Device ID register. This parameter is a 16-bit hexadecimal value that sets the device ID register in the pci_c configuration space. Any value can be entered for this parameter.

Table 9. pci_c MegaCore Function Parameters (Part 2 of 3)			
Name	**Format**	**Default Value**	**Description**
EXP_ROM_BAR	String	H"FF000000"	Expansion ROM. This value controls the number of bits in the expansion ROM BAR that are read/write and will be decoded during a memory transaction.
EXP_ROM_ENA	String	"NO"	Expansion ROM enable. This parameter controls whether to activate the expansion ROM base address register or not. This parameter must be set to "YES" for the expansion ROM BAR to be enabled. If this parameter is set to "NO", the EXP_ROM_BAR parameter is ignored.
HOST_BRIDGE_ENA *(2)*	String	"NO"	This parameter permanently enables the master capability in the pci_c function to be used in host bridge applications, which allows the pci_c function to generate the required configuration transactions during power-up. If the pci_c function is used as a host bridge, the local-side application must be able to perform master transactions at power-up. The pci_c MegaCore function can generate configuration cycles for other PCI bus agents, including its own target.
INTERNAL_ARBITER *(2)*	String	"NO"	This parameter allows reqn and gntn to be used in internal arbiter logic without requiring external device pins. If a FLEX device is used to implement the pci_c MegaCore function and is also used to implement a PCI bus arbiter, the reqn signal should feed internal logic and gntn should be driven by internal logic without using actual device pins. If this parameter is set to "YES," the tri-state buffer on the reqn signal is removed, allowing an arbiter to be implemented without using device pins for the reqn and gntn signals.
MAX_LATENCY *(2)*	Hexadecimal	H"00"	Maximum latency register. This parameter is an 8-bit hexadecimal value that sets the maximum latency register in the pci_c configuration space. This parameter must be set according to the guidelines in the PCI specifications.

Table 9. pci_c MegaCore Function Parameters (Part 3 of 3)

Name	Format	Default Value	Description
MIN_GRANT *(2)*	Hexadecimal	H"00"	Minimum grant register. This parameter is an 8-bit hexadecimal value that sets the minimum grant register in the pci_c configuration space. This parameter must be set according to the guidelines in the PCI specification.
NUMBER_OF_BARS	Decimal	1	Number of base address registers. Only the logic that is required to implement the number of BARs specified by this parameter is used—i.e., BARs that are not used do not take up additional logic resources. The pci_c MegaCore function sequentially instantiates the number of BARs specified by this parameter starting with BAR0.
REVISION_ID	Hexadecimal	H"01"	Revision ID register. This parameter is an 8-bit hexadecimal value that sets the revision ID register in the pci_c configuration space.
PCI_66MHZ_CAPABLE	Hexadecimal	"YES"	PCI 66-MHz capable. When set to "YES", this parameter sets bit 5 of the status register to enable 66-MHz operation.
SUBSYSTEM_ID	Hexadecimal	H"0000"	Subsystem ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem ID register in the pci_c configuration space. Any value can be entered for this parameter.
SUBSYSTEM_VEND_ID	Hexadecimal	H"0000"	Subsystem vendor ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem vendor ID register in the pci_c configuration space. The value for this parameter must be a valid PCI SIG-assigned vender ID number.
TARGET_DEVICE *(2)*	String	"EPF10K50EFC484"	This parameter should be set to your targeted Altera FLEX device for logic and performance optimization.
VEND_ID	Hexadecimal	H"1172"	Device vendor ID register. This parameter is a 16-bit hexadecimal value that sets the vendor ID register in the pci_c configuration space. The value for this parameter can be the Altera vendor ID (1172 Hex) or any other PCI SIG-assigned vendor ID number.

Notes:

(1) The BAR0 through BAR5 parameters control the options of the corresponding BAR instantiated in the PCI MegaCore function. Use BAR0 through BAR5 for I/O and 32-bit memory space. However, if you use a 64-bit BAR, you must use BAR0 and BAR1. Consequently, BAR2 through BAR5 can still be used for I/O and 32-bit memory space.

(2) For a listing of the supported Altera FLEX 10KE devices, refer to the **readme** file of the pci_c MegaCore function.

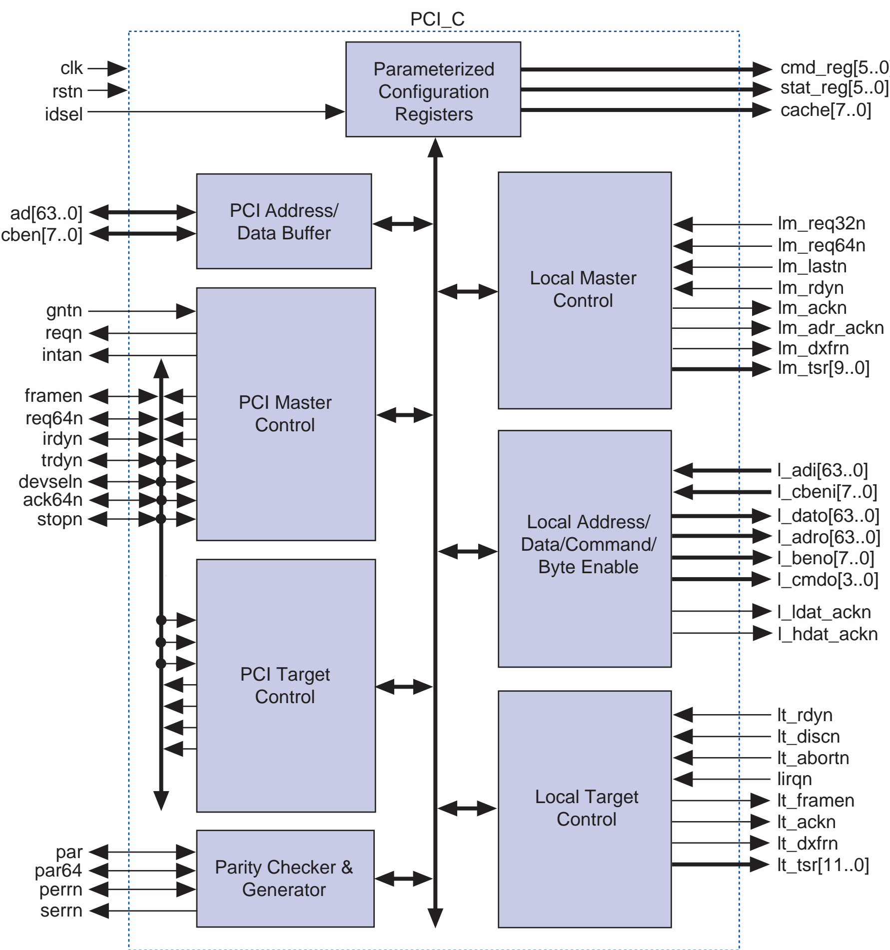
Functional Description

This section provides a general overview of pci_c operation. The pci_c function consists of three main elements:

- A parameterized PCI bus configuration register space
- Target interface control logic
- Master interface control logic

Figure 2 shows the pci_c functional block diagram.

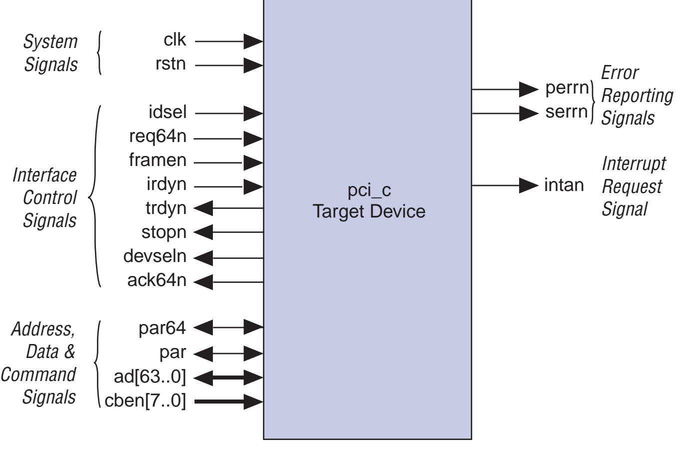
Figure 2. pci_c Functional Block Diagram



Target Device Signals & Signal Assertion

Figure 3 illustrates the signal directions for a PCI device connecting to the PCI bus in target mode. These signals apply to the `pci_c` function when it is operating in target mode. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the MegaCore function operating as a target on the PCI bus.

Figure 3. Target Device Signals



A 32-bit target sequence begins when the PCI master device asserts `framen` and drives the address and the command on the PCI bus. If the address matches one of the BARs in the MegaCore function, it asserts `devseln` to claim the transaction. The master then asserts `irdyn` to indicate to the target device that:

- For a read operation, the master device can complete a data transfer.
- For a write operation, valid data is on the `ad[31..0]` bus.

The MegaCore function drives the control signals `devseln`, `trdyn`, and `stopn` to indicate one of the following conditions to the PCI master:

- The MegaCore function has decoded a valid address for one of its BARs and it accepts the transactions (assert `devseln`).
- The MegaCore function is ready for the data transfer (assert `trdyn`). When both `trdyn` and `irdyn` are active, a data word is clocked from the sending to the receiving device.
- The master device should retry the current transaction.
- The master device should stop the current transaction.
- The master device should abort the current transaction.

Table 10 shows the control signal combinations possible on the PCI bus during a PCI transaction. The pci_c function processes the PCI signal assertion from the local side. Therefore, the pci_c function only drives the control signals per the *PCI Local Bus Specification, Revision 2.2*. The local-side application can force retry, disconnect, abort, successful data transfer, and target wait state cycles to appear on the PCI bus by driving the lt_rdyn, lt_discn, and lt_abortn signals to certain values. See "Target Transaction Terminations" on page 99 for more details.

The pci_c function accepts either 32-bit transactions or 64-bit transactions on the PCI side. In both cases, pci_c behaves as a 64-bit agent on the local side. A 64-Bit transaction differs from a 32-bit transaction as follows:

- In addition to asserting the framen signal, the PCI master asserts the req64n signal during the address phase informing the target device that it is requesting a 64-bit transaction.
- When the target device accepts the 64-bit transaction, it asserts ack64n in addition to devseln to inform the master device that it is accepting the 64-bit transaction.
- In a 64-bit transaction, the req64n signal behaves the same as the framen signal and the ack64n signal behaves the same as devseln. During data phases, data is driven over the ad[63..0] bus and byte enables are driven over the cben[7..0] bus. Additionally, parity for ad[63..32] and cben[7..4] is presented over the par64n signal.

Table 10. Control Signal Combination Transfer

Type	devseln	trdyn	stopn	irdyn
Claim transaction	Assert	Don't care	Don't care	Don't care
Retry (1)	Assert	De-Assert	Assert	Don't care
Disconnect with data	Assert	Assert	Assert	Don't care
Disconnect without data	Assert	De-assert	Assert	Don't care
Abort (2)	De-assert	De-assert	Assert	Don't care
Successful transfer	Assert	Assert	De-assert	Assert
Target wait state	Assert	De-assert	De-assert	Assert
Master wait state	Assert	Assert	De-assert	De-assert

Notes:
(1) A retry occurs before the first data phase.
(2) A device must assert the devseln signal for at least one clock before it signals an abort.

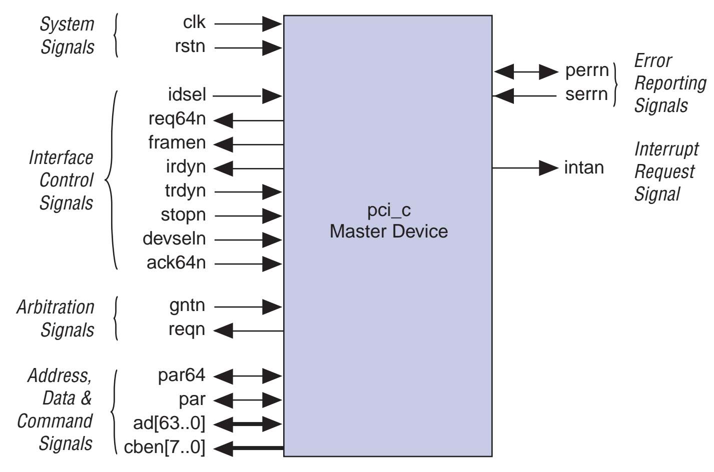
3 MegaCore Overview

The pci_c function supports unlimited burst access cycles. Therefore, pci_c can achieve a throughput from 132 Mbytes per second (for 32-bit, 33 MHz transactions) up to 528 Mbytes per second (for 64-bit, 66 MHz transactions). However, the *PCI Local Bus Specification, Revision* 2.2 does not recommend bursting beyond 16 data cycles because of the latency of other devices that share the bus. Designers should be aware of the trade-off between bandwidth and increased latency.

Master Device Signals & Signal Assertion

Figure 4 illustrates the PCI-compliant master device signals that connect to the PCI bus. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the pci_c function operating as a master on the PCI bus.

Figure 4. pci_c Master Device Signals



A 32-bit pci_c master sequence begins when the local side asserts lm_reqn32n to request mastership of the PCI bus. The pci_c function then asserts reqn to request ownership of the PCI bus. After receiving gntn from the PCI bus arbiter and after the bus idle state is detected, the pci_c function initiates the address phase by asserting framen, driving the PCI address on ad[31..0], and driving the bus command on cben[3..0] for one clock cycle.

For 64-bit addressing, the pci_c master generates a DAC. On the first address phase, the pci_c function drives the lower 32-bit PCI address on ad[31..0], the upper 32-bit PCI address on ad[63..32], the DAC command on cben[3..0], and the transaction command on cben[7..4]. On the second address phase, the pci_c function drives the upper 32-bit PCI address on ad[63..0] and the transaction command on cben[7..0].

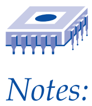
When the pci_c function is ready to present or accept data on the bus, it asserts irdyn. At this point, the pci_c master logic monitors the control signals driven by the target device. A target device is determined by the decoding of the address and command signals presented on the PCI bus during the address phase of the transaction. The target device drives the control signals devseln, trdyn, and stopn to indicate one of the following conditions:

- The data transaction has been decoded and accepted.
- The target device is ready for the data operation. When both trdyn and irdyn are active, a data word is clocked from the sending to the receiving device.
- The master device should retry the current transaction.
- The master device should stop the current transaction.
- The master device should abort the current transaction.

Table 10 on page 49 shows the possible control signal combinations on the PCI bus during a transaction. The pci_c function signals that it is ready to present or accept data on the bus by asserting irdyn. At this point, the pci_c master logic monitors the control signals driven by the target device and asserts its control signals appropriately. The local-side application can use the lm_tsr[9..0] signals to monitor the progress of the transaction. The master transaction can be terminated normally or abnormally. The local side signals a normal transaction termination by asserting the lm_lastn signal. The abnormal termination can be signaled by the target, master abort, or latency timer expiration. See "Abnormal Master Transaction Termination" on page 141 for more details.

In addition to single-cycle and burst 32-bit transactions, the local side master can request 64-bit transactions by asserting the lm_req64n signal. In 64-bit transactions, the pci_c function behaves the same a 32-bit transaction except for asserting the req64n signal with the same timing as the framen signal. Additionally, the pci_c function treats the local side as 64 bits when it requests 64-bit transactions and when the target device accepts 64-bit transactions by asserting the ack64n signal. See "Master Mode Operation" on page 108 for more information on 64-bit master transactions.

See note at the top of this page.



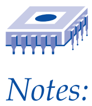
Notes:

Specifications

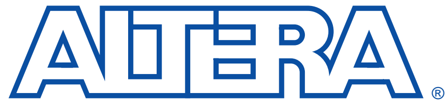
Contents

June 1999

PCI Bus Commands 55
Configuration Registers 56
Vendor ID Register 58
Device ID Register 59
Command Register 59
Status Register 60
Revision ID Register 61
Class Code Register 62
Cache Line Size Register 62
Latency Timer Register 62
Header Type Register 63
Base Address Registers 63
Subsystem Vendor ID Register 66
Subsystem ID Register 67
Expansion ROM Base Address Register 67
Capabilities Pointer 68
Interrupt Line Register 68
Interrupt Pin Register 68
Minimum Grant Register 69
Maximum Latency Register 69
Target Mode Operation 70
64-Bit Target Read Transactions 71
32-Bit Target Read Transactions 80
64-Bit Target Write Transactions 85
32-Bit Target Write Transactions 93
Target Transaction Terminations 99
Master Mode Operation 108
64-Bit Master Read Transactions 110
32-Bit Master Write Transactions 135
Abnormal Master Transaction Termination 141
64-Bit Addressing, Dual Address Cycle (DAC) 143
Target Mode Operation 143
Master Mode Operation 145



Notes:



Specifications

June 1999, ver. 1.1

This section describes the specifications of the `pci_c` MegaCore™ function, including the supported peripheral component interconnect (PCI) bus commands and configuration registers and the clock cycle sequence for both target and master read/write transactions.

PCI Bus Commands

Table 1 shows the PCI bus commands that can be initiated or responded to by the `pci_c` MegaCore function.

Table 1. PCI Bus Command Support Summary

cben[3..0] Value	Bus Command Cycle	Master	Target
0000	Interrupt acknowledge	Ignored	Ignored
0001	Special cycle	Ignored	Ignored
0010	I/O read	Yes	Yes
0011	I/O write	Yes	Yes
0100	Reserved	Ignored	Ignored
0101	Reserved	Ignored	Ignored
0110	Memory read	Yes	Yes
0111	Memory write	Yes	Yes
1000	Reserved	Ignored	Ignored
1001	Reserved	Ignored	Ignored
1010	Configuration read	Yes	Yes
1011	Configuration write	Yes	Yes
1100	Memory read multiple *(1)*	Yes	Yes
1101	Dual address cycle (DAC)	Yes	Yes
1110	Memory read line *(1)*	Yes	Yes
1111	Memory write and invalidate *(1)*	Yes	Yes

Note:
(1) The memory read multiple and memory read line commands are treated as memory reads. The memory write and invalidate command is treated as a memory write. The local side sees the exact command on the `l_cbeni[3..0]` bus with the encoding shown in Table 1.

During the address phase of a transaction, the `cben[3..0]` bus is used to indicate the transaction type. See Table 1.

The `pci_c` function responds to standard memory read/write, cache memory read/write, I/O read/write, and configuration read/write commands. The bus commands are discussed in greater detail in "Target Mode Operation" on page 70 and "Master Mode Operation" on page 108.

In master mode, the `pci_c` function can initiate transactions of standard memory read/write, cache memory read/write, I/O read/write, and configuration read/write commands. Per the PCI specification, the master must keep track of the number of words that are transferred and can only end the transaction at cache line boundaries during MRL and MWI commands. It is the responsibility of the local-side interface to ensure that this requirement is not violated. Additionally, it is the responsibility of the local-side interface to ensure that proper address and byte enable combinations are used during I/O read/write cycles.

Configuration Registers

Each logical PCI bus device includes a block of 64 configuration DWORDS reserved for the implementation of its configuration registers. The format of the first 16 DWORDS is defined by the PCI Special Interest Group (PCI SIG) ***PCI Local Bus Specification, Revision*** 2.2 and the *Compliance Checklist, Revision* 2.1. These specifications define two header formats, type one and type zero. Header type one is used for PCI-to-PCI bridges; header type zero is used for all other devices, including the `pci_c` function.

Table 2 shows the defined 64-byte configuration space. The registers within this range are used to identify the device, control PCI bus functions, and provide PCI bus status. The shaded areas indicate registers that are supported by the `pci_c` function.

Table 2. PCI Bus Configuration Registers

Address	Byte			
	3	2	1	0
00H	Device ID		Vendor ID	
04H	Status Register		Command Register	
08H	Class Code			Revision ID
0CH	BIST	Header Type	Latency Timer	Cache Line Size
10H	Base Address Register 0			
14H	Base Address Register 1			
18H	Base Address Register 2			
1CH	Base Address Register 3			
20H	Base Address Register 4			
24H	Base Address Register 5			
28H	Card Bus CIS Pointer			
2CH	Subsystem ID		Subsystem Vendor ID	
30H	Expansion ROM Base Address Register			
34H	Reserved			Capabilities Pointer
38H	Reserved			
3CH	Maximum Latency	Minimum Grant	Interrupt Pin	Interrupt Line

Table 3 summarizes the pci_c-supported configuration registers address map. Unused registers produce a zero when read, and they ignore a write operation. Read/write refers to the status at runtime, i.e., from the perspective of other PCI bus agents. Designers can set some of the read-only registers when creating a custom PCI design by setting the pci_c function parameters. For example, the designer can change the device ID register value from the default value by changing the DEVICE_ID parameter in the MAX+PLUS® II software. The specified default state is defined as the state of the register when the PCI bus is reset.

Table 3. Supported Configuration Registers Address Map

Address Offset (Hex)	Range Reserved (Hex)	Bytes Used/ Reserved	Read/Write	Mnemonic	Register Name
00	00-01	2/2	Read	ven_id	Vendor ID
02	02-03	2/2	Read	dev_id	Device ID
04	04-05	2/2	Read/write	comd	Command
06	06-07	2/2	Read/write	status	Status
08	08-08	1/1	Read	rev_id	Revision ID
09	09-0B	3/3	Read	class	Class code
0C	0C-0C	1/1	Read/write	cache	Cache line size
0D	0D-0D	1/1	Read/write	lat_tmr	Latency timer
0E	0E-0E	1/1	Read	header	Header type
10	10-13	4/4	Read/write	bar0	Base address register zero
14	14-17	4/4	Read/write	bar1	Base address register one
18	18-1B	4/4	Read/write	bar2	Base address register two
1C	1C-1F	4/4	Read/write	bar3	Base address register three
20	20-23	4/4	Read/write	bar4	Base address register four
24	24-27	4/4	Read/write	bar5	Base address register five
2C	2C-2D	2/2	Read	sub_ven_id	Subsystem vendor ID
2E	2E-2F	2/2	Read	sub_id	Subsystem ID
30H	30-33	4/4	Read/write	exp_rom_bar	Expansion ROM BAR
34H	34-35	1/1	Read	cap_ptr	Capabilities pointer
3C	3C-3C	1/1	Read/write	int_ln	Interrupt line
3D	3D-3D	1/1	Read	int_pin	Interrupt pin
3E	3E-3E	1/1	Read	min_gnt	Minimum grant
3F	3F-3F	1/1	Read	max_lat	Maximum latency

Vendor ID Register

Vendor ID is a 16-bit read-only register that identifies the manufacturer of the device. The value of this register is assigned by the PCI SIG; the default value of this register is the Altera® vendor ID value, which is 1172 hex. However, by setting the VEND_ID parameter, designers can change the value of the vendor ID register to their PCI SIG-assigned vendor ID value. See Table 4.

Table 4. Vendor ID Register Format

Data Bit	Mnemonic	Read/Write	Definition
15..0	vendor_id	Read	PCI vendor ID

Device ID Register

Device ID is a 16-bit read-only register that identifies the device type. The value of this register is assigned by the manufacturer. The default value of the device ID register is 0004 hex. Designers can change the value of the device ID register by setting the parameter DEVICE_ID. See Table 5.

Table 5. Device ID Register Format

Data Bit	Mnemonic	Read/Write	Definition
15..0	device_id	Read	Device ID

Command Register

Command is a 16-bit read/write register that provides basic control over the ability of the pci_c function to respond to the PCI bus and/or access it. See Table 6.

Table 6. Command Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	io_ena	Read/write	Read/write to I/O access enable.
1	mem_ena	Read/write	Memory access enable. When high, mem_ena lets the pci_c function respond to the PCI bus memory accesses as a target.
2	mstr_ena	Read/write	Master enable. When high, mstr_ena allows the pci_c function to acquire mastership of the PCI bus.
3	Unused	–	–
4	mwi_ena	Read/write	Memory write and invalidate enable. This bit controls whether the master may generate a MWI command. Although the pci_c function implements this bit, it is ignored. The local side must ensure that the mwi_ena output is high before it requests a master transaction using the MWI command.
5	Unused	–	–
6	perr_ena	Read/write	Parity error enable. When high, perr_ena enables the pci_c function to report parity errors via the perrn output.
7	Unused	–	–
8	serr_ena	Read/write	System error enable. When high, serr_ena allows the pci_c function to report address parity errors via the serrn output. However, to signal a system error, the perr_ena bit must also be high.
15..9	Unused	–	–

Status Register

Status is a 16-bit register that provides the status of bus-related events. Read transactions from the status register behave normally. However, write transactions are different from typical write transactions because bits in the status register can be cleared but not set. A bit in the status register is cleared by writing a logic one to that bit. For example, writing the value 4000 hex to the status register clears bit 14 and leaves the rest of the bits unchanged. The default value of the status register is 0400 hex. See Table 7.

Table 7. Status Register Format (Part 1 of 2)

Data Bit	Mnemonic	Read/Write	Definition
3..0	Unused	–	Reserved.
4	cap_list_ena	Read	Capabilities list enable. This bit is read only and is set by the user by setting the CAP_LIST_ENA parameter to "YES". When set, this bit enables the capabilities list pointer register at offset 34 hex. See "Capabilities Pointer" on page 68 for more details.
5	pci_66mhz_capable	Read	PCI 66 MHz capable. When set, pci_66mhz_capable indicates that the PCI device is capable of running at 66 MHz. The pci_c MegaCore function can run at either 66 MHz or 33 MHz depending on the device used. You can set this bit to one by setting the PCI_66MHZ_CAPABLE parameter to "YES".
7..6	Unused	–	Reserved.
8	dat_par_rep	Read/write	Data parity reported. When high, dat_par_rep indicates that during a read transaction the pci_c function asserted the perrn output as a master device, or that during a write transaction the perrn output was asserted as a target device. This bit is high only when the perr_ena bit (bit 6 of the command register) is also high. This signal is driven to the local side on the stat_reg[0] output.
10..9	devsel_tim	Read	Device select timing. The devsel_tim bits indicate target access timing of the pci_c function via the devseln output. The pci_c function is designed to be a slow target device, i.e., devsel_tim = B"10".
11	tabort_sig	Read/write	Target abort signaled. This bit is set when a local peripheral device terminates a transaction. The pci_c function automatically sets this bit if it issued a target abort after the local side asserted lt_abortn. This bit is driven to the local side on the stat_reg[1] output.

Table 7. Status Register Format (Part 2 of 2)			
Data Bit	Mnemonic	Read/Write	Definition
12	tar_abrt_rec	Read/write	Target abort. When high, tar_abrt_rec indicates that the pci_c function in master mode has detected a target abort from the current target device. This bit is driven to the local side on the stat_reg[2] output.
13	mstr_abrt	Read/write	Master abort. When high, mstr_abrt indicates that the pci_c function in master mode has terminated the current transaction with a master abort. This bit is driven to the local side on the stat_reg[3] output.
14	serr_set	Read/write	Signaled system error. When high, serr_set indicates that the pci_c function drove the serrn output active, i.e., an address phase parity error has occurred. The pci_c function signals a system error only if an address phase parity error was detected and serr_ena was set. This signal is driven to the local side on the stat_reg[4] output.
15	det_par_err	Read/write	Detected parity error. When high, det_par_err indicates that the pci_c function detected either an address or data parity error. Even if parity error reporting is disabled (via perr_ena), the pci_c function sets the det_par_err bit. This signal is driven to the local side on the stat_reg[5] output.

Revision ID Register

Revision ID is an 8-bit read-only register that identifies the revision number of the device. The value of this register is assigned by the manufacturer (e.g., Altera for the pci_c function). For Altera PCI MegaCore functions, the default value of the revision ID register is the revision number of the pci_c function. See Table 8. Designers can change the value of the revision ID register by setting the REVISION_ID parameter.

Table 8. Revision ID Register Format			
Data Bit	Mnemonic	Read/Write	Definition
7..0	rev_id	Read	PCI revision ID

Class Code Register

Class code is a 24-bit read-only register divided into three sub-registers: base class, sub-class, and programming interface. Refer to the *PCI Local Bus Specification, Revision 2.2* for detailed bit information. The default value of the class code register is FF0000 hex. Designers can change the value by setting the CLASS_CODE parameter. See Table 9.

Table 9. Class Code Register Format

Data Bit	Mnemonic	Read/Write	Definition
23..0	class	Read	Class code

Cache Line Size Register

The cache line size register specifies the system cache line size in DWORDS. This read/write register is written by system software at power-up. The value in this register is driven to the local side on the cache[7..0] bus. The local side must use this value when using the memory write and invalidate command in master mode. See Table 10.

Table 10. Cache Line Size Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	cache	Read/write	Cache line size

Latency Timer Register

The latency timer register is an 8-bit register with bits 2, 1, and 0 tied to ground. The register defines the maximum amount of time, in PCI bus clock cycles, that the pci_c function can retain ownership of the PCI bus. After initiating a transaction, the pci_c function decrements its latency timer by one on the rising edge of each clock. The default value of the latency timer register is 00 hex. See Table 11.

Table 11. Latency Timer Register Format

Data Bit	Mnemonic	Read/Write	Definition
2..0	lat_tmr	Read	Latency timer register
7..3	lat_tmr	Read/write	Latency timer register

Header Type Register

Header type is an 8-bit read-only register that identifies the `pci_c` function as a single-function device. The default value of the header type register is `00` hex. See Table 12.

Table 12. Header Type Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	`header`	Read	PCI header type

Base Address Registers

The `pci_c` function supports up to six BARs. Each base address register (BAR*n*) has identical attributes. You can control the number of BARs that are instantiated in the function by setting the parameter `NUMBER_OF_BARS`. Depending on the value set by this parameter, one or more of the BARs in the `pci_c` function is instantiated. The logic for the unused BARs is reduced automatically by the MAX+PLUS II software when you compile the `pci_c` function.

Each BAR has its own parameter BAR*n* (where *n* is the BAR number). Each BAR should be a 32-bit hexadecimal number, which selects a combination of the following BAR options:

- Type of address space reserved by the BAR
- Location of the reserved memory
- Sets the reserved memory as prefetchable or non-prefetchable
- Size of memory or I/O address space reserved for the BAR

When compiling the `pci_c` function, the MAX+PLUS II software generates informational messages informing you of the number and options of the BARs you have specified.

The BAR is formatted per the *PCI Local Bus Specification, Revision 2.2*. Bit 0 of each BAR is read only, and is used to indicate whether the reserved address space is memory or I/O. BARs that map to memory space must hardwire bit 0 to 0, and BARs that map to I/O space must hardwire bit 0 to 1. Depending on the value of bit 0, the format of the BAR changes. You can set the type of BAR you want to instantiate by setting the individual bit 0 of the corresponding BAR*n* parameter.

In a memory BAR, bits 2 and 1 indicate the location of the address space in the memory map. You can control the location of each BAR address space independently by setting the value of bit 2 and 1 in the corresponding BAR*n* parameter.

Bit 3 of a memory BAR controls whether the BAR is prefetchable. You can control whether the BAR is prefetchable independently by setting the value for bit 3 in the corresponding BAR*n* parameter. See Table 13.

Table 13. Memory BAR Format

Data Bit	Mnemonic	Read/Write	Definition
0	mem_ind	Read	Memory indicator. The mem_ind bit indicates that the register maps into memory address space. This bit must be set to 0 in the BAR*n* parameter.
2..1	mem_type	Read	Memory type. The mem_type bits indicate the type of memory that can be implemented in the pci_c memory address space. Only the following three possible values are valid for pci_c: locate memory space in the 32-bit address space, locate memory space below 1 Mbyte, and locate memory space in the 64-bit address space.
3	pre_fetch	Read	Memory prefetchable. The pre_fetch bit indicates whether the blocks of memory are prefetchable by the host bridge.
31..4	bar	Read/write	Base address registers.

In addition to the type of space reserved by the BAR, the parameter value BAR*n* determines the number of read/write bits instantiated in the corresponding BAR. The number of read/write bits in a BAR determines the size of address space reserved (See Section 6.2.5 in the ***PCI Local Bus Specification, Revision*** 2.2). You can indicate the number of read/write bits instantiated in a BAR by the number of 1s in the corresponding BAR*n* value starting from bit 31. The BAR*n* parameter should contain 1s from bit 31 down to the required bit without any 0s in between. For example, a value of "FF000000" hex is a legal value for a BAR*n* parameter, but the value "FF700000" hex is not, because bits 24 and 22 are 1s and bit 23 is 0. As another example, if you set the BAR0 parameter to "FFC00008", BAR0 would have the following options:

- Memory BAR
- Located anywhere in the 32-bit address space
- Prefetchable
- Reserved memory space = $2^{(32 - 10)}$ = 4 Mbytes

Additionally, for high-end systems that require more than 2 Gbytes of memory space, the `pci_c` function supports 64-bit addressing. BAR0 and BAR1 are used for a 64-bit BAR. BAR0 contains the lower 32-bit BAR, and BAR1 contains the upper 32-bit BAR. For BAR0, bit 0 must be set to 0 to indicate a memory space. Bits 2 and 1 must be set to `B"10"` respectively, to indicate a memory space located anywhere in the 64-bit address space. Also, bit 3 of a memory BAR controls whether the BAR is prefetchable. Bits [31..4] of BAR0 are read/write registers that are used to indicate the size of the memory, along with BAR1. For BAR1, the upper 24-bits [31..8] are read-only bits and are tied to ground. However, in the parameters field of the `pci_c` symbol, the upper 24 bits [31..8] of BAR1 in a 64-bit BAR must still be set to `"FFFFFF"` hex. The 8 least significant bits [7..0] of BAR1 are read/write registers, and along with bits [31..4] of BAR0, they indicate the size of the memory. For example, if you set the `BAR1` parameter to `"FFFFFFFF"` hex and the `BAR0` parameter to `"0000000C"` hex, BAR1 and BAR0 would have the following options:

If BAR1 is used as a 32-bit BAR, the upper 24 bits [31..8] are read/write registers, along with bits [7..4]. The four least significant bits [3..0] are read-only bits and are defined in Table 13 on page 64.

- Memory BAR
- Located anywhere in the 64-bit address space
- Prefetchable
- Reserved memory space = $2^{(64-32)}$ = 4 Gbytes.

Reserved memory space can also be calculated by the following formula: $2^{(40-8)}$ = 4 Gbytes, where 40 = actual available registers and 8 = user assigned read/write register.

If BAR0 and BAR1 are used for a 64-bit memory base address register, the `NUMBER_OF_BARS` parameter should be set to 2. The `BAR5` through `BAR2` parameters can still be used for 32-bit memory or I/O base address registers in conjunction with a 64-bit BAR setting. If BAR5 through BAR2 are used with a 64-bit BAR setting, the `NUMBER_OF_BARS` parameter should be set to 6.

Like a memory BAR, the corresponding BAR*n* parameter can be used to instantiate an I/O BAR in any of the six BARs available for the `pci_c` function. You can instantiate an I/O BAR by setting bit 0 of the corresponding BAR*n* parameter to 1 instead of 0.

In an I/O BAR, bit 1 is always reserved and you should set it to 1. Like the memory BAR, the read/write bits in the most significant part of the BAR control the amount of address space reserved. You can indicate the number of read/write bits you would like to instantiate in a BAR by setting the appropriate bits to a 1 in the corresponding BAR*n* parameter. The *PCI Local Bus Specification, Revision* 2.2 prevents any single I/O BAR from reserving more than 256 bytes of I/O space. See Table 14.

For example, if you set the BAR1 parameter to "FFFFFFC1", BAR 1 would have the following options:

- I/O BAR
- Reserved I/O space = $2^{(32 - 26)}$ = 64 bytes

Table 14. I/O Base Address Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	io_ind	Read	I/O indicator. The io_ind bit indicates that the register maps into I/O address space. This bit must be set to 1 in the BAR*n* parameter.
1	Reserved	–	–
31..2	bar	Read/write	Base address registers.

Subsystem Vendor ID Register

Subsystem vendor ID is a 16-bit read-only register that identifies add-in cards from different vendors that have the same functionality. The value of this register is assigned by the PCI SIG. See Table 15. The default value of the subsystem vendor ID register is 0000 hex. However, designers can change the value by setting the SUBSYSTEM_VEND_ID parameter.

Table 15. Subsystem Vendor ID Register Format

Data Bit	Mnemonic	Read/Write	Definition
15..0	sub_vend_id	Read	PCI subsystem/vendor ID

Subsystem ID Register

The subsystem ID register identifies the subsystem. The value of this register is defined by the subsystem vendor, i.e., the designer. See Table 16. The default value of the subsystem ID register is 0000 hex. However, designers can change the value by setting the SUBSYSTEM_ID parameter.

Table 16. Subsystem ID Register Format

Data Bit	Mnemonic	Read/Write	Definition
15..0	sub_id	Read	PCI subsystem ID

Expansion ROM Base Address Register

The expansion ROM base address register contains a 32-bit hexadecimal number that defines the base address and size information of the expansion ROM. You can instantiate the expansion ROM BAR by setting the parameter EXP_ROM_ENA="YES". The expansion ROM BAR functions exactly like a 32-bit BAR, except that the encoding of the bottom bits is different. Bit 0 in the register is a read/write and is used to indicate whether or not the device accepts accesses to its expansion ROM. You can disable the expansion ROM address space by setting bit 0 to 0. You can enable the address decoding of the expansion ROM by setting bit 0 to 1. The upper 21 bits correspond to the upper 21 bits of the expansion ROM base address. The amount of address space a device requests must not be greater than 16 Mbytes. The expansion ROM BAR is formatted per the *PCI Local Bus Specification, Revision* 2.2. See Table 17.

Table 17. Expansion ROM Base Address Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	adr_ena	Read/write	Address decode enable. The adr_ena bit indicates whether or not the device accepts accesses to its expansion ROM. You can disable the expansion ROM address space by setting this bit to 0. You can enable the address decoding of the expansion ROM by setting this bit to 1.
10..1	Reserved	–	–
31..11	bar	Read/write	Expansion ROM base address registers.

Capabilities Pointer

The capabilities pointer register is an 8-bit read-only register. The value set to the parameter CAP_PTR points to the first item in the list of capabilities. For a list of the capability IDs, see appendix H in the *PCI Local Bus Specification, Revision* 2.2. The address location of the pointer must be 40 hex or greater, and each capability must be within DWORD boundaries. To enable the capabilities pointer register, the parameter CAP_LIST_ENA must be set to "YES". See Table 18.

Table 18. Interrupt Line Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	cap_ptr	Read/write	Capabilities pointer register

Interrupt Line Register

The interrupt line register is an 8-bit register that defines to which system interrupt request line (on the system interrupt controller) the intan output is routed. The interrupt line register is written by the system software upon power-up; the default value is FF hex. See Table 19.

Table 19. Interrupt Line Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	int_ln	Read/write	Interrupt line register

Interrupt Pin Register

The interrupt pin register is an 8-bit read-only register that defines the pci_c function PCI bus interrupt request line to be intan. The default value of the interrupt pin register is 01 hex. See Table 20.

Table 20. Interrupt Pin Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	int_pin	Read	Interrupt pin register

Minimum Grant Register

The minimum grant register is an 8-bit read-only register that defines the length of time the pci_c function would like to retain mastership of the PCI bus. The value set in this register indicates the required burst period length in 250-ns increments. Designers can set this register with the parameter MIN_GRANT. See Table 21.

Table 21. Minimum Grant Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	min_gnt	Read	Minimum grant register

Maximum Latency Register

The maximum latency register is an 8-bit read-only register that defines the frequency in which the pci_c function would like to gain access to the PCI bus. See Table 22. Designers can set this register with the parameter MAX_LAT.

Table 22. Maximum Latency Register Format

Data Bit	Mnemonic	Read/Write	Definition
7..0	max_lat	Read	Maximum latency register

Target Mode Operation

This section describes all supported target transactions for the pci_c function and includes waveform diagrams showing typical PCI cycles in target mode. The MegaCore function supports both 32-bit and 64-bit transactions. The pci_c function supports the following 64-bit memory transactions:

- 64-bit memory single-cycle target read
- 64-bit memory burst target read
- 64-bit memory single-cycle target write
- 64-bit memory burst target write

The pci_c function also supports the following 32-bit transactions:

- 32-bit memory single-cycle target read
- 32-bit memory burst target read
- I/O target read
- Configuration target read
- 32-bit memory single-cycle target write
- 32-bit memory burst target write
- I/O target write
- Configuration target write

The pci_c function assumes that the local side is 64 bits during memory transactions and 32 bits during I/O transactions. Therefore, the pci_c function automatically reads 64-bit data on the local side and transfers the data to the PCI master, one DWORD at a time, if the PCI bus is 32 bits wide.

A read or write transaction begins after a master device acquires mastership of the PCI bus and asserts framen to indicate the beginning of a bus transaction. If the transaction is a 64-bit transaction, the master device asserts the req64n signal at the same time it asserts the framen signal. The clock cycle, where the framen signal is asserted, is called the address phase. During the address phase, the master device drives the transaction address and command on ad[31..0] and cben[3..0], respectively. When framen is asserted, the MegaCore function latches the address and command signals on the first clock edge and starts the address decode phase. If the transaction address and command match the pci_c target, the pci_c target asserts the devseln signal to claim the transaction. In the case of 64-bit transactions, pci_c also asserts the ack64n signal at the same time as the devseln signal indicating that it accepts the 64-bit transaction. The MegaCore functions implement slow decode, i.e., the devseln and ack64n signals are asserted three clock cycles after a valid address is presented on the PCI bus. In all operations except configuration read/write, one of the lt_tsr[5..0] signals is driven high, indicating the BAR range address of the current transaction.

Configuration transactions are always single-cycle 32-bit transactions. The MegaCore function has complete control over configuration transactions and informs the local-side device of the progress and command of the transaction. The MegaCore function asserts all control signals, provides data in the case of a read, and receives data in the case of a write without interaction from the local-side device.

Memory transactions can be single-cycle or burst. In target mode, the MegaCore function supports an unlimited length of zero-wait-state memory burst read or write. In a read transaction, data is transferred from the local side to the PCI master. In a write transaction, data is transferred from the PCI master to the local-side device. A memory transaction can be terminated by either the PCI master or the local-side device. The local-side device can terminate the memory transaction using one of three types of terminations: retry, disconnect, or target abort. "Target Transaction Terminations" on page 99 describes how to initiate the different types of termination.

The MegaCore function treats the memory read line and memory read multiple commands as memory read. Similarly, the function treats the memory write and invalidate command as a memory write. The local-side application must implement any special requirements required by these commands.

I/O transactions are always single-cycle 32-bit transactions. Therefore, the MegaCore function handles them like single-cycle memory commands. Any of the six BARs in the `pci_c` function can be configured to reserve I/O space. See "Base Address Registers" on page 63 for more information on how to configure a specific BAR to be an I/O BAR. Like memory transactions, I/O transactions can be terminated normally by the PCI master, or the local-side device can instruct the MegaCore function to terminate the transactions with a retry or target abort. Because all I/O transactions are single-cycle, terminating a transaction with a disconnect does not apply.

64-Bit Target Read Transactions

In target mode, the MegaCore functions support two types of read transactions:

- Memory single-cycle read
- Memory burst read

For both types of read transactions, the sequence of events is the same and can be divided into the following steps:

1. The address phase occurs when the PCI master asserts `framen` and `req64n` signals and drives the address and command on `ad[31..0]` and `cben[3..0]`, correspondingly. Asserting the `req64n` signal indicates to the target device that the master device is requesting a 64-bit data transaction.

2. Turn-around cycles on the `ad[63..0]` bus occur during the clock immediately following the address phase. During the turn-around cycles, the PCI master tri-states the `ad[63..0]` bus, but drives correct byte-enables on `cben[7..0]` for the first data phase. This process is necessary because the PCI agent driving the `ad[63..0]` bus changes during read cycles.

3. If the address of the transactions match one of the base address registers, the `pci_c` function turns on the drivers for the `ad[63..0]`, `devseln`, `ack64n`, `trdyn`, and `stopn` signals. The drivers for `par` and `par64` are turned on in the following clock.

4. The `pci_c` function drives and asserts `devseln` and `ack64n` to indicate to the master device that it is accepting the 64-bit transaction.

5. One or more data phases follow next, depending on the type of read transaction.

64-Bit Single-Cycle Target Read Transaction

Figure 1 shows the waveform for a 64-bit single-cycle target read transaction.

Figure 1. 64-Bit Single-Cycle Target Read Transaction

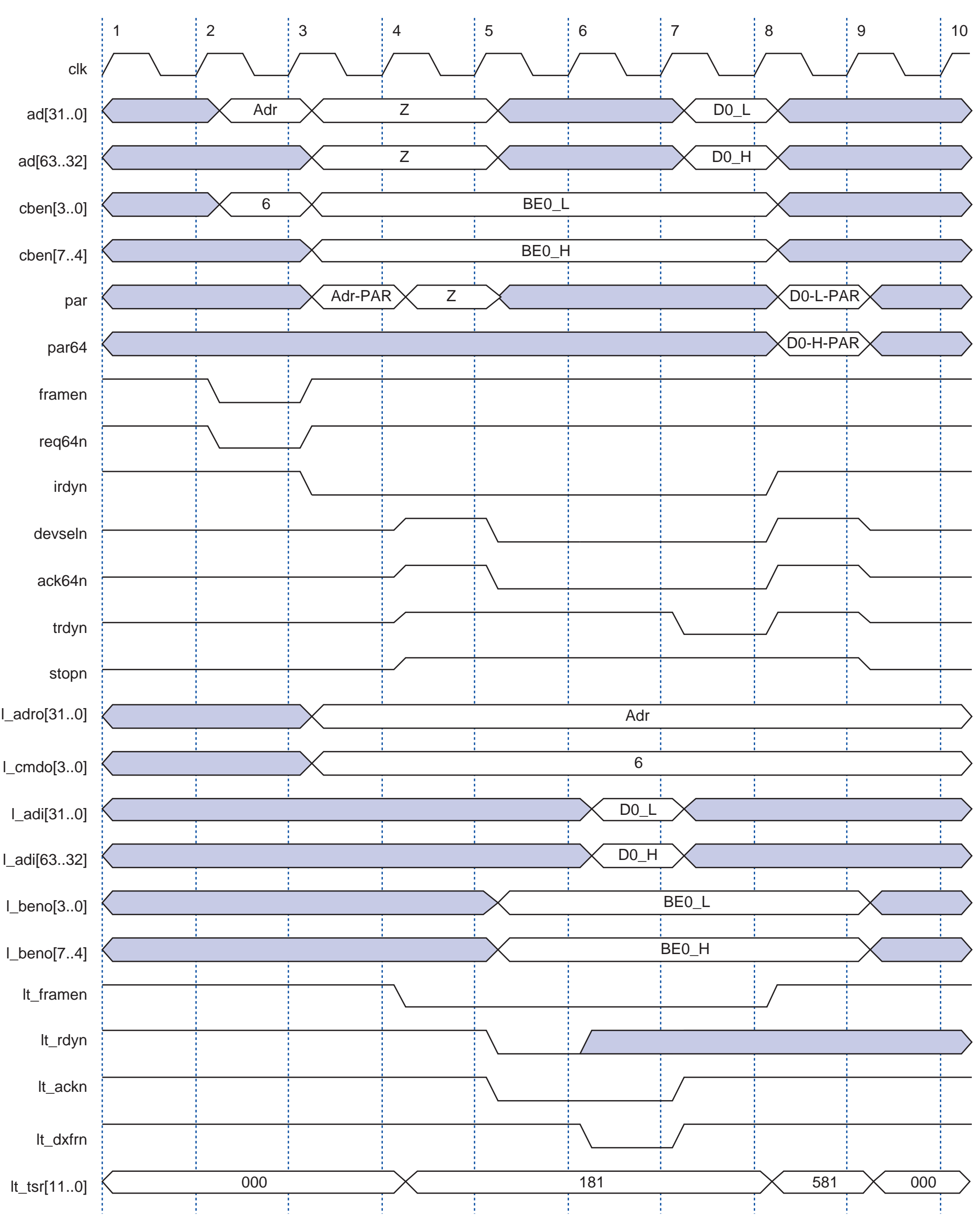


Table 23 shows the sequence of events for a single-cycle target read transaction.

Table 23. Single-Cycle Target Read Transaction (Part 1 of 2)

Clock Cycle	Event
1	The PCI bus is idle.
2	The address phase occurs.
3	The MegaCore function latches the address and command, and decodes the address to check if it falls within the range of one of its BARs. During clock 3, the master deasserts the `framen` and `req64n` signals and asserts `irdyn` to indicate that only one data phase remains in the transaction. For a single-cycle target read, this phase is the only data phase in the transaction. The MegaCore function begins to decode the address during clock 3, and if the address falls in the range of one of its BARs, the transaction is claimed. The PCI master tri-states the `ad[63..0]` bus for the turn-around cycle.
4	If the MegaCore function detects an address hit in clock 3, several events occur during clock 4: ■ The MegaCore function informs the local-side device that it is going to claim the read transaction by asserting one of the `lt_tsr[5..0]` signals and `lt_framen`. In Figure 1, `lt_tsr[0]` is asserted indicating that a base address register zero hit. ■ The MegaCore function drives the transaction command on `l_cmdo[3..0]` and address on `l_adro[31..0]`. ■ The MegaCore function turns on the drivers of `devseln`, `ack64n`, `trdyn`, and `stopn`, getting ready to assert `devseln` and `ack64n` in clock 5. ■ `lt_tsr[7]` is asserted to indicate that the pending transaction is 64-bits. ■ `lt_tsr[8]` is asserted to indicate that the PCI side of the `pci_c` function is busy. ■ `lt_tsr[9]` is not asserted indicating that the current transaction is single-cycle. ☞ A burst transaction can be identified if both the `irdyn` and `framen` signals are asserted at the same time during a transaction. The `pci_c` function asserts `lt_tsr[9]` if both `irdyn` and `framen` are asserted during a valid target transaction. If `lt_tsr[9]` is not asserted during a transaction, it indicates that `irdyn` and `framen` have not been detected or asserted during the transaction. Typically this situation indicates that the current transaction is single-cycle. However, this situation is not guaranteed because it is possible for the master to delay the assertion of `irdyn` in the first data phase by up to 8 clocks. In other words, if `lt_tsr[9]` is asserted during a valid target transaction, it indicates that the pending transaction is a burst, but if `lt_tsr[9]` is not asserted it may or may not indicate that the transaction is single-cycle.
5	The MegaCore function asserts `devseln` and `ack64n` to claim the transaction. The function also drives `lt_ackn` to the local-side device to indicate that it is ready to accept data on `l_adi[63..0]`. The MegaCore function also enables the output drivers of the `ad[63..0]` bus to ensure that it is not tri-stated for a long time while waiting for valid data. Although the local side asserts `lt_rdyn` during clock 5, the data transfer does not occur until clock 6.

Table 23. Single-Cycle Target Read Transaction (Part 2 of 2)	
6	lt_rdyn is asserted in clock 5, indicating that valid data is available on l_adi[63..0] in clock 6. The MegaCore function registers the data into its internal pipeline on the rising edge of clock 7. The local side transfer is indicated by the lt_dxfrn signal. The lt_dxfrn signal is low during the clock where a data transfer on the local side occurs.
7	The rising edge of clock 7 registers the valid data from l_adi[63..0] and drives the data on the ad[63..0] bus. At the same time, the pci_c function asserts the trdyn signal to indicate that there is valid data on the ad[63..0] bus.
8	The MegaCore function deasserts trdyn, devseln, and ack64n to end the transaction. To satisfy the requirements for sustained tri-state buffers, the MegaCore function drives devseln, ack64n, trdyn, and stopn high during this clock cycle. Additionally, the MegaCore function tri-states the ad[63..0] bus because the cycle is complete. The rising edge of clock 8 signals the end of the last data phase because framen is deasserted and irdyn and trdyn are asserted. In clock 8, the pci_c function also informs the local side that no more data is required by deasserting lt_framen, and lt_tsr[10] is asserted to indicate a successful data transfer on the PCI side during the previous clock cycle.
9	The MegaCore function informs the local-side device that the transaction is complete by deasserting the lt_tsr[11..0] signals. Additionally, the MegaCore function tri-states devseln, ack64n, trdyn, and stopn to begin the turn-around cycle on the PCI bus.

The local-side device must ensure that PCI latency rules are not violated while the MegaCore function waits for data. If the local-side device is unable to meet the latency requirements, it must assert lt_discn to request that the MegaCore function terminate the transaction. The PCI target latency rules state that the time to complete the first data phase must not be greater than 16 PCI clocks, and the subsequent data phases must not take more than 8 PCI clock cycles to complete.

64-Bit Memory Burst Read Transaction

The sequence of events for a burst read transaction is the same as that of a single-cycle read transaction. However, during a burst read transaction, more data is transferred and both the local-side device and the PCI master can insert waits states at any point during the transaction. Figure 2 illustrates a burst read transaction.

Figure 2. 64-Bit Zero Wait State Target Burst Read Transaction

1 2 3 4 5 6 7 8 9 10 11 12 13

clk

ad[31..0] Adr Z D0_L D1_L D2_L D3_L

ad[63..32] Z D0_H D1_H D2_H D3_H

cben[3..0] 6 BE_L

cben[7..4] BE_H

par Adr-PAR Z D0-L-PAR D1-L-PAR D2-L-PAR D3-L-PAR

par64 D0-H-PAR D1-H-PAR D2-H-PAR D3-H-PAR

framen

req64n

irdyn

devseln

ack64n

trdyn

stopn

l_adro[31..0] Adr

l_cmdo[3..0] 6

l_adi[31..0] D0_L D1_L D2_L D3_L D4_L

l_adi[63..32] D0_H D1_H D2_H D3_H D4_H

l_beno[3..0] BE_L

l_beno[7..4] BE_H

lt_framen

lt_rdyn

lt_ackn

lt_dxfrn

lt_tsr[11..0] 000 381 781 000

Figure 2 shows a 64-bit zero wait state burst transaction with four data phases. The local side transfers five quad words (QWORDs) in clocks 6 through 10. The PCI side transfers data in clocks 7 through 10. Because of the zero wait state requirement of the `pci_c` function, it reads ahead from the local side. If the local side is not prefetchable (i.e., reading ahead will result in lost or corrupt data), it must not accept burst read transactions, and it should disconnect after the first QWORD transfer on the local side. Additionally, Figure 2 shows the `lt_tsr[9]` signal asserted in clock 4 because the master device has `framen` and `irdyn` signals asserted, thus indicating a burst transaction.

Figure 3 shows the same transaction as in Figure 2 with the PCI bus master asserting a wait state. The PCI bus master asserts a wait state by deasserting `irdyn` in clock 8. The effect of this wait state on the local side is shown in clock 9 because `lt_ackn` is deasserted, and as a result `lt_dxfrn` is also deasserted. This situation prevents further data from being transferred on the local side because the internal pipeline of the `pci_c` function is full.

Figure 3. 64-Bit Target Burst Read Transaction with PCI Master Wait State

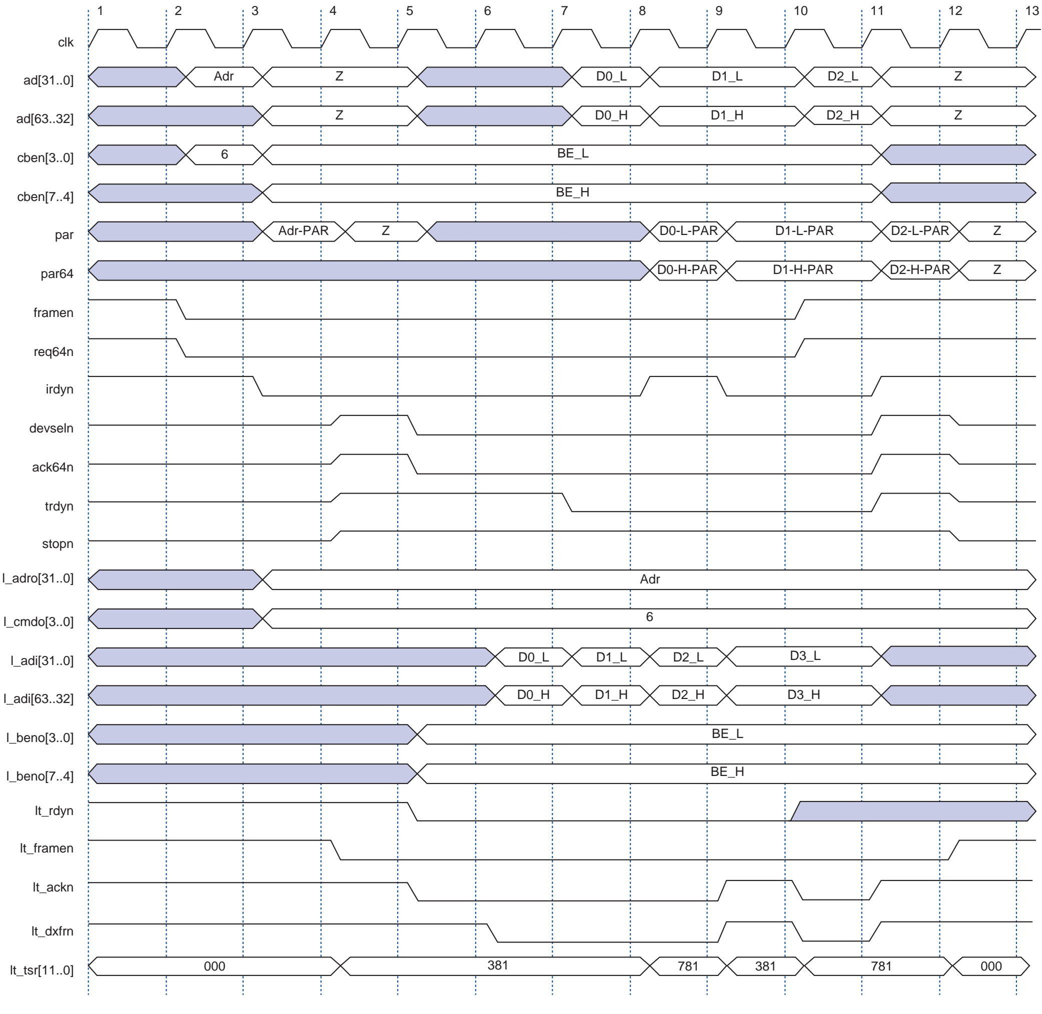
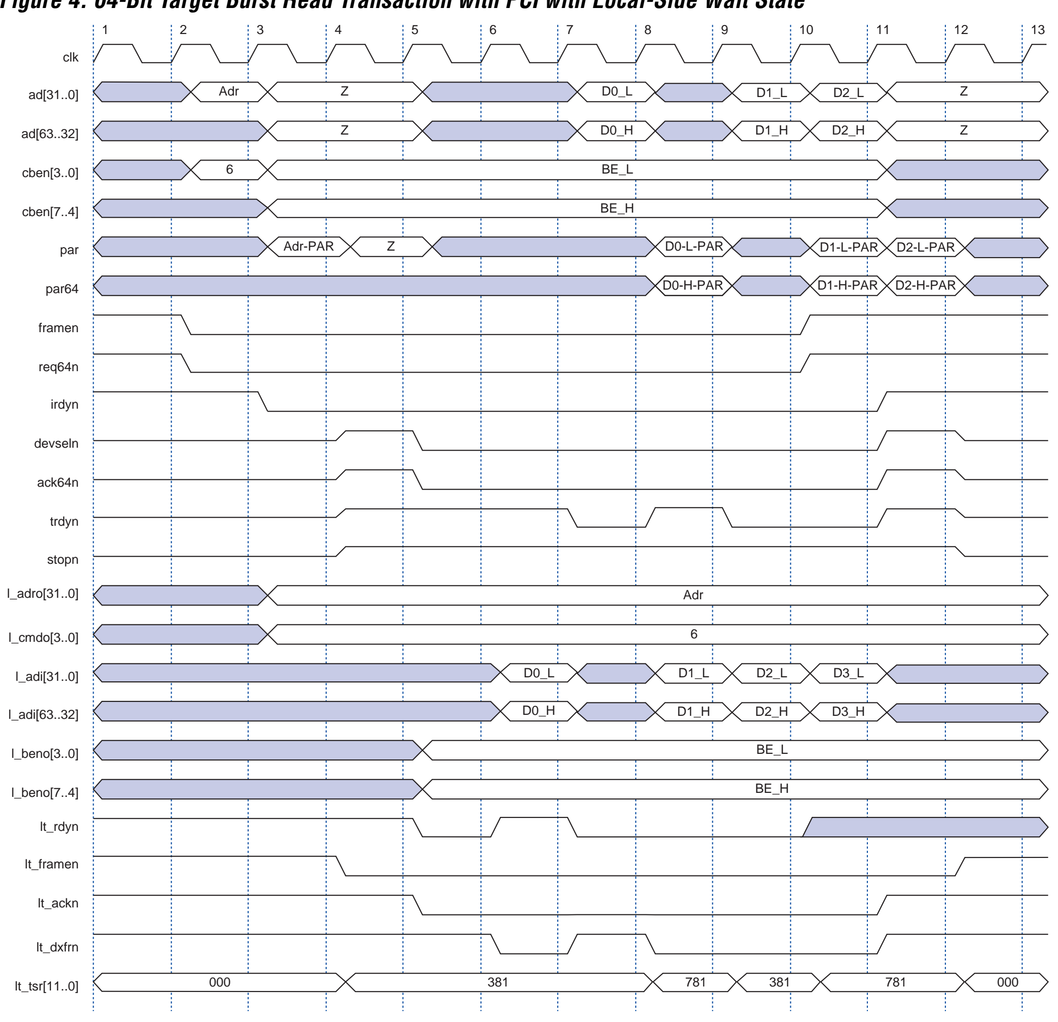


Figure 4 shows the same transaction as shown in Figure 2 with the local side asserting a wait state. The local side deasserts `lt_rdyn` in clock 6. Deasserting `lt_rdyn` in clock 6 suspends the local side data transfer in clock 7 by deasserting the `lt_dxfrn` signal. Because no data is transferred in clock 7 from the local side, the `pci_c` function deasserts `trdyn` in clock 8 thus inserting a PCI wait state.

Figure 4. 64-Bit Target Burst Read Transaction with PCI with Local-Side Wait State



The local-side device must ensure that PCI latency rules are not violated while the MegaCore function waits for data. If the local-side device is unable to meet the latency requirements, it must assert `lt_discn` to request that the MegaCore function terminate the transaction. The PCI target latency rules state that the time to complete the first data phase must not be greater than 16 PCI clocks, and the subsequent data phases must not take more than 8 PCI clocks to complete.

32-Bit Target Read Transactions

The `pci_c` function responds to three types of 32-bit target read transactions:

- Memory read transactions
- I/O read transactions
- Configuration read transactions

32-Bit Memory Read Transactions

Memory transactions are either single-cycle or burst. For memory transactions, the `pci_c` function always assumes a 64-bit local side. The `pci_c` function automatically reads 64-bit data on the local side and transfers the data to the PCI master, one DWORD at a time, if the PCI bus is 32 bits wide. In a memory read cycle, `pci_c` asserts both `l_ldat_ackn` and `l_hdat_ackn` to indicate that data is transferred 64 bits at a time on the local side. The `pci_c` function decodes whether the low or high DWORD is addressed by the master, based on the starting address of the transaction:

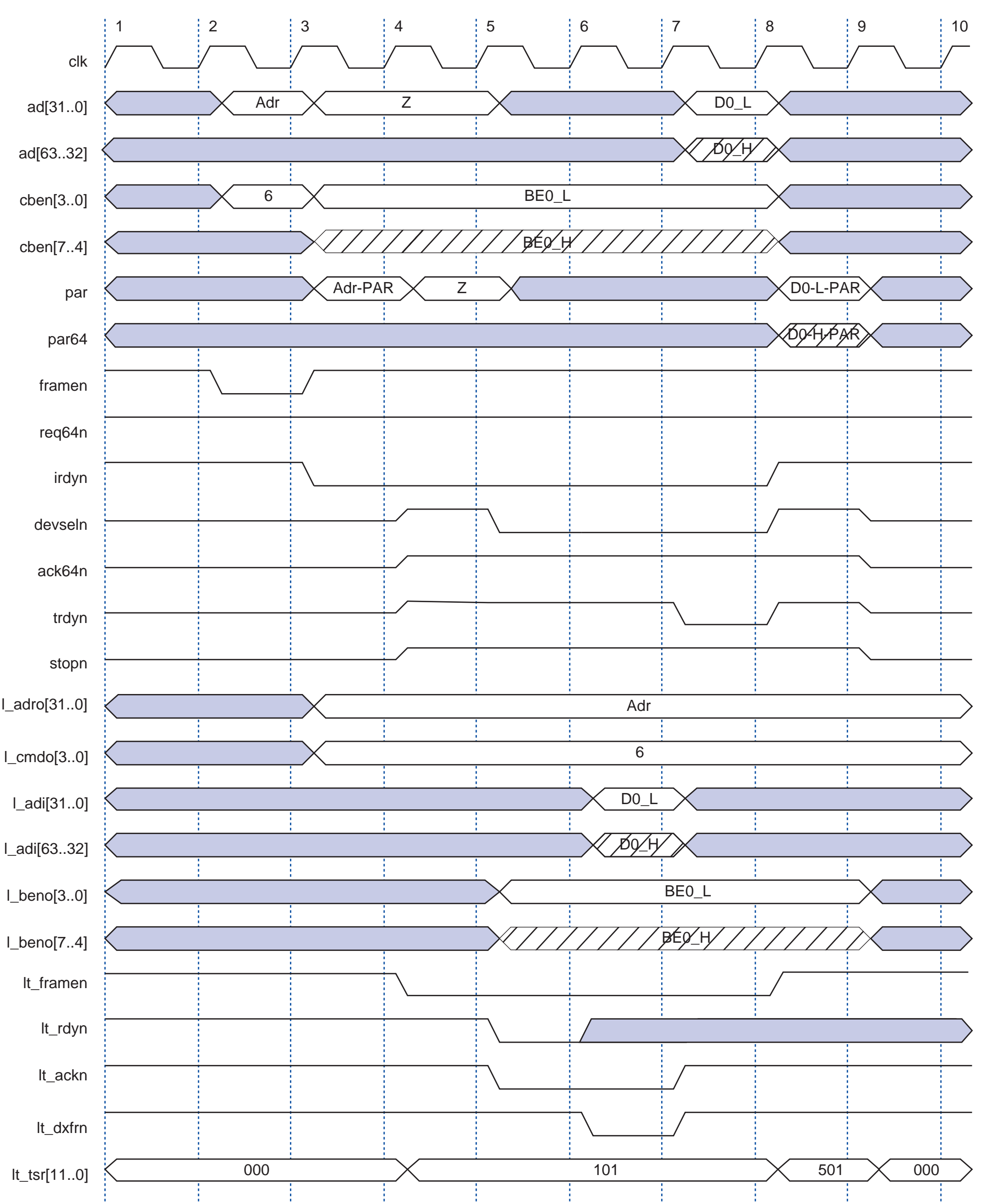
- If the address of the transaction is a QWORD boundary (`ad[2..0] == B"000"`), the first DWORD transferred to the PCI side is the low DWORD, and `pci_c` asserts both `l_ldat_ackn` and `l_hdat_ackn`.
- However, if the address of the transaction is not at QWORD boundary (`ad[2..0] == B"100"`), the first DWORD transferred to the PCI side is the high DWORD of the first 64-bit data phase. The low DWORD of the first 64-bit data phase is not transferred to the PCI side. For the following 64-bit data phases after the first, the low DWORD is transferred first to the PCI side, followed by the high DWORD.

Figure 5 shows a 32-bit single-cycle memory read transaction. The sequence of events in Figure 5 is exactly the same as in Figure 1, except for the following cases:

- During the address phase (clock 3), the master does not assert `req64n` because the transaction is 32 bits.
- The `pci_c` function does not assert `ack64n` when it asserts `devseln`.
- The local side is informed that the pending transaction is 32 bits because `lt_tsr[7]` is not asserted while `lt_framen` is asserted.

Figure 5 shows that the local side transfers a full QWORD in clock 6. However, the `pci_c` function transfers only the least significant DWORD to the PCI master but still drives the full 64-bit word in clock 7. The `pci_c` function also drives the correct parity value on the `par64` signal in clock 8.

Figure 5. 32-Bit Single-Cycle Memory Read Transaction



The pci_c function always transfers 64-bit data on the local side. In a 32-bit single-cycle memory read transaction, only the least significant DWORD is transferred to the PCI master. Therefore, the local side is only required to transfer the least significant DWORD in a 32-bit single-cycle transaction. See Figure 5.

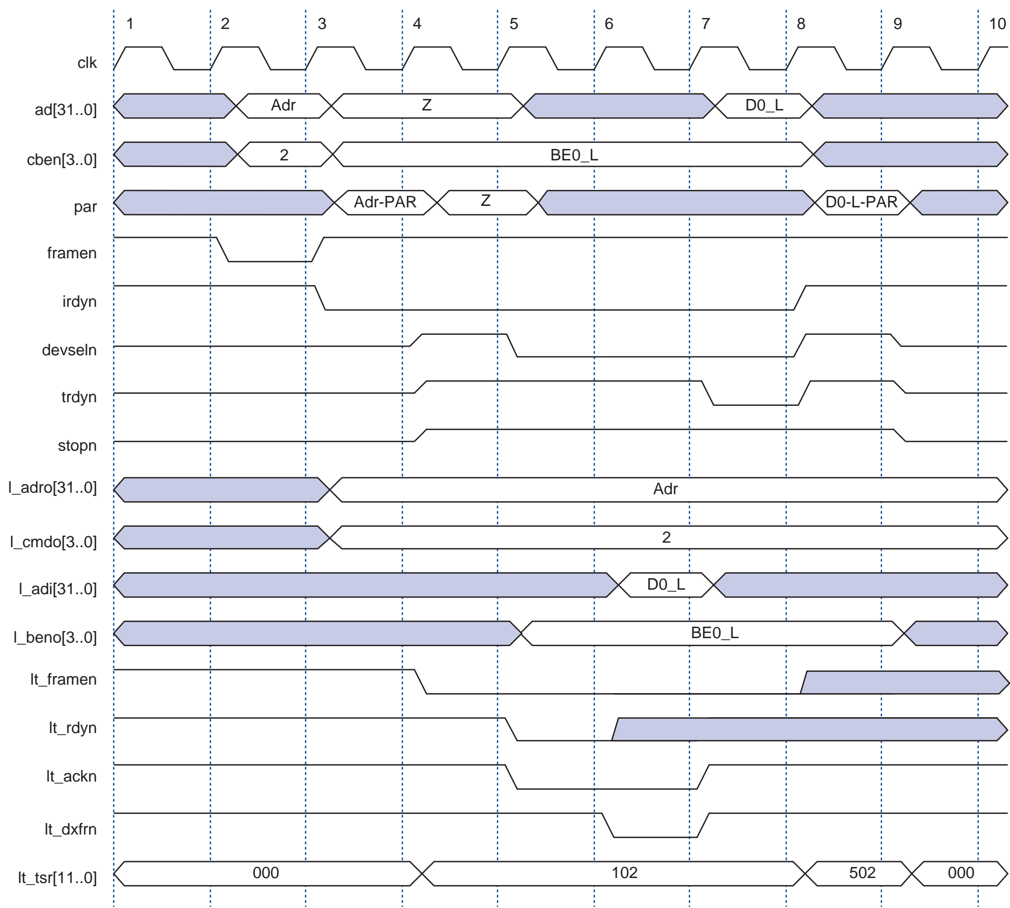
Figure 6 shows a 32-bit burst memory read transaction. The events in Figure 6 are the same as in Figure 2. The main difference between the two is that a 64-bit transfer takes one clock on the local side, but requires two clocks on the PCI side. Therefore, the pci_c function automatically asserts local wait states in clocks 7 and 9 to temporarily suspend the local transfer allowing sufficient time for the data to be transferred on the PCI side. In Figure 6, lt_tsr[7] is not asserted and lt_tsr[9] is asserted indicating that the transaction is a 32-bit burst. If the local side cannot handle 32-bit burst transactions, it can disconnect after the first local transfer.

Figure 6. 32-Bit Burst Memory Read Transaction

I/O Read Transaction

I/O read transactions by definition are 32 bits. Figure 7 shows a sample I/O read transaction. The sequence of events is the same as 32-bit single-cycle memory read transactions. The main distinction between the two transactions is the command on the `lt_cmdo[3..0]` bus. In Figure 7, `lt_tsr[11..0]` indicates that the base address register that detected the address hit is BAR1. Additionally, during an I/O transaction `l_ldat_ackn` and `l_hdat_ackn` are not relevant.

Figure 7. I/O Read Transaction



Configuration Read Transaction

Configuration read transactions are 32 bits. Configuration cycles are automatically handled by the `pci_c` function and do not require local side actions. Figure 8 shows a typical configuration read transaction. The configuration read transaction is similar to 32-bit single-cycle transactions, except for the following terms:

- During the address phase, `idsel` must be asserted
- Because the configuration read does not require data from the local side, `pci_c` asserts `trdyn` independent from the `lt_rdyn` signal. This situation results in `trdyn` being asserted in clock 6 instead of clock 7 as shown in Figure 4. The configuration read cycle ends in clock 8.

Figure 8. Configuration Read Transaction

The local side cannot retry, disconnect, or abort configuration cycles.

64-Bit Target Write Transactions

In target mode, the MegaCore function supports two types of 64-bit memory write transactions.

- Memory single-cycle write
- Memory burst write

For both types of write transactions, the events follow the sequence described below:

1. The address phase occurs when the PCI master asserts the `framen` and `req64n` signals and drives the address and command on `ad[31..0]` and `cben[3..0]` correspondingly. Asserting `req64n` indicates to the target device that the master device is requesting a 64-bit data transaction.

2. If the address of the transaction matches one of the BARs, the `pci_c` function turns on the drivers for `ad[63..0]`, `devseln`, `ack64n`, `trdyn`, and `stopn`. The drivers for `par` and `par64` are turned on during the following clock.

3. The `pci_c` function asserts `devseln` and `ack64n` to indicate to the master device that it is accepting the 64-bit transaction.

4. One or more data phases follow next, depending on the type of write transaction.

64-Bit Single-Cycle Target Write Transaction

Figure 9 shows the waveform for a 64-bit single-cycle target write transaction.

Figure 9. 64-Bit Single-Cycle Target Write Transaction

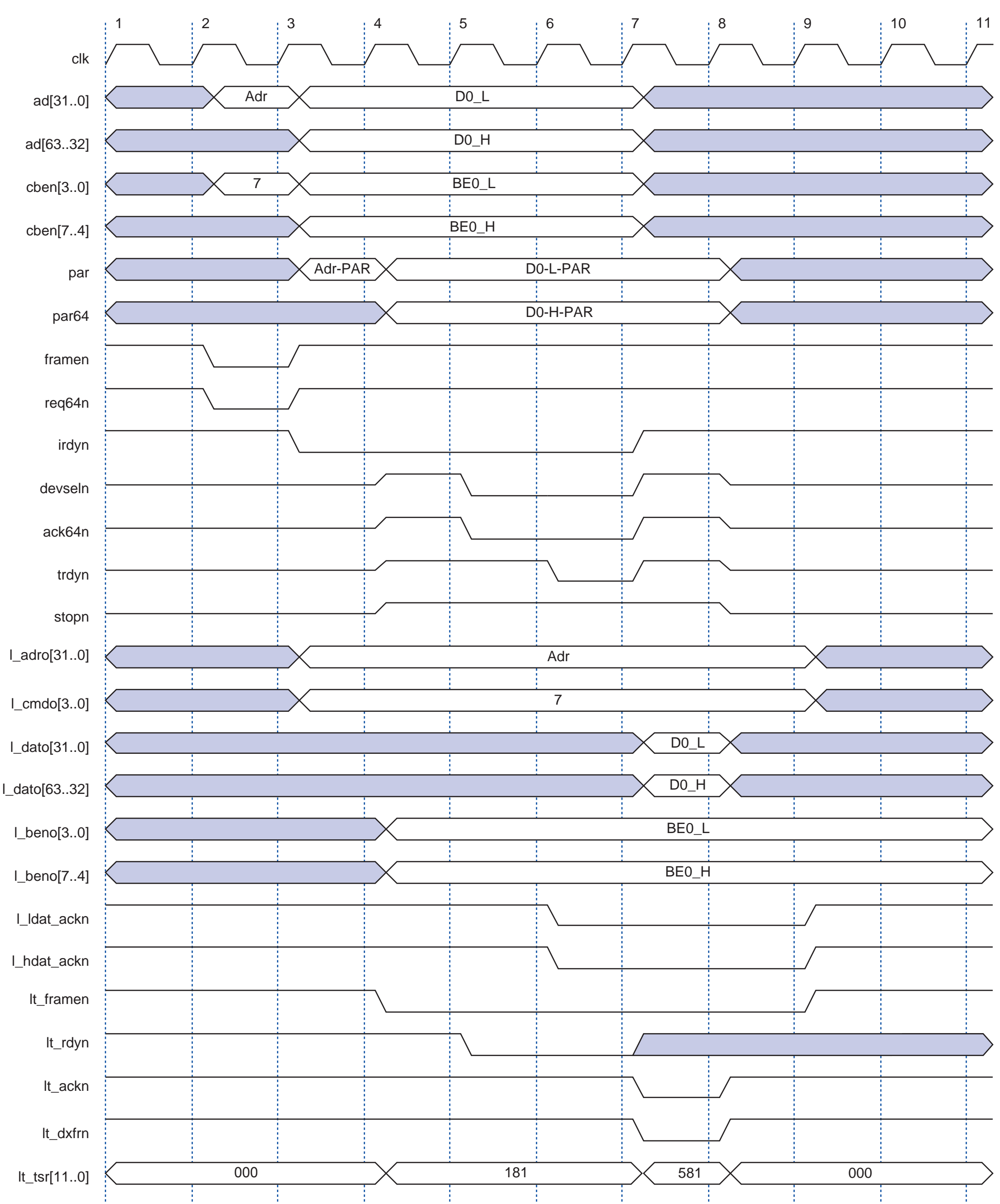


Table 24 shows the sequence of events for a single-cycle target write transaction.

Table 24. 64-Bit Single-Cycle Target Write Transactions (Part 1 of 2)

Clock Cycle	Event
1	The PCI bus is idle.
2	The address phase occurs.
3	The MegaCore function latches the address and command, and decodes the address to check if it falls within the range of one of its BARs. During clock 3, the master deasserts the `framen` and `req64n` signals and asserts `irdyn` to indicate that only one data phase remains in the transaction. For a single-cycle target write, this phase is the only data phase in the transaction. The MegaCore function uses clock 3 to decode the address, and if the address falls in the range of one of its BARs, the transaction is claimed.
4	If the MegaCore function detects an address hit in clock 3, several events occur during clock 4: ■ The MegaCore function informs the local-side device that it is going to claim the write transaction by asserting one of the `lt_tsr[5..0]` signals and `lt_framen`. In Figure 9, `lt_tsr[0]` is asserted indicating that a base address register zero hit. ■ The MegaCore function drives the transaction command on `l_cmdo[3..0]` and address on `l_adro[31..0]`. ■ The MegaCore function turns on the drivers of `devseln`, `ack64n`, `trdyn`, and `stopn` getting ready to assert `devseln` and `ack64n` in clock 5. ■ `lt_tsr[7]` is asserted to indicate that the pending transaction is 64 bits. ■ `lt_tsr[8]` is asserted to indicate that the PCI side of the `pci_c` function is busy. ■ `lt_tsr[9]` is not asserted indicating that the current transaction is a single-cycle. A burst transaction can be identified if both the `irdyn` and `framen` signals are asserted at the same time during a transaction. The `pci_c` function asserts `lt_tsr[9]` if both `irdyn` and `framen` are asserted during a valid target transaction. If `lt_tsr[9]` is not asserted during a transaction, it indicates that `irdyn` and `framen` have not been detected or asserted during the transaction. Typically this event indicates that the current transaction is single-cycle. However, this indication is not guaranteed because it is possible for the master to delay the assertion of `irdyn` in the first data phase by up to 8 clocks. In other words, if `lt_tsr[9]` is asserted during a valid target transaction, it indicates that the pending transaction is a burst, but if the `lt_tsr[9]` is not asserted it may or may not indicate that the transaction is single-cycle.
5	The MegaCore function asserts `devseln` to claim the transaction. Figure 9 also shows the local side asserting `lt_rdyn`, indicating that it is ready to receive data from the MegaCore function in clock 6. To allow the local side ample time to issue a retry for the write cycle, the `pci_c` function does not assert `trdyn` in the first data phase unless the local side asserts `lt_rdyn`. If the `lt_rdyn` signal is not asserted in clock 5 (Figure 9), the `pci_c` function delays the assertion of `trdyn` accordingly.

Table 24. 64-Bit Single-Cycle Target Write Transactions (Part 2 of 2)

Clock Cycle	Event
6	The MegaCore function asserts `trdyn` to inform the PCI master that it is ready to accept data. Because `irdyn` is already asserted, this clock is the first and last data phase in this cycle.
7	The rising edge of clock 7 registers the valid data from `ad[63..0]` and drives the data on the `l_dato[63..0]` bus, registers valid byte enables from `cben[7..0]`, and drives the byte enables on `l_beno[7..0]`. At the same time, `pci_c` asserts the `lt_ackn` signal to indicate that there is valid data on the `l_dato[63..0]` bus and a valid byte enable on the `l_beno[7..0]` bus. Because `lt_rdyn` is asserted during clock 6, and `lt_ackn` is asserted in clock 7, data will be transferred in clock 7. `lt_dxfrn` is asserted in clock 7 to signify a local-side transfer. `lt_tsr[10]` is asserted to indicate a successful data transfer on the PCI side during the previous clock cycle. The MegaCore function also deasserts `trdyn`, `devseln`, and `ack64n` to end the transaction. To satisfy the requirements for sustained tri-state buffers, the MegaCore function drives `devseln`, `ack64n`, `trdyn`, and `stopn` high during this clock cycle.
8	The `pci_c` function resets all `lt_tsr[11..0]` signals because the PCI side has completed the transaction. The `pci_c` function also tri-states its control signals.
9	The `pci_c` function deasserts `lt_framen` indicating to the local side that no additional data is in the internal pipeline.

64-Bit Target Burst Write Transaction

The sequence of events in a burst write transaction is the same as for a single-cycle write transaction. However, in a burst write transaction, more data is transferred and both the local-side device and the PCI master can insert wait-states.

Figure 10 shows a 64-bit zero wait state burst transaction with five data phases. The PCI master writes five QWORDs to `pci_c` during clocks 6 through 10. The local side transfers the same data during clocks 7 through 11 correspondingly. Additionally, Figure 10 shows the `lt_tsr[9]` signal asserted in clock 4 because the master device has the `framen` and `irdyn` signals asserted, thus indicating a burst transaction.

Figure 10. 64-Bit Zero Wait State Target Burst Write Transaction

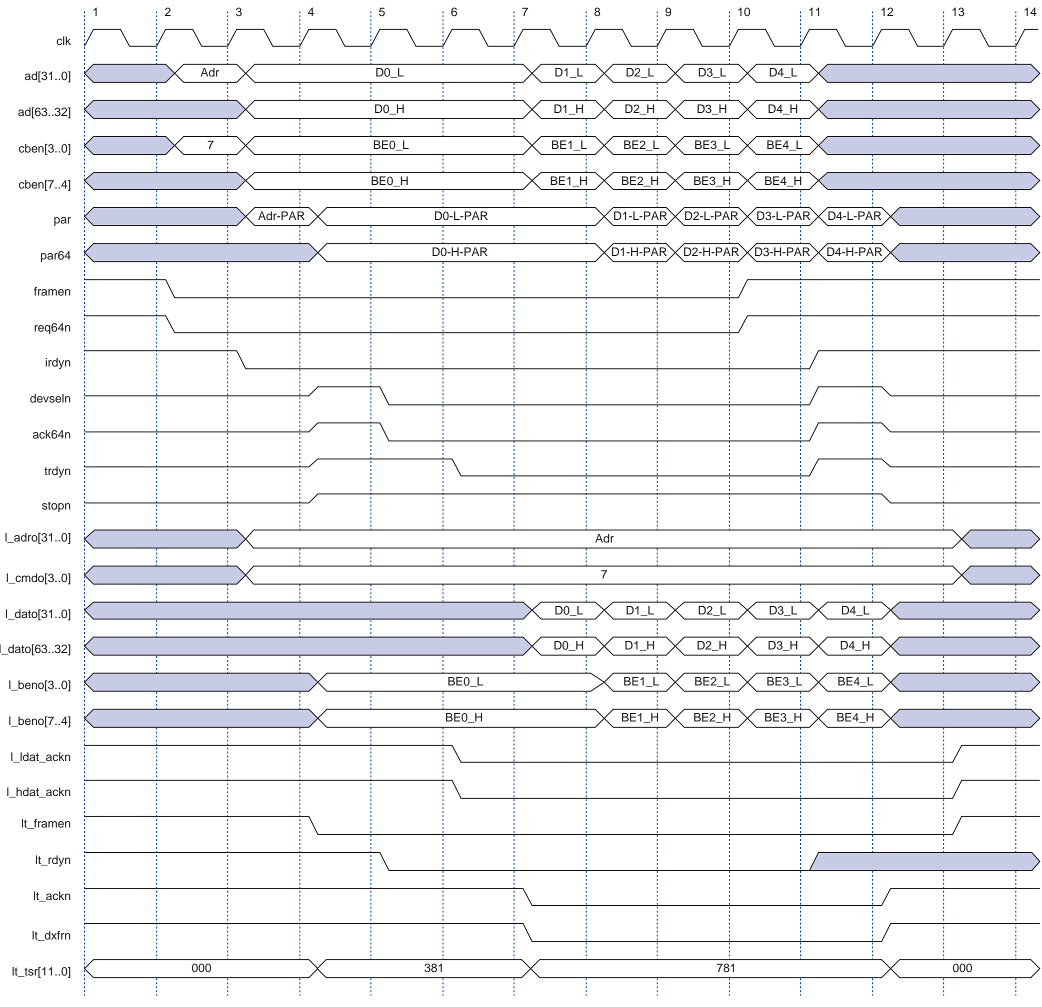
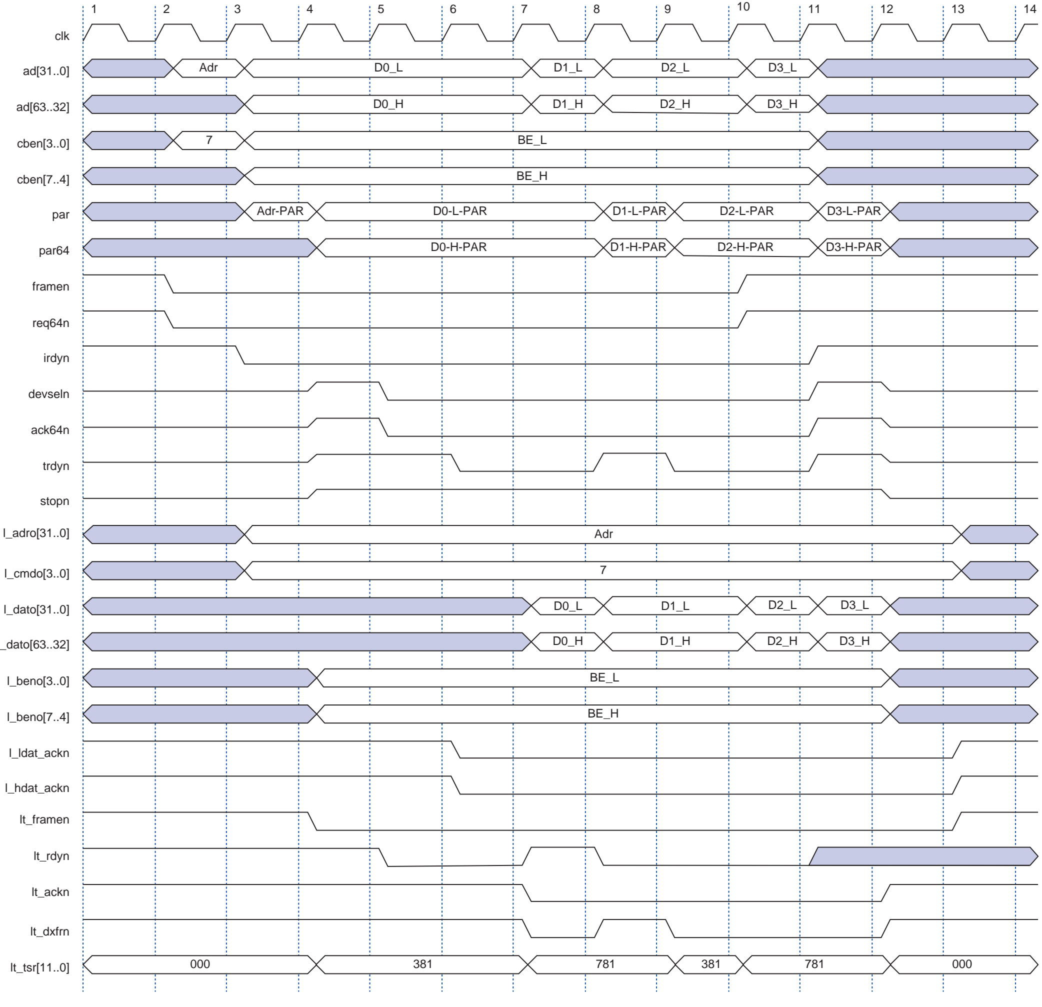


Figure 11 shows the same transaction as in Figure 10 with the PCI bus master asserting a wait-state. The PCI bus master asserts a wait state by deasserting the `irdyn` signal in clock 7. The effect of this wait state on the local side is shown in clock 8 with `lt_ackn` deasserted, and as a result `lt_dxfrn` is also deasserted. This transaction prevents data from being transferred to the local side in clock 8 because the internal pipeline of the `pci_c` function does not have valid data.

Figure 11. 64-Bit Target Burst Write Transaction with PCI Master Wait State

Figure 12 shows the same transaction as in Figure 10 with the local side asserting a wait-state. The local side deasserts `lt_rdyn` in clock 7. The `pci_c` function shows that deasserting `lt_rdyn` in clock 7 suspends the local side data transfer in clock 8 by deasserting the `lt_dxfrn` signal. Because the local side is unable to accept additional data in clock 8, `pci_c` deasserts `trdyn` in clock 8 as well, preventing PCI data from being transferred from the master device.

Figure 12. 64-Bit Target Burst Write Transaction with Local-Side Wait State



The local-side device must ensure that PCI latency rules are not violated while the MegaCore function waits to transfer data. If the local-side device is unable to meet the latency requirements, it must assert `lt_discn` to request that the MegaCore function terminate the transaction. The PCI target latency rules state that the time to complete the first data phase must not be greater than 16 PCI clocks, and the subsequent data phases must not take more than 8 PCI clocks to complete.

32-Bit Target Write Transactions

The pci_c function responds to three types of 32-bit target write transactions

- Memory write transaction
- I/O write transaction
- Configuration write transaction

The following sections explain the variations of each type in more detail.

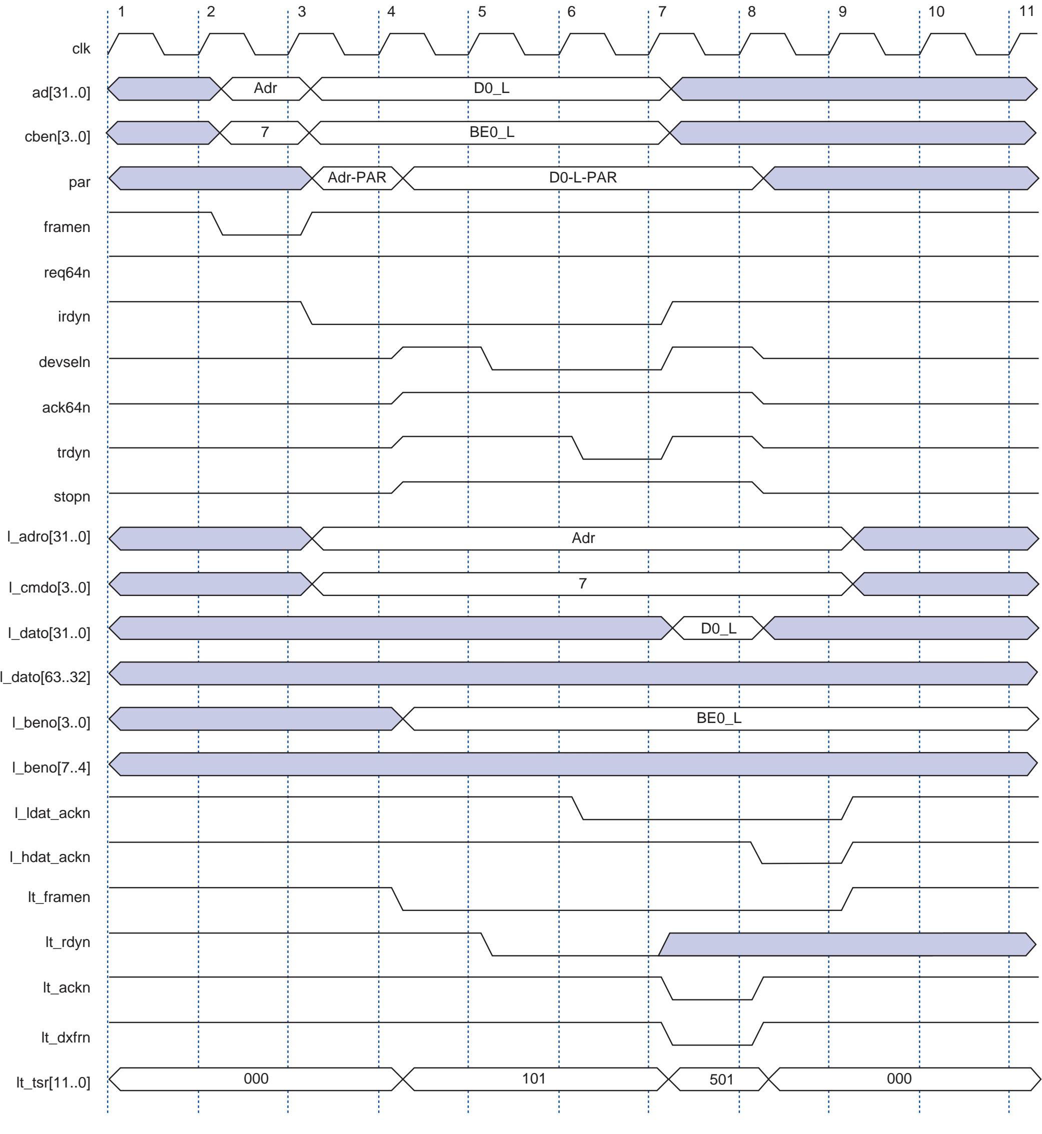
32-Bit Memory Write Transaction

Memory transactions are either single-cycle or burst. For memory transactions, the pci_c function always assumes a 64-bit local side. The pci_c function automatically transfers 32-bit data from the PCI side and drives that data to both the l_dato[31..0] and l_dato[63..32] buses. The pci_c function also indicates which DWORD the local side is transferring by asserting either l_ldat_ackn to indicate that the low DWORD is valid (ad[31..0]) or l_hdat_ackn to indicate that the high DWORD is valid (ad[63..32]). The pci_c function decodes whether the low or high DWORD is addressed by the master, based on the starting address of the transaction. If the address of the transaction is a QWORD boundary (ad[2..0] == B"000"), the first DWORD transferred is considered the low DWORD and pci_c asserts l_ldat_ackn accordingly; if the address of the transaction is not at QWORD boundary (ad[2..0] == B"100"), the first DWORD transferred is considered to be the high DWORD and the pci_c function asserts l_hdat_ackn accordingly.

Figure 13 shows a 32-bit single-cycle memory write transaction. The sequence of events in Figure 13 is exactly the same as in Figure 9, except for the following:

- During the address phase (clock 3) the master does not assert req64n because the transaction is 32 bits.
- The pci_c function does not assert ack64n when it asserts devseln.
- The local side is informed that the pending transaction is 32 bits because the lt_tsr[7] is not asserted while lt_framen is asserted in clock 4.

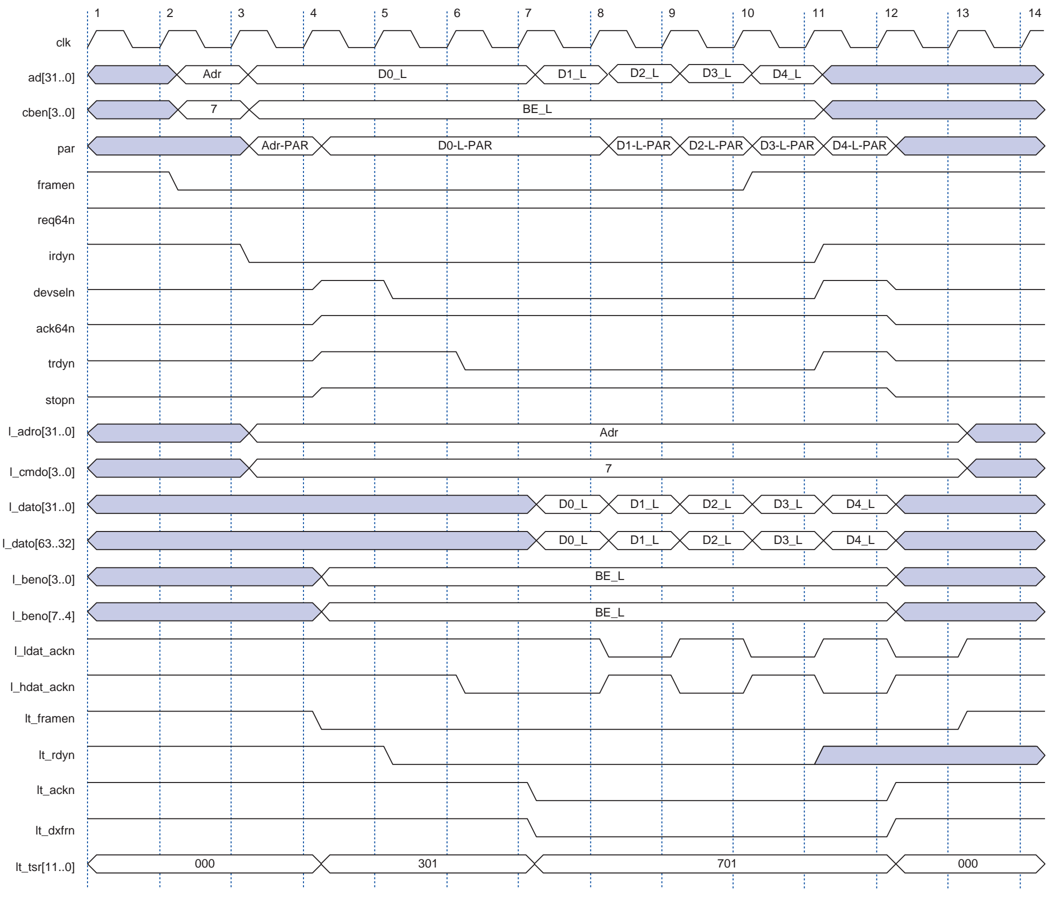
Figure 13. 32-Bit Single-Cycle Memory Write Transaction



In Figure 13, the local-side transfer occurs in clock 7 because `lt_dxfrn` is asserted during that clock. At the same time, `l_ldat_ackn` is asserted to indicate that the low DWORD is valid. This event occurs because the address used in the example is at QWORD boundary.

Figure 14 shows a 32-bit burst memory write transaction; the events are the same for Figure 10. The main difference between the two figures is that l_ldat_ackn and l_hdat_ackn toggle to indicate which DWORD is valid on the local side. In Figure 14, the high DWORD is transferred first because the address used is not a QWORD boundary. This situation occurs because l_hdat_ackn is asserted during clock 6 and continues to be asserted until the first DWORD is transferred on the local side during clock 7. The local side is informed that the pending transaction is a 32-bit burst because lt_tsr[7] is not asserted and lt_tsr[9] is asserted. If the local side cannot handle 32-bit burst transactions, it can disconnect after the first local transfer.

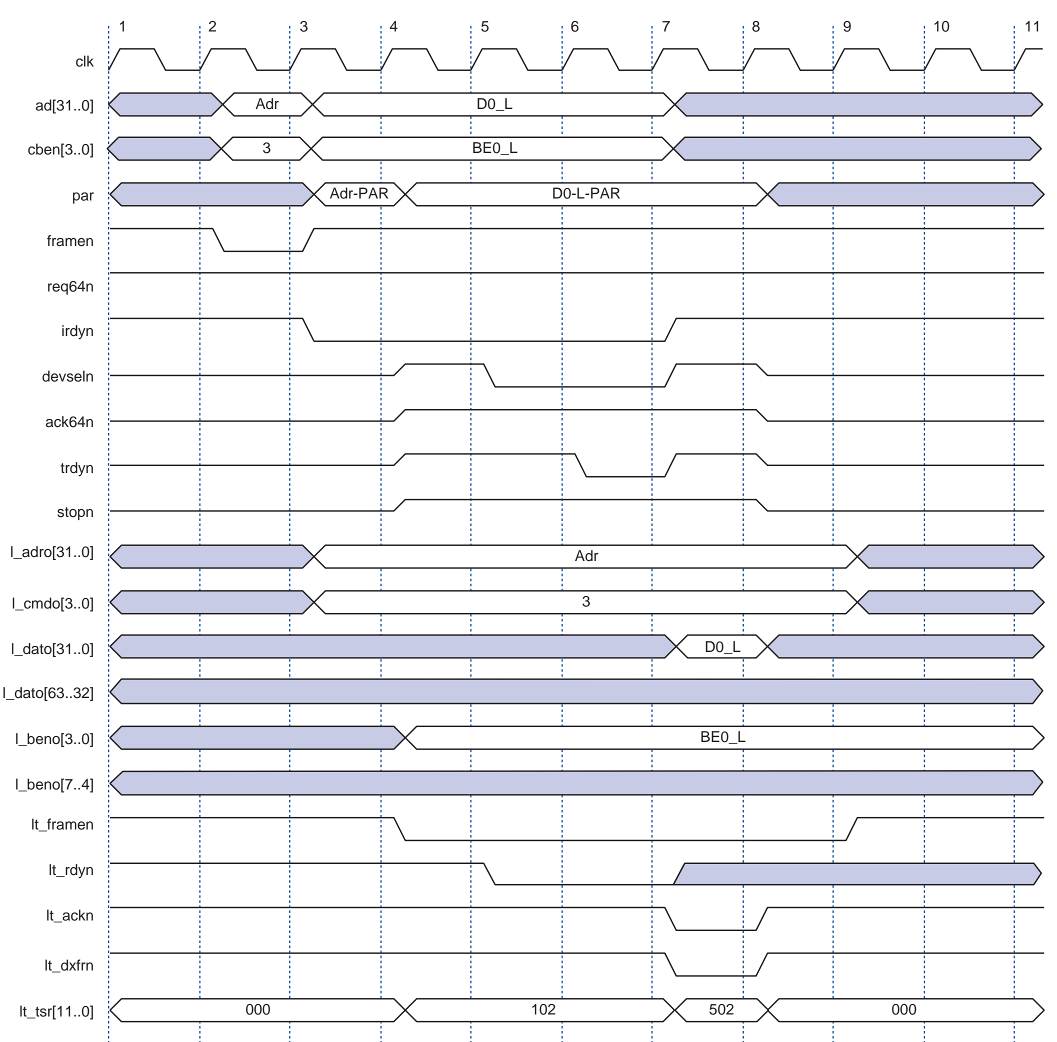
Figure 14. 32-Bit Burst Memory Write Transaction



I/O Write Transaction

Figure 15 shows a sample I/O write transaction. The sequence of events is the same as 32-bit single-cycle memory write transactions. The main distinction between the two transactions is the command on the `lt_cmdo[3..0]` bus.

Figure 15. 32-Bit I/O Write Transaction

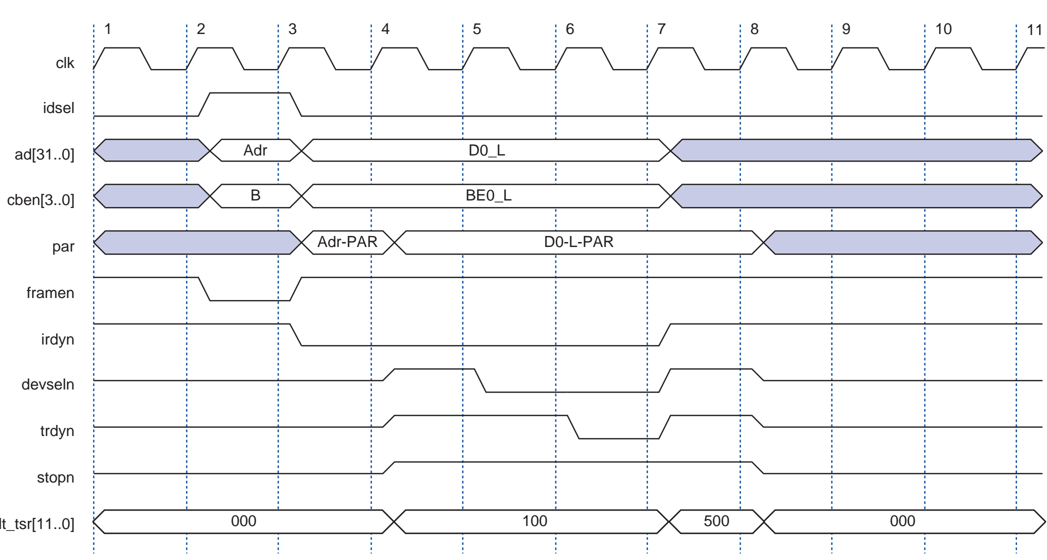


Configuration Write Transaction

Configuration cycles are automatically handled by `pci_c` and do not require local side actions. Figure 16 shows a typical configuration write transaction. The configuration write transaction is similar to a 32-bit single-cycle transaction, except for the following:

- During the address phase, `idsel` must be asserted in a configuration transaction
- Because the configuration write does not require local side actions, `pci_c` asserts `trdyn` independent from the `lt_rdyn` signal.

Figure 16. 32-Bit Configuration Write Transaction



The local side cannot retry, disconnect, or abort configuration cycles.

Target Transaction Terminations

For all transactions except configuration transactions, the local-side device can request a transaction to be terminated with one of several termination schemes defined by the ***PCI Local Bus Specification, Revision** 2.2*. The local-side device can use the `lt_discn` signal to request a retry or disconnect. These termination types are considered graceful terminations and are normally used by a target device to indicate that it is not ready to receive or supply the requested data. A retry termination forces the PCI master that initiated the transaction to retry the same transaction at a later time. A disconnect, on the other hand, does not force the PCI master to retry the same transaction.

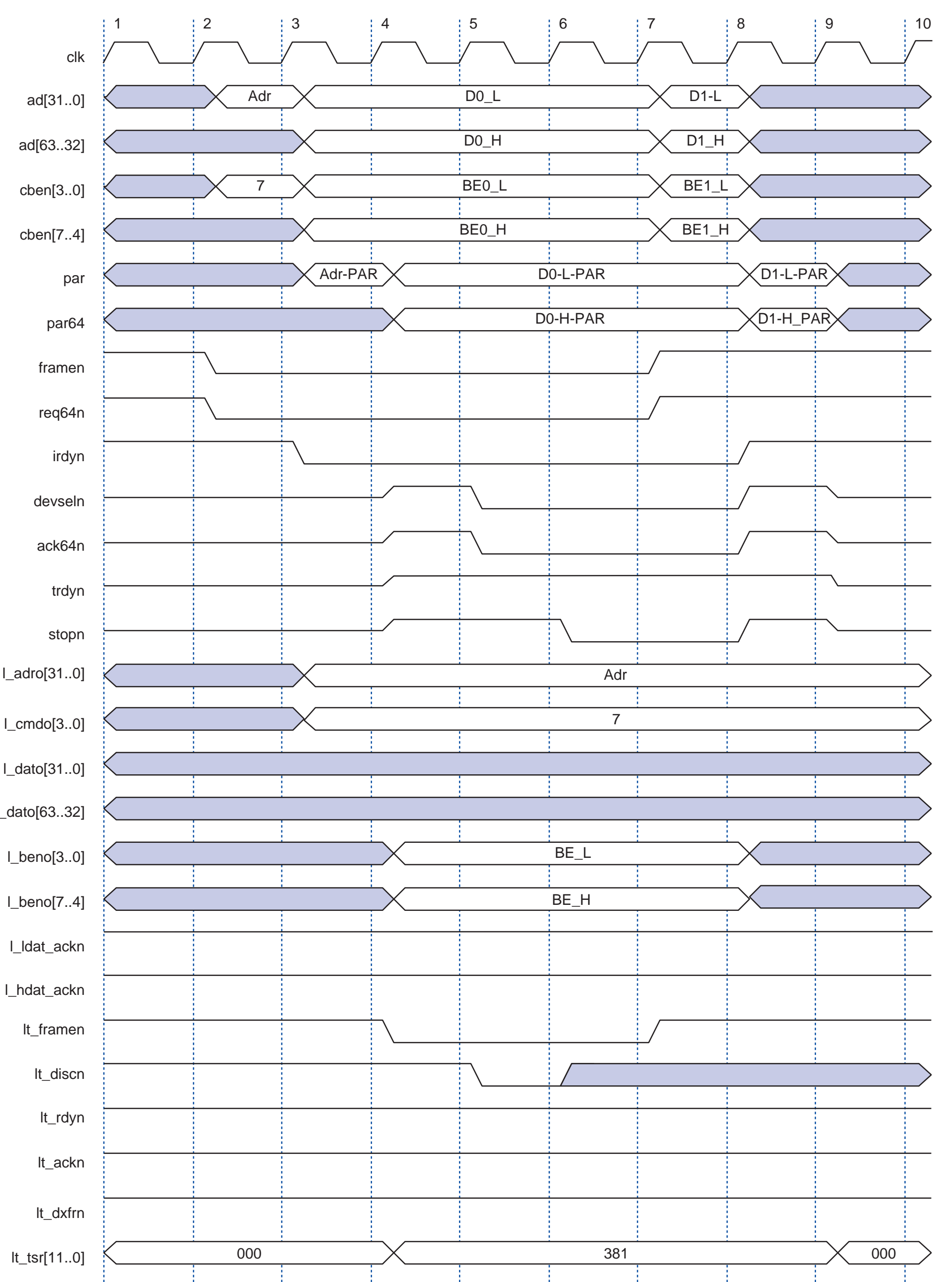
The local-side device can also request a target abort, which indicates that a catastrophic error has occurred in the device. This termination is requested by asserting `lt_abortn` during a target transaction other than a configuration transaction.

For more details on these termination types, refer to the ***PCI Local Bus Specification, Revision 2.2***.

Retry

The local-side device can request a retry, for example, because the device cannot meet the initial latency requirement or because there is a conflict for an internal resource. A target device signals a retry by asserting `devseln` and `stopn`, while deasserting `trdyn` before the first data phase. The local-side device can request a retry as long as it did not supply or request at least one data bit in a burst transaction. In a write transaction, the local-side device may request a retry by asserting `lt_discn` as long as it did not assert the `lt_rdyn` signal to indicate it is ready for a data transfer. If `lt_rdyn` is asserted, it can result in `pci_c` asserting the `trdyn` signal on the PCI bus. Therefore, asserting `lt_discn` forces a disconnect instead of a retry. In a read transaction, the local-side device can request a retry as long as data has not been transferred to the `pci_c` function. Figure 17 shows a write transaction where the MegaCore function issues a retry in response to the local side asserting `lt_discn` during clock 5.

Figure 17. Target Retry



Disconnect

A PCI target can signal a disconnect by asserting `stopn` and `devseln` after at least one data phase is complete. There are two types of disconnects: disconnect with data and disconnect without data. In a disconnect with data, `trdyn` is asserted while `stopn` is asserted. Therefore, more data phases are completed while the PCI bus master finishes the transaction. A disconnect without data occurs when the target device deasserts `trdyn` while `stopn` is asserted, thus ensuring that no more data phases are completed in the transaction. Depending on the sequence of `lt_rdyn` and `lt_discn` assertion, the `pci_c` function issues either a disconnect with data or disconnect without data.

Figure 18 shows an example of a disconnect with data which ensures that only a single data phase is completed during a burst write transaction. In Figure 18, both `lt_rdyn` and `lt_discn` are asserted in clock 5. This transaction informs the `pci_c` function that the local side is ready to accept data but also wants to disconnect. As a result, `pci_c` issues a disconnect with data and accepts only one data phase.

Figure 18. Single Data Phase Disconnect in a Burst Write Transaction

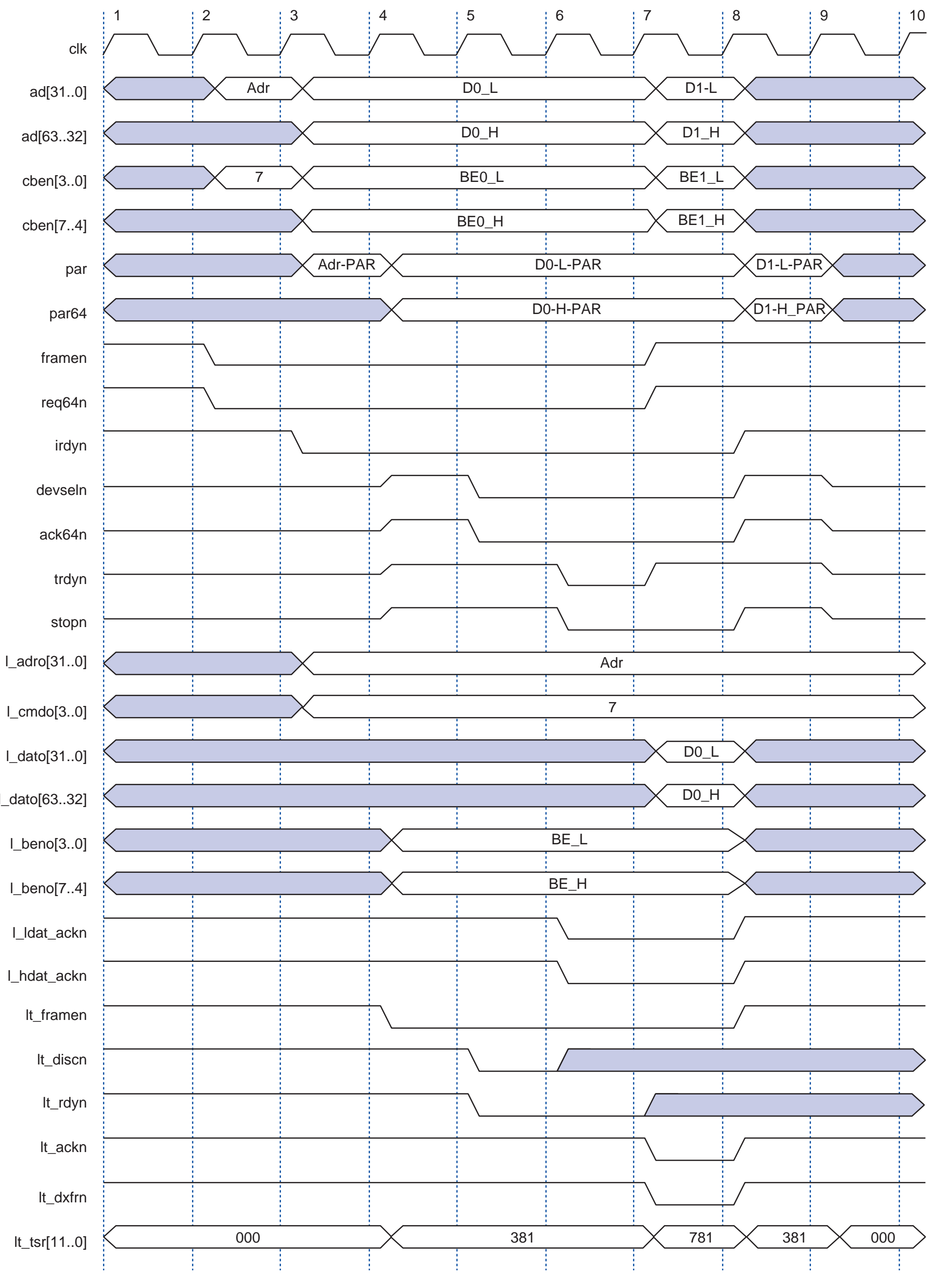


Figure 19 shows an example of a disconnect with data that ensures that only a single data phase is completed during a burst read transaction. In Figure 19, `lt_rdyn` is asserted in clock 4, and `lt_discn` is asserted in clock 5. This transaction ensures one data phase is completed on the local side before the `pci_c` function detects a disconnect request. Subsequently, `pci_c` issues a disconnect cycle on the PCI side to ensure that only one data phase is completed successfully.

Figure 19. Single Data Phase Disconnect in a Burst Read Transaction

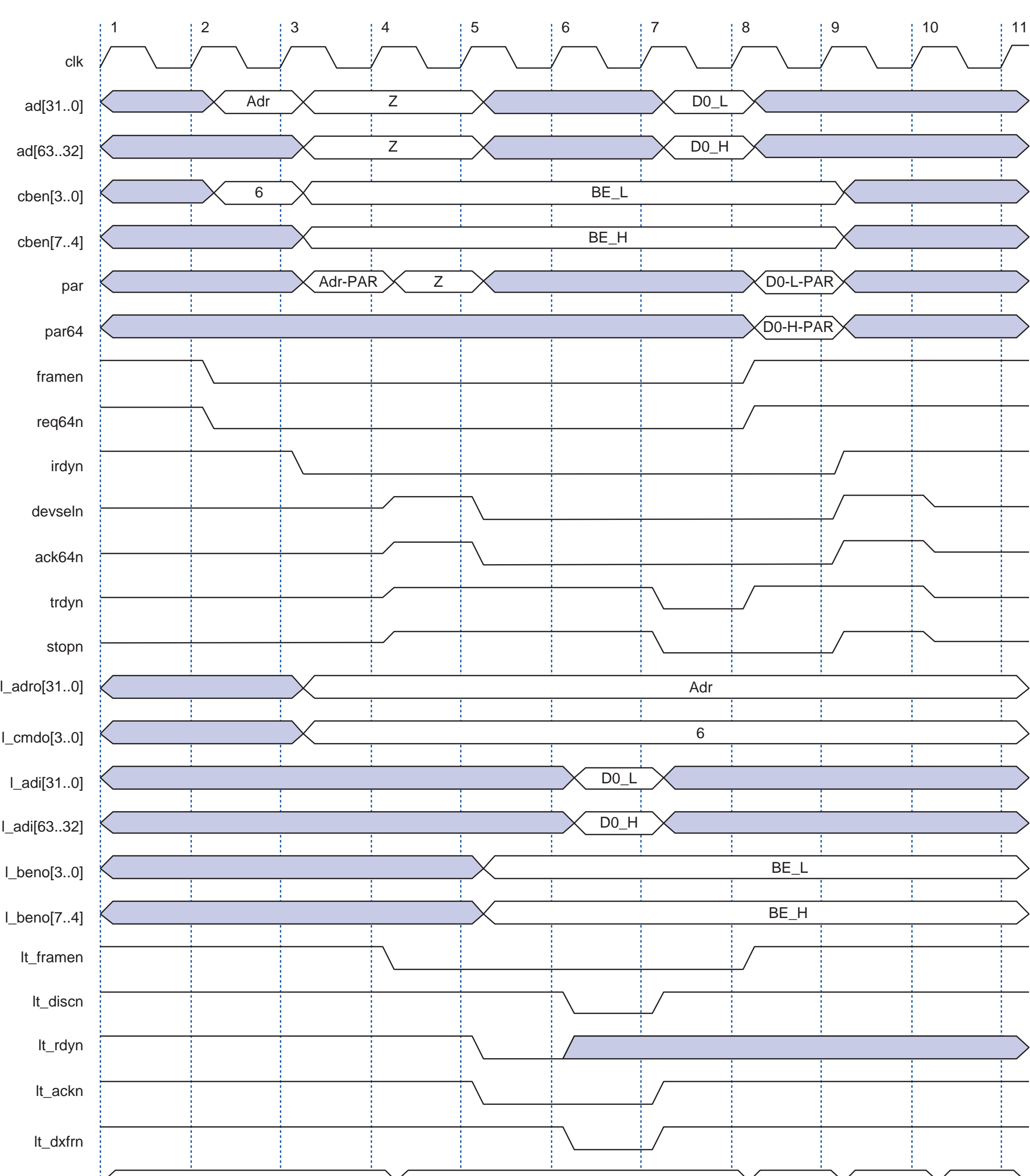
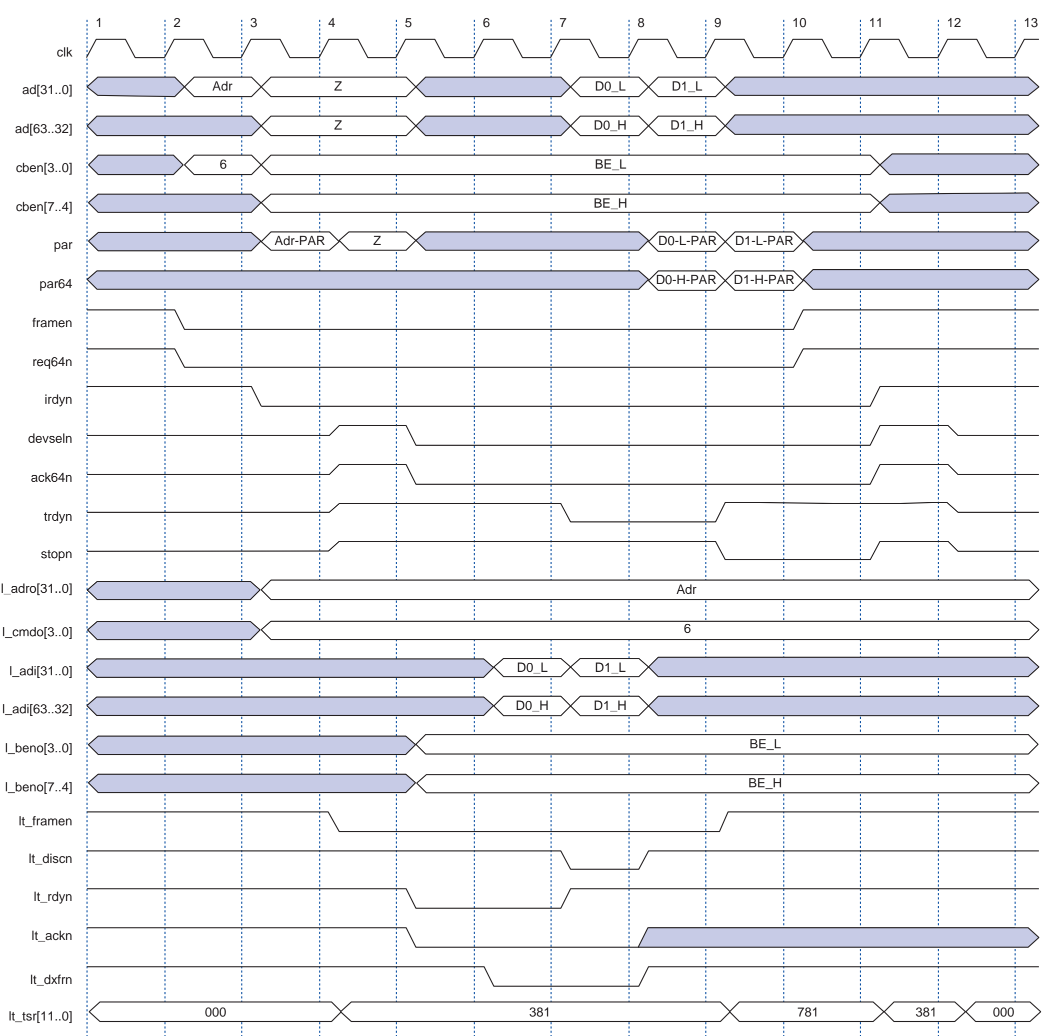


Figure 20 shows a disconnect without data during a burst read transaction.

Figure 20. Disconnect Without Data During a Burst Read Transaction



The *PCI Local Bus Specification, Revision 2.2* requires that a target device issues a disconnect if a burst transaction goes beyond its address range. In this case, the local-side device must request a disconnect. The local-side device must keep track of the current data transfer address; if the transfer exceeds its address range, the local side should request a disconnect by asserting `lt_discn`.

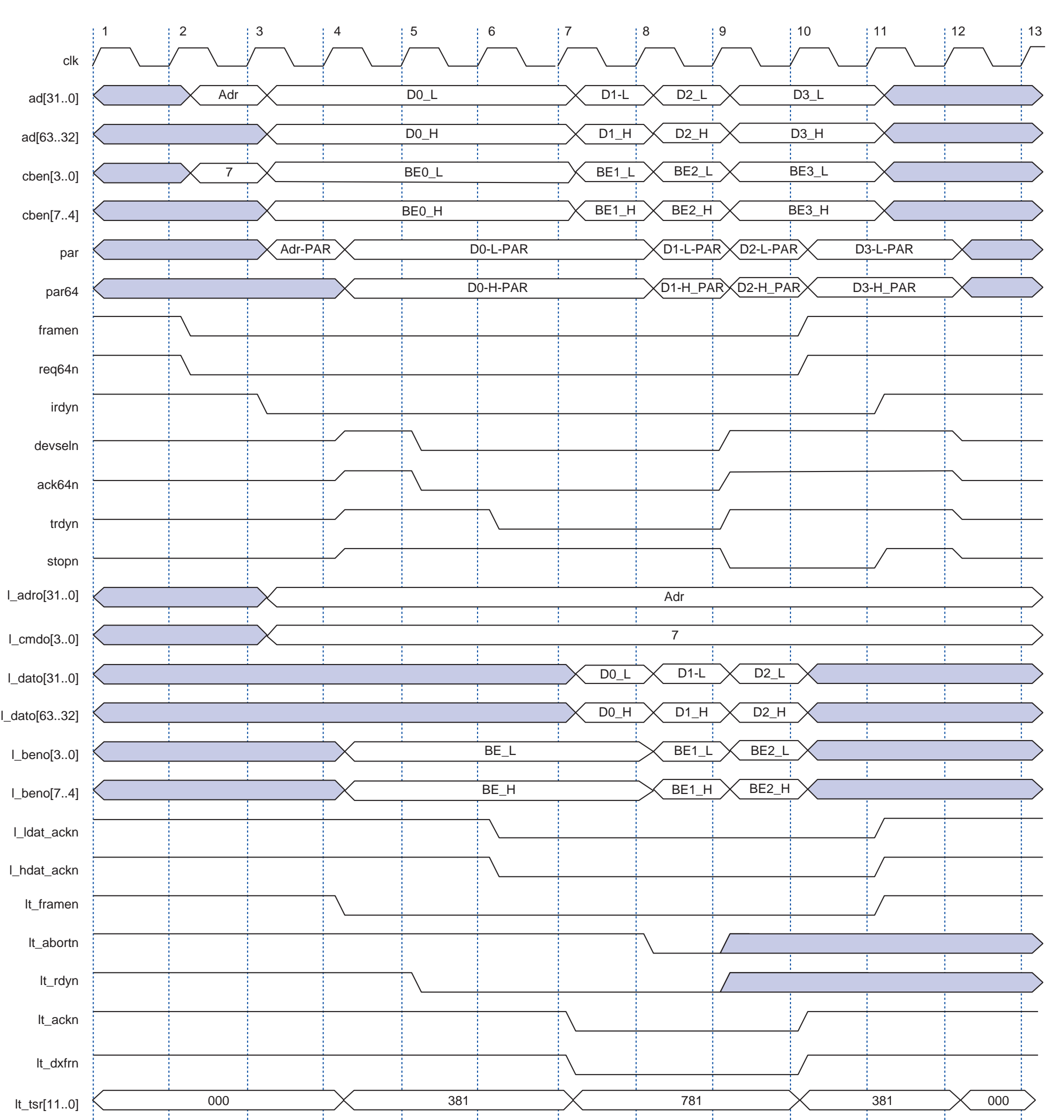
Target Abort

Target abort refers to an abnormal termination because either the local logic detected a fatal error, or the target will never be able to complete the request. An abnormal termination may cause a fatal error for the application that originally requested the transaction. A target abort allows the transaction to complete gracefully, thus preserving normal operation for other agents.

A target device issues an abort by deasserting `devseln` and `trdyn` and asserting `stopn`. A target device must set the `tabort_sig` bit in the PCI status register whenever it issues a target abort. See "Status Register" on page 60 for more details. Figure 21 shows the `pci_c` function issuing an abort during a burst write cycle.

The *PCI Local Bus Specification, Revision 2.2* requires that a target device issues an abort if the target device shares bytes in the same DWORD with another device, and the byte enable combination received byte requests outside its address range. This condition most commonly occurs during I/O transactions. The local-side device must ensure that this requirement is met, and if it receives this type of transaction, it must assert `lt_abortn` to request a target abort termination.

Figure 21. Target Abort



4 Specifications

Master Mode Operation

This section describes all supported master transactions for the pci_c function and includes waveform diagrams showing typical PCI cycles in master mode. The MegaCore functions support both 64-bit and 32-bit transactions. The pci_c function supports the following 64-bit PCI memory transactions:

- 64-bit memory burst master read
- 64-bit memory single-cycle master read
- 64-bit memory burst master write

The pci_c function also supports the following 32-bit PCI transactions:

- 32-bit memory burst master read
- 32-bit memory single-cycle master read
- Configuration master read
- I/O master read
- 32-bit memory burst master write
- Configuration master write
- I/O master write

A master operation begins when the local-side master interface asserts the lm_req64n signal to request a 64-bit transaction or the lm_req32n signal to request a 32-bit transaction. The pci_c function asserts the reqn signal to the PCI bus arbiter to request bus ownership. When the PCI bus arbiter grants the pci_c function bus ownership by asserting the gntn signal, the pci_c function asserts the lm_adr_ackn signal on the local side to acknowledge the transaction address and command. The local side must provide the address on l_adi[31..0] and the command on l_cbeni[3..0] during the same clock cycle when the lm_adr_ackn signal is asserted.

The pci_c function begins the transaction with the address phase by asserting framen and driving the transaction address on ad[31..0] and command on cben[3..0]. During the address phase, the local side must also provide the byte-enable values on l_cbeni[7..0] for the first data phase because pci_c is required to drive them on the PCI bus in the following clock. During burst transactions, the local-side must ensure that l_cbeni[7..0] is B"00000000".

After the address phase, the local side asserts lm_rdyn to signal that it is ready to input data from the PCI side in a master read, or it is ready to output data to the PCI side in a master write. The pci_c function asserts lm_ackn to acknowledge that the PCI side is ready to output data to the local side in a master read, or it is ready to input data from the local side in a master write. In a master read, the pci_c function outputs data to the local side through the l_dato[63..0]data lines. While in a master write transaction, the pci_c function inputs data from the local side through the l_adi[63..0]data lines. Valid data is transferred on the local side during the same clock cycle when the lm_dxfrn signal is asserted by the pci_c function. The pci_c function asserts l_ldat_ackn and l_hdat_ackn to signal whether the lower bits [31..0] or the upper bits [63..32] or both are being sent from the PCI side to the local side in a master read, or in the opposite direction in a master write. Therefore, l_ldat_ackn, l_hdat_ackn, and lm_dxfrn signals can be used to qualify when valid data is transferred from the local side.

The pci_c function can generate any transaction in master mode because the local side provides the pci_c function with the exact command. When the local side requests I/O or configuration cycles, the pci_c function automatically issues a single-cycle read/write transaction. In all other transactions, the local side must assert lm_lastn to inform the pci_c function when to end the transaction. The pci_c function treats memory write and invalidate, memory read multiple, and memory read line commands in a similar manner to the corresponding memory write/read commands. Therefore, the local side must implement any special handling required by these commands. The pci_c function outputs the cache line size register value to the local side for this purpose.

The local-side device may require a long time to transfer data to/from the pci_c function during a burst transaction. The local-side device must ensure that PCI latency rules are not violated while the pci_c function waits for data. Therefore, the local-side device must not insert more than eight wait states before asserting lm_rdyn.

The pci_c function uses the transaction status register outputs (lm_tsr[9..0]) to inform the local-side application of the transaction status. See "Status Register" on page 60 for a description of each bit in this bus. The following sections provide additional details about pci_c master mode operation.

4 Specifications

64-Bit Master Read Transactions

In master mode, the pci_c function supports two types of 64-bit read transactions:

- Burst memory read
- Single-cycle read

The burst memory read and single-cycle read transactions differ in the following ways:

- The burst transaction transfers more data.
- The l_cbeni[3..0] bus can only enable specific bytes in the lower DWORD during single-cycle transactions.
- The l_cbeni[7..4] bus can only enable specific bytes in the upper DWORD during single-cycle transactions.

For both types of transactions, the sequence of events is the same and can be divided into the following steps:

1. The local side asserts lm_req64n to request a 64-bit transaction. Consequently, the pci_c function asserts reqn to request bus ownership from the PCI arbiter.

2. When the PCI arbiter grants bus ownership by asserting the gntn signal, the pci_c function asserts lm_adr_ackn on the local side to acknowledge the transaction address and command. During the same clock cycle when lm_adr_ackn in asserted, the local side should provide the address on l_adi[31..0] and the command on l_cbeni[3..0]. At the same time, the pci_c function turns on the drivers for framen and req64n.

3. The pci_c function begins the PCI address phase by asserting framen and req64n and driving the address and the command on ad[31..0] and cben[3..0]. Also, during the address phase, the local side should provide the byte enables for the transaction on l_cbeni[7..0]. At the same time, the pci_c function turns on the driver for irdyn.

4. A turn-around cycle on the ad[63..0] occurs during the clock immediately following the address phase. During the turn-around cycle, the pci_c function tri-states ad[63..0], but drives the correct byte enables on cben[7..0] for the first data phase. This process is necessary because the pci_c function must release the bus so another PCI agent can drive it.

5. If the address of the transaction matches one of the base address registers of a PCI target, the PCI target should assert `devseln` to claim the transaction. One or more data phases follow next, depending on the type of read transaction.

The `pci_c` function treats memory read, memory read multiple, and memory read line commands in the same way. Any additional requirements for the memory read multiple and memory read line commands must be implemented by the local-side application.

Figure 22 shows the waveform for a 64-bit zero wait state master burst memory read transaction. In this transaction, three 64-bit words are transferred from the PCI side to the local side.

Figure 22. 64-Bit Zero-Wait-State Master Burst Memory Read Transaction

1 2 3 4 5 6 7 8 9 10 11 12 13

clk
reqn
gntn
ad[31..0] 0 Adr Z D0_L D1_L D2_L Z
ad[63..32] 0 Z D0_H D1_H D2_H Z
cben[3..0] 0 6 BE_L Z
cben[7..4] 0 BE_H Z
par Adr-PAR Z D0-H-PAR D1-H-PAR D2-H-PAR
par64 Z D0-H-PAR D1-H-PAR D2-H-PAR
framen
req64n
irdyn
devseln
ack64n
trdyn
stopn
lm_req64n
l_adi[31..0] Adr
l_cbeni[3..0] 6 BE_L
l_cbeni[7..4] BE_H
lm_adr_ackn
lm_lastn
lm_rdyn
l_ldat_ackn
l_hdat_ackn
lm_ackn
lm_dxfrn
l_dato[31..0] D0_L D1_L D2_L
l_dato[63..32] D0_H D1_H D2_H
lm_tsr[9..0] 000 001 002 004 008 208 308 200 000

Table 25 shows the sequence of events for a 64-bit zero-wait-state master burst memory read transaction.

Table 25. 64-Bit Zero Wait State Master Burst Memory Read Transaction (Part 1 of 3)

Clock Cycle	Event
1	The local side asserts lm_req64n to request a 64-bit transaction.
2	The pci_c function outputs reqn to the PCI bus arbiter to request bus ownership. At the same time, the pci_c function asserts lm_tsr[0] to indicate to the local side that the pci_c master is requesting the PCI bus.
3	The PCI bus arbiter asserts gntn to grant the PCI bus to the pci_c function. Although Figure 22 shows that the grant occurs immediately and the PCI bus is idle at the time gntn is asserted, this action may not occur immediately in a real transaction. The pci_c function waits for gntn to be asserted while the PCI bus is idle before it proceeds. A PCI bus idle state occurs when both framen and irdyn are deasserted.
5	The pci_c function turns on its output drivers, getting ready to begin the address phase. The pci_c function also asserts lm_adr_ackn to indicate to the local side that it has acknowledged its request. During the same clock cycle, the local side should provide the PCI address on l_adi[31..0] and the PCI command on l_cbeni[3..0]. The pci_c function continues to assert its reqn signal until the end of the address phase. The pci_c function also asserts lm_tsr[1] to indicate to the local side that the PCI bus has been granted.
6	The pci_c function begins the 64-bit memory read transaction with the address phase by asserting framen and req64n. At the same time, the local side must provide the byte enables for the transaction on l_cbeni[7..0]. The local side also asserts lm_rdyn to indicate that it is ready to accept data. The pci_c function asserts lm_tsr[2] to indicate to the local side that the PCI bus is in its address phase.
7	The pci_c function asserts irdyn to inform the target that pci_c is ready to receive data. The pci_c function asserts irdyn regardless if the local side asserts lm_rdyn to indicate that it is ready to accept data, only for the first data phase on the PCI side. For subsequent data phases, the pci_c function will not assert irdyn unless the local side is ready to accept data. The target claims the transaction by asserting devseln. In this case, the target performs a fast address decode. The target also asserts ack64n to inform the pci_c function that it can transfer 64-bit data. During this clock cycle, the pci_c function also asserts lm_tsr[3] to inform the local side that it is in data transfer mode.

4 Specifications

Table 25. 64-Bit Zero Wait State Master Burst Memory Read Transaction (Part 2 of 3)	
Clock Cycle	Event
8	The target asserts trdyn to inform the pci_c function that it is ready to transfer data. Because the pci_c function has already asserted irdyn, a data phase is completed on the rising edge of clock 9. At the same time, lm_tsr[9] is asserted to indicate to the local side that the target can transfer 64-bit data.
9	The pci_c function asserts lm_ackn to inform the local side that pci_c has registered data from the PCI side on the previous cycle and is ready to send the data to the local side master interface. Because lm_rdyn was asserted in the previous cycle and lm_ackn is asserted in the current cycle, pci_c asserts lm_dxfrn. The assertion of the lm_dxfrn, l_ldat_ackn, and l_hdat_ackn signals indicate to the local side that valid data is available on the l_dato[63..0] data lines. Because irdyn and trdyn are asserted, another data phase is completed on the PCI side on the rising edge of clock 10. On the local side, the lm_lastn signal is asserted. Because lm_lastn, irdyn, and trdyn are asserted during this clock cycle, this action guarantees to the local side that, at most, two more data phases will occur on the PCI side: one during this clock cycle and another on the following clock cycle (clock 10). The last data phase on the PCI side takes place during clock 10. The pci_c function also asserts lm_tsr[8] in the same clock to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock.
10	Because lm_lastn was asserted and a data phase was completed in the previous cycle, framen and req64n are deasserted, while irdyn and trdyn are asserted. This action indicates that the last data phase is completed on the PCI side on the rising edge of clock 11. On the local side, the pci_c function continues to assert lm_ackn, informing the local side that pci_c has registered data from the PCI side on the previous cycle and is ready to send the data to the local side master interface. Because lm_rdyn was asserted in the previous cycle and lm_ackn is asserted in the current cycle, pci_c asserts lm_dxfrn. The assertion of the lm_dxfrn, l_ldat_ackn, and l_hdat_ackn signals indicate to the local side that another valid data bit is available on the l_dato[63..0] data lines. The local side has now received two valid 64-bit data. The pci_c function continues to assert lm_tsr[8] informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.

Table 25. 64-Bit Zero Wait State Master Burst Memory Read Transaction (Part 3 of 3)

Clock Cycle	Event
11	On the PCI side, irdyn, devseln, ack64n, and trdyn are deasserted, indicating that the current transaction on the PCI side is completed. There will be no more data phases. On the local side, the pci_c function continues to assert lm_ackn, informing the local side that pci_c has registered data from the PCI side on the previous cycle and is ready to send the data to the local side master interface. Because lm_rdyn was asserted in the previous cycle and lm_ackn is asserted in the current cycle, pci_c asserts lm_dxfrn. The assertion of the lm_dxfrn, l_ldat_ackn, and l_hdat_ackn signals indicate to the local side that another valid data is available on the l_dato[63..0] data lines. The local side has now received three valid 64-bit data. Because the local side has received all the data that was registered from the PCI side, the local side can now deassert lm_rdyn. Otherwise, if there is still some data that has not been transferred from the PCI side to the local side, then lm_rdyn must continue to be asserted. The pci_c function continues to assert lm_tsr[8] informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.
12	The pci_c function deasserts lm_tsr[3], informing the local side that the data transfer mode is completed. Therefore, lm_ackn and lm_dxfrn are also deasserted.

64-Bit Master Burst Memory Read Transaction with Local-Side Wait State

Figure 23 shows the same transaction as in Figure 22 with the local side asserting a wait state. The local side deasserts lm_rdyn in clock 9. Consequently, on the following clock cycle (clock 10), the pci_c function suspends data transfer on the local side by deasserting the lm_dxfrn signal and on the PCI side by deasserting the irdyn signal.

Figure 23. 64-Bit Master Burst Memory Read Transaction with Local Wait State

64-Bit Master Burst Memory Read Transaction with PCI Wait State

Figure 24 shows the same transaction as in Figure 22 with the PCI bus target asserting a wait state. The PCI target asserts a wait state by deasserting `trdyn` in clock 9. Consequently, on the following clock cycle (clock 10), the `pci_c` function deasserts the `lm_ackn` and `lm_dxfrn` signal on the local side. Data transfer is suspended on the PCI side in clock 9 and on the local side in clock 10.

Figure 24. 64-Bit Master Burst Memory Read Transaction with PCI Wait State

64-Bit Master Single-Cycle Memory Read Transaction

The pci_c function can perform 64-bit master single-cycle memory read transactions. If you are using a purely 64-bit system and the local side wants to transfer one 64-bit data, then Altera recommends that you perform a 64-bit single-cycle memory read transaction. However, if you are not using a purely 64-bit system and the local side wants to transfer one 64-bit data, Altera recommends that a 32-bit burst memory read transaction is performed.

Figure 25 shows the same transaction as in Figure 22 with just one data phase. In clock 6, framen and req64n are asserted to begin the address phase. At the same time, the local side should assert the lm_lastn signal on the local side to indicate that it wants to transfer only one 64-bit data. In a real application, in order to indicate a single-cycle 64-bit data transfer, the lm_lastn signal can be asserted on any clock cycle between the assertion of lm_req64n and the address phase.

Figure 25. 64-Bit Master Single-Cycle Memory Read Transaction

1 2 3 4 5 6 7 8 9 10 11 12 13

clk
reqn
gntn
ad[31..0] 0 Adr Z D0_L Z
ad[63..32] 0 Z D0_H Z
cben[3..0] 0 6 BE_L Z
cben[7..4] 0 BE_H Z
par Adr-PAR Z D0-L-PAR
par64 Z D0-H-PAR
framen
req64n
irdyn
devseln
ack64n
trdyn
stopn
lm_req64n
l_adi[31..0] Adr
l_cbeni[3..0] 6 BE_L
l_cbeni[7..4] BE_H
lm_adr_ackn
lm_lastn
lm_rdyn
l_ldat_ackn
l_hdat_ackn
lm_ackn
lm_dxfrn
l_dato[31..0] D0_L
l_dato[63..32] D0_H
lm_tsr[9..0] 000 001 002 004 008 208 308 000

32-Bit Master Read Transactions

In master mode, the pci_c function supports three types of 32-bit read transactions:

- Memory read transactions
- I/O read transactions
- Configuration read transactions

The 32-bit master read transactions are similar to 64-bit master read transactions, but the upper address ad[63..32] and the upper command/byte enables cben[7..4] are invalid.

32-Bit PCI & 64-Bit Local-Side Master Burst Memory Read Transaction

Figure 26 shows the same transaction as in Figure 22, but the PCI target cannot transfer 64-bit transactions. In this transaction, the local-side master interface requests a 64-bit transaction by asserting lm_req64n. The pci_c function asserts req64n on the PCI side. However, the PCI target cannot transfer 64-bit data, and therefore does not assert ack64n in clock 7. Since this is the case, the upper address ad[63..32] and the upper command/byte enables cben[7..4] are invalid.

Also, because the PCI side is 32 bits wide and the local side is 64 bits wide, the l_ldat_ackn and l_hdat_ackn signals toggle to indicate whether the lower l_dato[31..0] or the upper l_dato[63..32] have valid data. Along with these signals, valid data transfer on the local side is qualified when lm_dxfrn is asserted.

Because the local-side master interface is 64 bits and the PCI target is only 32 bits, these transactions always begin on 64-bit boundaries with the first data being sent to the lower l_dato[31..0] and the next DWORD being sent to the upper l_dato[63..32]. These transactions should end with the last data being sent to the upper l_dato[63..32] bus.

Figure 26. 32-Bit PCI & 64-Bit Local-Side Master Burst Memory Read Transaction

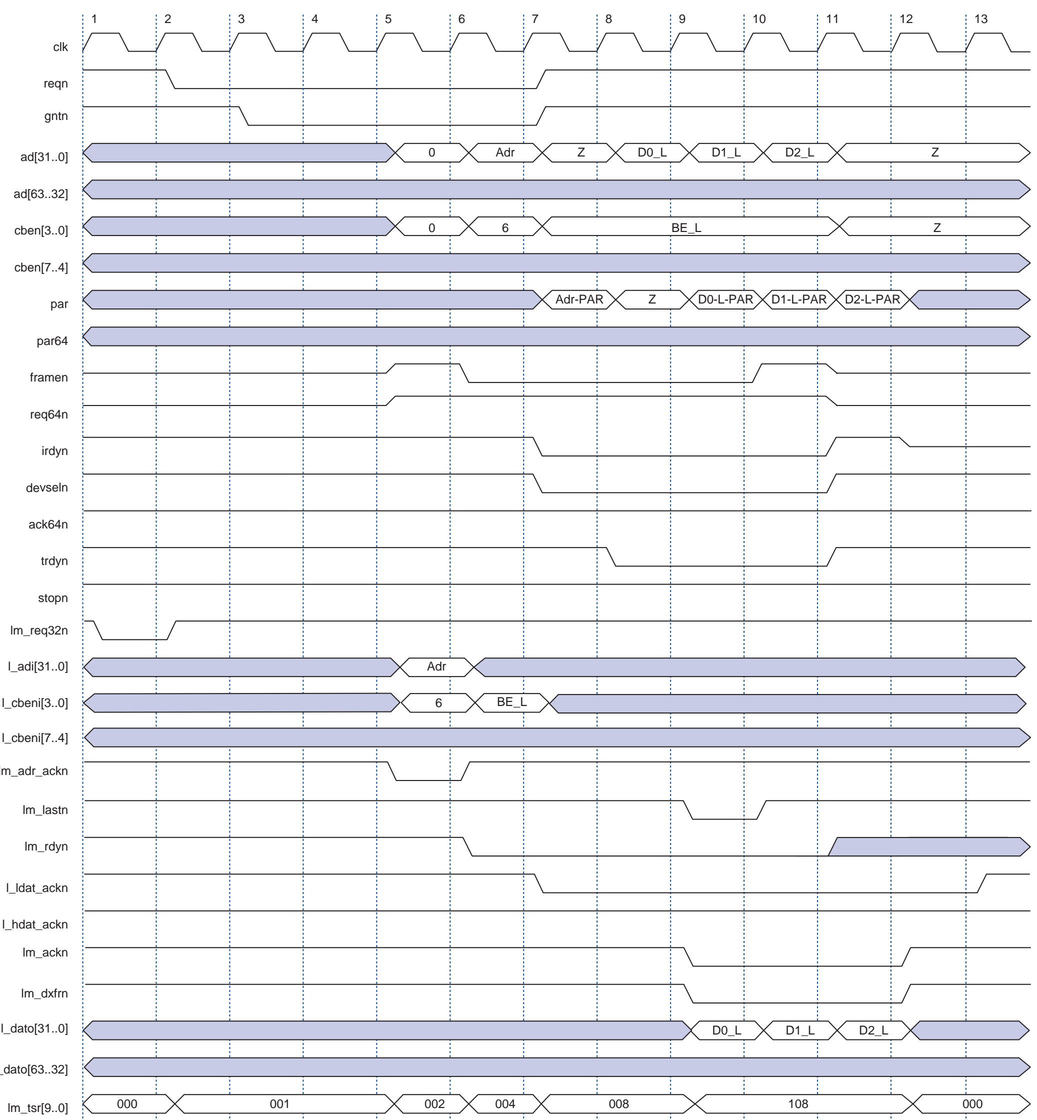
1 2 3 4 5 6 7 8 9 10 11 12 13

clk
reqn
gntn
ad[31..0] 0 Adr Z D0_L D1_L D2_L D3_L Z
ad[63..32]
cben[3..0] 0 6 BE_L Z
cben[7..4]
par Adr-PAR Z D0-L-PAR D1-L-PAR D2-L-PAR D3-L-PAR
par64
framen
req64n
irdyn
devseln
ack64n
trdyn
stopn
lm_req64n
l_adi[31..0] Adr
l_cbeni[3..0] 6 BE_L
l_cbeni[7..4] BE_H
lm_adr_ackn
lm_lastn
lm_rdyn
l_ldat_ackn
l_hdat_ackn
lm_ackn
lm_dxfrn
l_dato[31..0] D0_L D1_L D2_L D3_L
l_dato[63..32] D0_L D1_L D2_L D3_L
lm_tsr[9..0] 000 001 002 004 008 108 000

32-Bit PCI & 32-Bit Local-Side Master Burst Memory Read Transaction

Figure 27 shows the same transaction as in Figure 22, but the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. The `pci_c` function does not assert `req64n` on the PCI side. Therefore, the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid.

Figure 27. 32-Bit PCI & 32-Bit Local-Side Master Burst Memory Read Transaction



32-Bit PCI & 32-Bit Local Side Single-Cycle Memory Read Transaction

Figure 28 shows the same transaction as in Figure 27, but the local side master interface transfers only one data phase. This waveform also applies to the following types of single-cycle transactions:

- I/O read
- Configuration read

Figure 28. 32-Bit PCI & 32-Bit Local-Side Single-Cycle Memory Read Transaction

1 2 3 4 5 6 7 8 9 10

clk
reqn
gntn
ad[31..0] 0 Adr Z D0_L Z
ad[63..32]
cben[3..0] 0 6 BE_L Z
cben[7..4]
par Adr-PAR Z D0-L-PAR
par64
framen
req64n
irdyn
devseln
ack64n
trdyn
stopn
lm_req32n
l_adi[31..0] Adr
l_cbeni[3..0] 6 BE_L
l_cbeni[7..4]
lm_adr_ackn
lm_lastn
lm_rdyn
l_ldat_ackn
l_hdat_ackn
lm_ackn
lm_dxfrn
l_dato[31..0] D0_L
l_dato[63..32]
lm_tsr[9..0] 000 001 002 004 008 108 000

64-Bit Master Write Transactions

In master mode, the pci_c function supports 64-bit memory write transactions. The pci_c function does not perform 64-bit, memory single-cycle write transactions. If the local side wants to transfer single 64-bit data, Altera recommends performing a 32-bit memory burst write.

For this type of transaction, the sequence of events can be divided into the following steps:

1. The local side asserts lm_req64n to request a 64-bit transaction. Consequently, the pci_c function asserts reqn to request bus ownership from the PCI arbiter.

2. When the PCI arbiter grants bus ownership by asserting the gntn signal, the pci_c function asserts lm_adr_ackn on the local side to acknowledge the transaction's address and command. During the same clock cycle when lm_adr_ackn is asserted, the local side should provide the address on l_adi[31..0] and the command on l_cbeni[3..0]. At the same time, the pci_c function turns on the drivers for framen and req64n.

3. The pci_c function begins the PCI address phase by asserting framen and req64n and driving the address and the command on ad[31..0] and cben[3..0]. Also, during the address phase, the local side should provide the byte enables for the transaction on l_cbeni[7..0]. At the same time, the pci_c function turns on the driver for irdyn.

4. If the address of the transaction matches one of the base address registers of a PCI target, the PCI target should assert devseln to claim the transaction. One or more data phases follow next, depending on the type of write transaction.

The pci_c function treats memory write and memory write and invalidate in the same way. Any additional requirements for the memory write and invalidate command must be implemented by the local-side application.

64-Bit Zero-Wait-State Master Burst Memory Write Transaction

Figure 29 shows the waveform for a 64-bit zero wait state master burst memory write transaction. In this transaction, three 64-bit words are transferred from the local side to the PCI side.

Figure 29. 64-Bit Zero-Wait-State Master Burst Memory Write Transaction

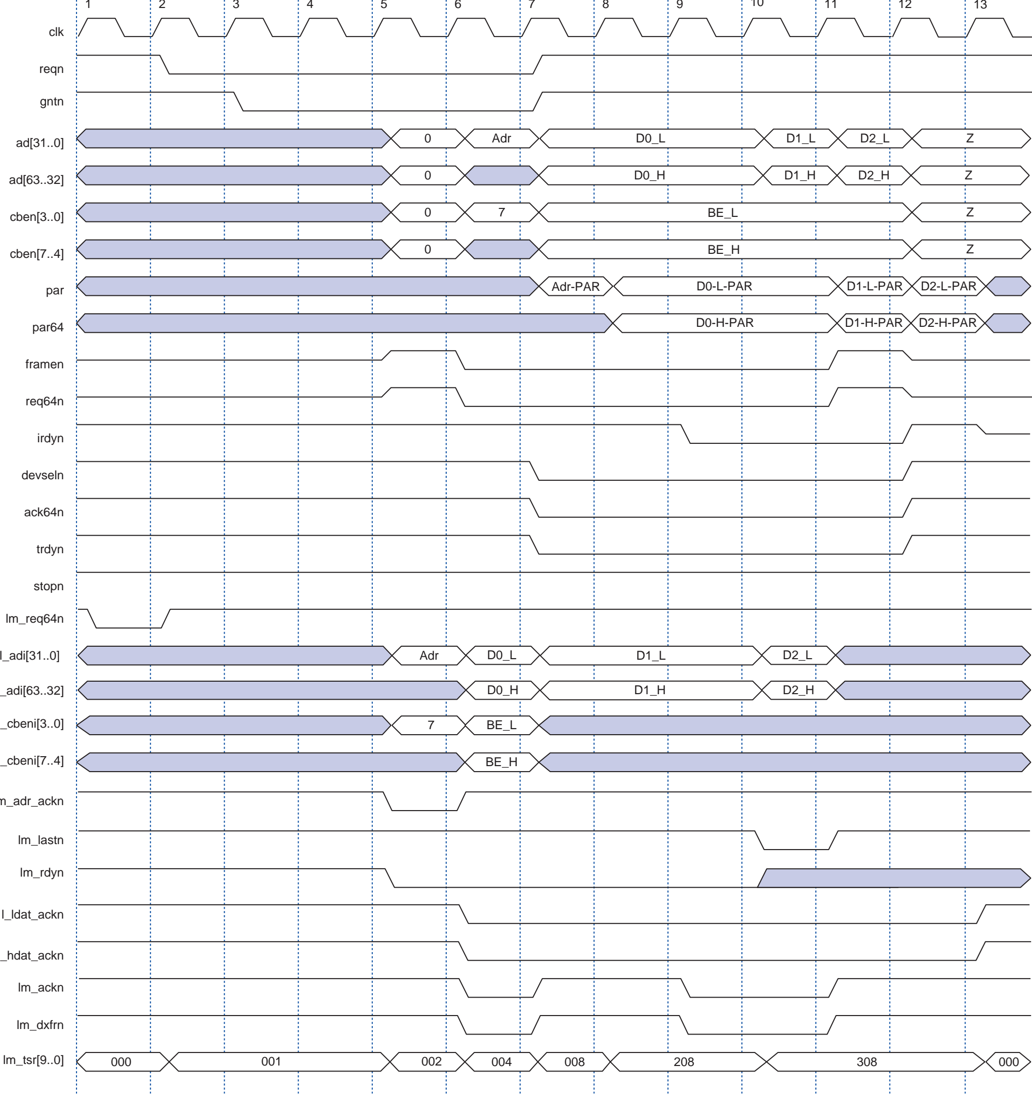


Table 26 shows the sequence of events for a 64-bit zero wait state master burst memory write transaction.

Table 26. 64-Bit Zero Wait State Master Burst Memory Write Transaction (Part 1 of 3)

Clock Cycle	Event
1	The local side asserts lm_req64n to request a 64-bit transaction.
2	The pci_c function outputs reqn to the PCI bus arbiter to request bus ownership. At the same time, the pci_c function asserts lm_tsr[0] to indicate to the local side that the pci_c master is requesting control of the PCI bus.
3	The PCI bus arbiter asserts gntn to grant the PCI bus to the pci_c function. Although Figure 22 shows that the grant occurs immediately and the PCI bus is idle at the time gntn is asserted, this action may not occur immediately in a real transaction. The pci_c function waits for gntn to be asserted while the PCI bus is idle before it proceeds. A PCI bus idle state occurs when both framen and irdyn are deasserted.
5	The pci_c function turns on its output drivers, getting ready to begin the address phase. The pci_c function also outputs lm_adr_ackn to indicate to the local side that it has acknowledged its request. During this same clock cycle, the local side should provide the PCI address on l_adi[31..0] and the PCI command on l_cbeni[3..0]. The local side master interface asserts lm_rdyn to indicate that it is ready to send data to the PCI side. The pci_c function does not assert irdyn regardless if the local side asserts lm_rdyn to indicate that it is ready to send data, only for the first data phase on the local side. For subsequent data phases, the pci_c function asserts irdyn if the local side is ready to send data. The pci_c function continues to assert its reqn signal until the end of the address phase. The pci_c function also asserts lm_tsr[1] to indicate to the local side that the PCI bus has been granted.
6	The pci_c function begins the 64-bit memory read transaction with the address phase by asserting framen and req64n. At the same time, the local side must provide the byte enables for the transaction on l_cbeni[7..0]. The pci_c function asserts lm_ackn regardless if the PCI side is ready to accept data, only for the first data phase on the local side. For subsequent data phases, the pci_c function does not assert lm_ackn unless the PCI side is ready to accept data. Because lm_rdyn was asserted in the previous cycle and lm_ackn is asserted in the current cycle, pci_c asserts lm_dxfrn. The assertion of the lm_dxfrn, l_ldat_ackn, and l_hdat_ackn signals indicate to the local side that the l_adi[63..0] data buses have valid data. The pci_c function asserts lm_tsr[2] to indicate to the local side that the PCI bus is in its address phase.

Table 26. 64-Bit Zero Wait State Master Burst Memory Write Transaction (Part 2 of 3)	
Clock Cycle	**Event**
7	The target claims the transaction by asserting devseln. In this case, the target performs a fast address decode. The target also asserts ack64n to inform the pci_c function that it can transfer 64-bit data. The target also asserts trdyn to inform the pci_c function that it is ready to receive data. During this clock cycle, the pci_c function also asserts lm_tsr[3] to inform the local side that it is in data transfer mode.
8	The pci_c function asserts lm_tsr[9] to indicate to the local side that the target can transfer 64-bit data.
9	The pci_c function asserts irdyn to inform the target that pci_c is ready to send data. Because irdyn and trdyn are asserted, the first 64-bit data is transferred to the PCI side on the rising edge of clock 10. The pci_c function asserts lm_ackn to inform the local side that the PCI side is ready to accept data. Because lm_rdyn was asserted in the previous cycle and lm_ackn is asserted in the current cycle, pci_c asserts lm_dxfrn. The assertion of the lm_dxfrn, l_ldat_ackn, and l_hdat_ackn signals indicates to the local side that the l_adi[63..0] data lines have valid data.
10	Because irdyn and trdyn are asserted, the second 64-bit data is transferred to the PCI side on the rising edge of clock 11. The pci_c function asserts lm_ackn to inform the local side that the PCI side is ready to accept data. Because lm_rdyn was asserted in the previous cycle and lm_ackn is asserted in the current cycle, pci_c asserts lm_dxfrn. The assertion of the lm_dxfrn, l_ldat_ackn, and l_hdat_ackn signals indicate to the local side that the l_adi[63..0] data lines have valid data. Also, the assertion of the lm_lastn signal indicates that this is the last data phase on the local side. The pci_c function also asserts lm_tsr[8] in the same clock to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock.
11	Because lm_lastn was asserted and a data phase was completed in the previous cycle, framen and req64n are deasserted, while irdyn and trdyn are asserted. This action indicates that the last data phase is completed on the PCI side on the rising edge of clock 12. On the local side, the pci_c function deasserts lm_ackn and lm_dxfrn since the last data phase on the local side was completed on the previous cycle. The pci_c function continues to assert lm_tsr[8] informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.

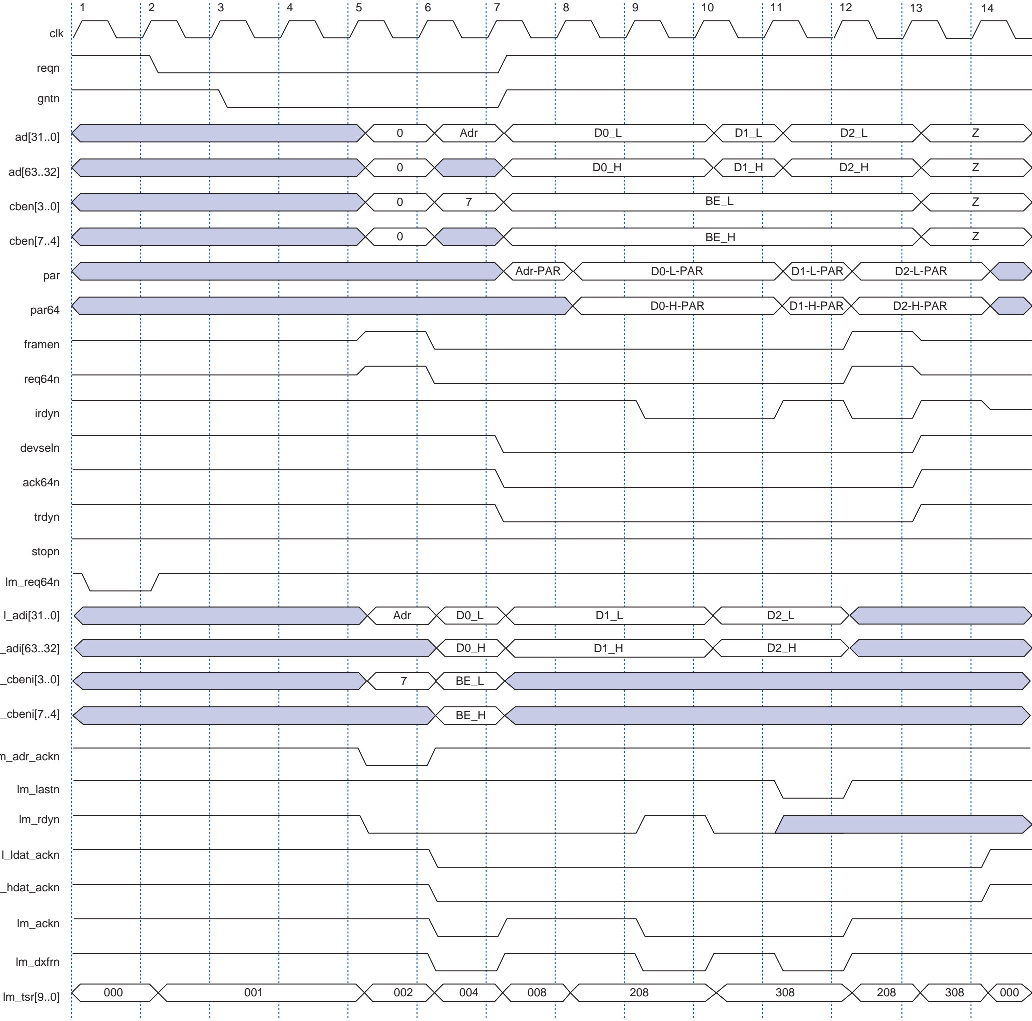
Table 26. 64-Bit Zero Wait State Master Burst Memory Write Transaction (Part 3 of 3)

Clock Cycle	Event
12	On the PCI side, irdyn, devseln, ack64n, and trdyn are deasserted, indicating that the current transaction on the PCI side is completed. There will be no more data phases. The pci_c function continues to assert lm_tsr[8] informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.
13	The pci_c function deasserts lm_tsr[3], informing the local side that the data transfer mode is completed.

64-Bit Master Burst Memory Write Transaction with Local Wait State

Figure 30 shows the same transaction as in Figure 29 with the local side asserting a wait state. The local side deasserts lm_rdyn in clock 9. Consequently, on the following clock cycle (clock 10), the pci_c function suspends data transfer on the local side by deasserting the lm_dxfrn signal. Because there is no data transfer on the local side in clock 10, the pci_c function suspends data transfer on the PCI side by deasserting the irdyn signal in clock 11.

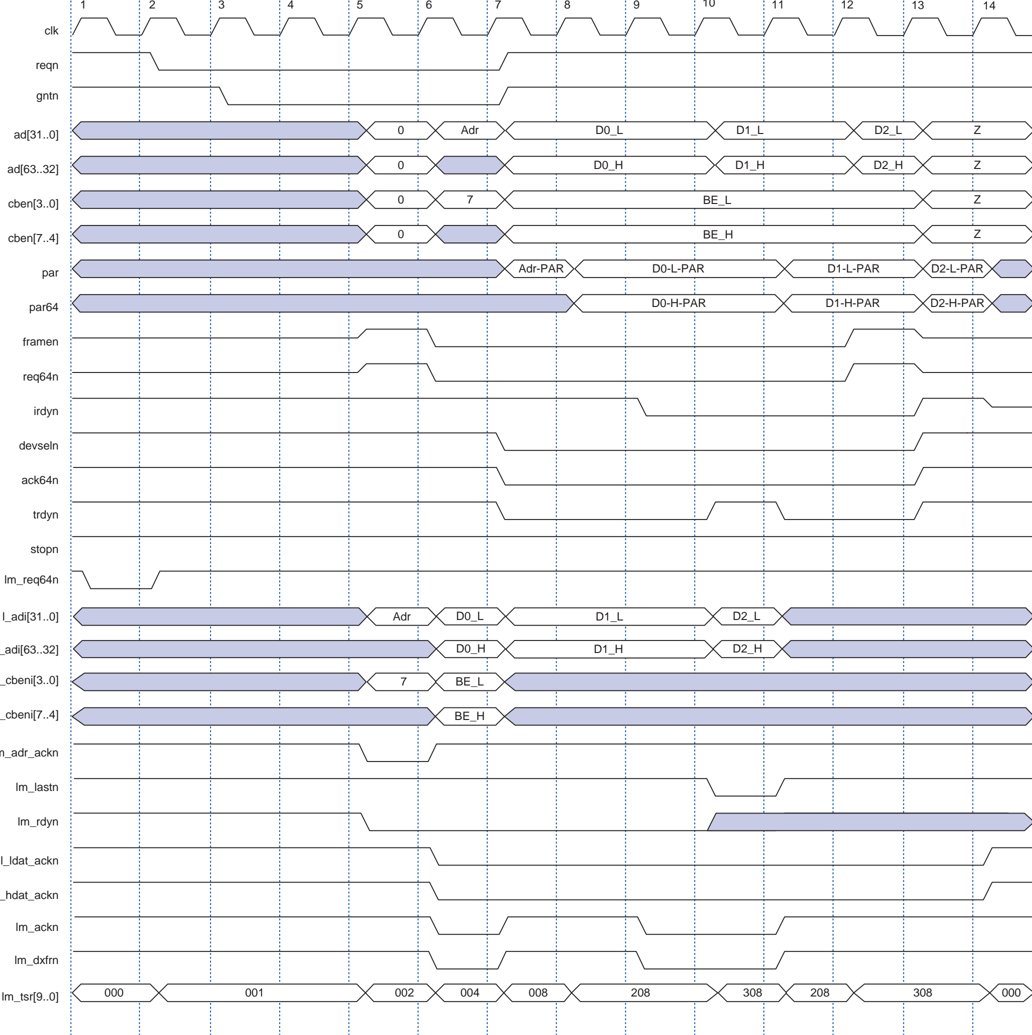
Figure 30. 64-Bit Master Burst Memory Write Transaction with Local Wait State



64-Bit Master Burst Memory Write Transaction with PCI Wait State

Figure 31 shows the same transaction as in Figure 29 with the PCI bus target asserting a wait state. The PCI target asserts a wait state by deasserting `trdyn` in clock 10. Consequently, on the following clock cycle (clock 11), the `pci_c` function deasserts the `lm_ackn` and `lm_dxfrn` signal on the local side. Data transfer is suspended on the PCI side in clock 10 and on the local side in clock 11. Also, because `lm_lastn` is asserted and `lm_rdyn` is deasserted in clock 10, the `lm_ackn` and `lm_dxfrn` signals remain deasserted after clock 11.

Figure 31. 64-Bit Master Burst Memory Write Transaction with PCI Wait State



32-Bit Master Write Transactions

In master mode, the pci_c function supports three types of 32-bit write transactions:

- Memory write transactions
- I/O write transactions
- Configuration read transactions

The 32-bit master write transactions are similar to 64-bit master read transactions, but the upper address ad[63..32] and the upper command/byte enables cben[7..4] are invalid.

32-Bit PCI & 64-Bit Local-Side Master Burst Memory Write Transaction

Figure 32 shows the same transaction as in Figure 29, but the PCI target cannot transfer 64-bit transactions. In this transaction, the local-side master interface requests a 64-bit transaction by asserting lm_req64n. The pci_c function asserts req64n on the PCI side. However, the PCI target cannot transfer 64-bit data, and therefore does not assert ack64n in clock 7. Since this is the case, the upper address ad[63..32] and the upper command/byte enables cben[7..4] are invalid.

Also, because the PCI side is 32 bits wide and the local side is 64 bits wide, the pci_c function assumes that the transactions are within 64-bit boundaries. Therefore, the pci_c function registers l_adi[63..0] on the local side and transfers the lower 32-bit data l_adi[31..0] on the PCI side first, and the upper 32-bit data l_adi[63..32] afterwards.

Figure 32. 32-Bit PCI & 64-Bit Local-Side Master Burst Memory Write Transaction

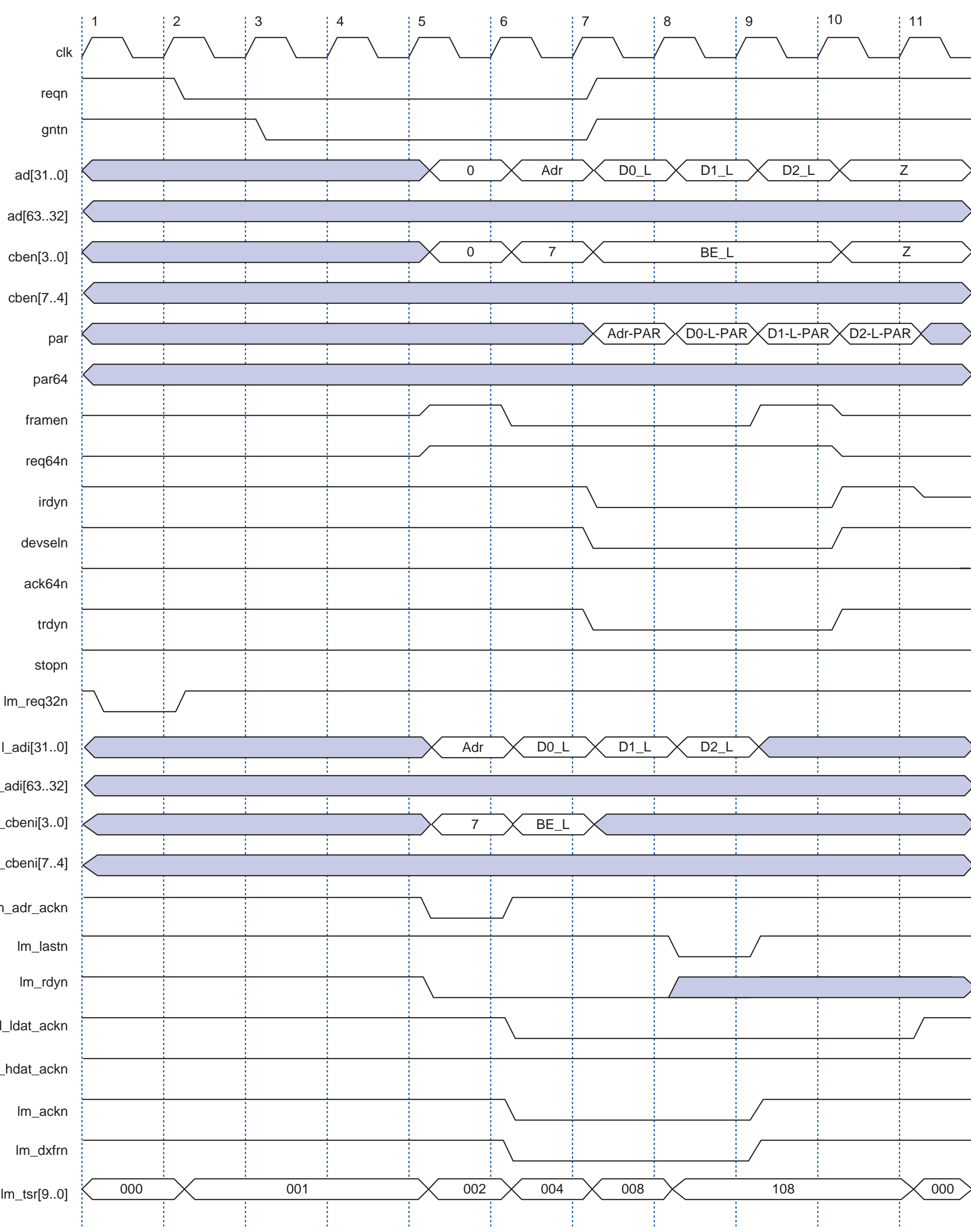
1 2 3 4 5 6 7 8 9 10 11 12 13 14

clk
reqn
gntn
ad[31..0] 0 Adr D0_L D0_H D1_L D1_H Z
ad[63..32]
cben[3..0] 0 7 BE_L Z
cben[7..4]
par Adr-PAR D0-L-PAR D0-H-PAR D1-L-PAR D1-H-PAR
par64
framen
req64n
irdyn
devseln
ack64n
trdyn
stopn
lm_req64n
l_adi[31..0] Adr D0_L D1_L
l_adi[63..32] D0_H D1_H
l_cbeni[3..0] 7 BE_L
l_cbeni[7..4] BE_H
lm_adr_ackn
lm_lastn
lm_rdyn
l_ldat_ackn
l_hdat_ackn
lm_ackn
lm_dxfrn
lm_tsr[9..0] 000 001 002 004 008 108 000

32-Bit PCI & 32-Bit Local-Side Master Burst Memory Write Transaction

Figure 33 shows the same transaction as in Figure 29, but the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. The `pci_c` function does not assert `req64n` on the PCI side. Therefore, the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid.

Figure 33. 32-Bit PCI & 32-Bit Local-Side Master Burst Memory Read Transaction



32-Bit PCI & 32-Bit Local-Side Single-Cycle Memory Write Transaction

Figure 34 shows the same transaction as in Figure 33, but the local side master interface transfers only one data phase. This waveform also applies to the following types of single-cycle transactions:

- I/O write
- Configuration write

Figure 34. 32-Bit PCI & 32-Bit Local-Side Single-Cycle Memory Write Transaction

Abnormal Master Transaction Termination

An abnormal transaction is one in which the local side did not explicitly request the termination of a transaction by asserting the lm_lastn signal. A master transaction can be terminated abnormally for several reasons. This section describes the behavior of the pci_c function during the following abnormal termination conditions:

- Latency timer expires
- Retry
- Disconnect without data
- Disconnect with data
- Target abort
- Master abort

Latency Timer Expires

The PCI specification requires that the master device end the transaction as soon as possible after the latency timer expires and the gntn signal is deasserted. The pci_c function adheres to this rule, and when it ends the transaction because the latency timer expired, it asserts lm_tsr[4] (tsr_lat_exp) until the beginning of the next master transaction.

Retry

The target issues a retry by asserting stopn and devseln during the first data phase. When the pci_c function detects a retry condition (see "Retry" on page 99 for details), it ends the cycle and asserts lm_tsr[5] until the beginning of the next transaction. This process informs the local-side device that it has ended the transaction because the target issued a retry.

The PCI specification requires that the master retry the same transaction with the same address at a later time. It is the responsibility of the local-side application to ensure that this requirement is met.

Disconnect Without Data

The target device issues a disconnect without data if it is unable to transfer additional data during the transaction. The signal pattern for this termination is described in "Disconnect" on page 101. When the pci_c function ends the transaction because of a disconnect without data, it asserts lm_tsr[6] (tsr_disc_wod) until the beginning of the next master transaction.

Disconnect with Data

The target device issues a disconnect with data if it is unable to transfer additional data in the transaction. The signal pattern for this termination is described in "Disconnect" on page 101. When the pci_c function ends the transaction because of a disconnect with data, it asserts lm_tsr[7] (tsr_disc_wd) until the beginning of the next master transaction.

Target Abort

A target device issues this type of termination when a catastrophic failure occurs in the target. The signal pattern for a target abort is shown in "Target Abort" on page 106. When the pci_c function ends the transaction because of a target abort, it asserts the tabort_rcvd signal, which is the same as the PCI status register bit 12. Therefore, the signal remains asserted until it is reset by the host.

Master Abort

The pci_c function terminates the transaction with a master abort when no target claims the transaction by asserting devseln. Except for special cycles and configuration transactions, a master abort is considered to be a catastrophic failure. When a cycle ends in a master abort, the pci_c function informs the local-side device by asserting the mabort_rcvd signal, which is the same as the PCI status register bit 13. Therefore, the signal remains asserted until it is reset by the host.

64-Bit Addressing, Dual Address Cycle (DAC)

This section describes and includes waveform diagrams for 64-bit addressing transactions using dual address cycle (DAC). All 32-bit addressing transactions for master and target mode operation described in the previous sections are supported by 64-bit addressing transactions. This includes both 32-bit and 64-bit data transfers.

Target Mode Operation

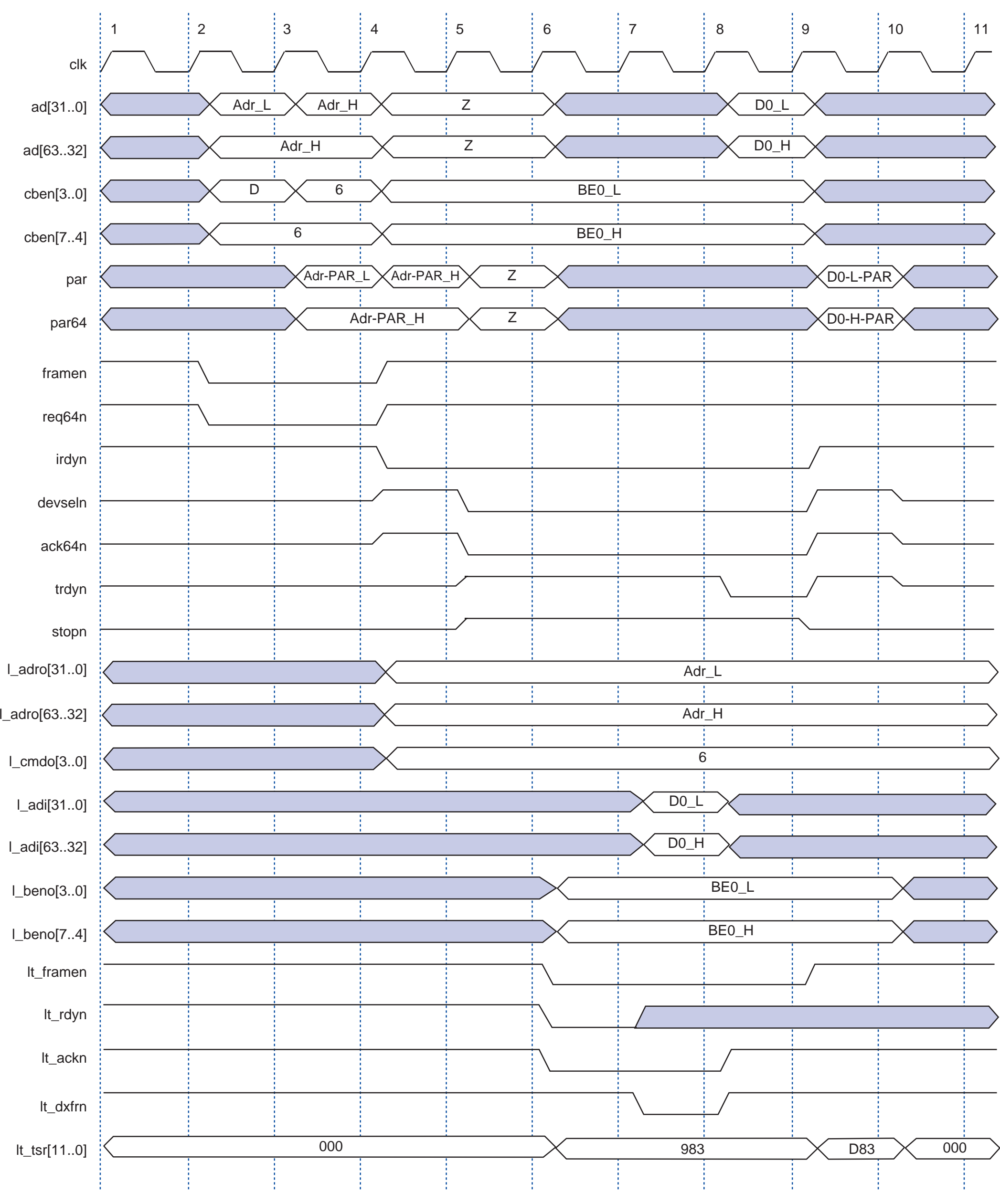
A read or write transaction begins after a master acquires mastership of the PCI bus and asserts `framen` to indicate the beginning of a bus transaction. If the transaction is a 64-bit transaction, the master device asserts the `req64n` signal at the same time it asserts the `framen` signal. The `pci_c` function asserts the `framen` signal in the first clock cycle, which is called the first address phase. During the first address phase, the master device drives the 64-bit transaction address on `ad[63..0]`, the DAC command on `cben[3..0]`, and the transaction command on `cben[7..4]`. On the following clock cycle, during the second address phase, the master device drives the upper 32-bit transaction address on both `ad[63..32]` and `ad[31..0]`, and the transaction command on both `cben[7..4]` and `cben[3..0]`. During these two address phases, the MegaCore function latches the transaction address and command, and decodes the address. If the transaction address matches the `pci_c` target, the `pci_c` target asserts the `devseln` signal to claim the transaction. In 64-bit transactions, `pci_c` also asserts the `ack64n` signal at the same time as the `devseln` signal indicating that it accepts the 64-bit transaction. The `pci_c` function implements slow decode, i.e., the `devseln` and `ack64n` signals are asserted after the second address phase is presented on the PCI bus. Also, both of the `lt_tsr[1..0]` signals are driven high to indicate that the BAR0 and BAR1 address range matches the current transaction address.

64-Bit Address, 64-Bit Data Single-Cycle Target Read Transaction

Figure 35 shows the waveform for a 64-bit address, 64-bit data single-cycle target read transaction. Figure 35 is exactly the same as Figure 1, except Figure 35 has two address phases (described in previous paragraph). Also, both `lt_tsr[1..0]` signals are asserted to indicate that the BAR0 and BAR1 address range of `pci_c` matches the current transaction address. In addition, the current transaction upper 32-bit address is latched on `l_adro[63..32]`, and the lower 32-bit address is latched on `l_adro[31..0]`.

All target transactions described in the Target Mode Operation section for 32-bit addressing are applicable for 64-bit addressing transactions with the differences described in the previous paragraph.

Figure 35. 64-Bit Address, 64-Bit Data Single-Cycle Target Read Transaction



Master Mode Operation

A master operation begins when the local-side master interface asserts the lm_req64n signal to request a 64-bit transaction or the lm_req32n signal to request a 32-bit transaction. The pci_c function outputs the reqn signal to the PCI bus arbiter to request bus ownership. The pci_c function also outputs the lm_adr_ackn signal to the local side to acknowledge the request. When the lm_adr_ackn signal is asserted, the local side provides the PCI address on the l_adi[63..0] bus, the DAC command on l_cbeni[3..0], and the transaction command on the l_cbeni[7..4]. When the PCI bus arbiter grants the bus to the pci_c function by asserting gntn, pci_c begins the transaction with a dual address phase. The pci_c function asserts the framen signal in the first clock cycle, which is called the first address phase. During the first address phase, the pci_c function drives the 64-bit transaction address on ad[63..0], the dual address cycle command on cben[3..0], and the transaction command on cben[7..4]. On the following clock cycle, during the second address phase, the pci_c function drives the upper 32-bit transaction address on both ad[63..32] and ad[31..0], and the transaction command on both cben[7..4] and cben[3..0].

64-Bit Address, 64-Bit Data Master Burst Memory Read Transaction

Figure 36 shows the waveform for a 64-bit address, 64-bit data master burst memory read transaction. Figure 36 is exactly the same as Figure 22, except Figure 36 has two address phases (described in previous paragraph).

All master transactions described in the Master Mode Operation section for 32-bit addressing are applicable for 64-bit addressing transactions with the differences described in the previous paragraph.

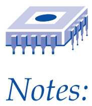
Figure 36. 64-Bit Address, 64-Bit data Master Burst Memory Read Transaction

PCI SIG Protocol Checklists

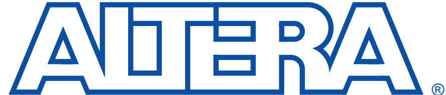
Contents

June 1999

Checklists 149
Component Configuration 149
Component Configuration Space Summary....150
Device Control summary....151
Command Register Summary....151
Device Status....152
Status Register Summary....152
Component Master Checklist....153
Component Target Checklist....155
PCI SIG Test Scenarios 156
Test Scenario: 1.1. PCI Device Speed (As Indicated by devsel) Tests 157
Test Scenario: 1.2. PCI Bus Target Abort Cycles 158
Test Scenario: 1.3. PCI Bus Target Retry Cycles 160
Test Scenario: 1.4. PCI Bus Single Data Phase Disconnect Cycles 161
Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles....162
Test Scenario: 1.6. PCI Bus Multi-Data Phase Retry Cycles....164
Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles....165
Test Scenario: 1.8. Multi-Data Phase & trdyn Cycles....167
Test Scenario: 1.9. Bus Data Parity Error Single Cycles....169
Test Scenario: 1.10. Bus Data Parity Error Multi-Data Phase Cycles....170
Test Scenario: 1.11. Bus Master Timeout 171
Test Scenario: 1.13. PCI Bus Master Parking....171
Test Scenario: 1.14. PCI Bus Master Arbitration....171
Test Scenario: 2.1. Target Reception of an Interrupt Cycle 172
Test Scenario: 2.2. Target Reception of Special Cycle 172
Test Scenario: 2.3. Target Detection of Address & Data Parity Error for Special Cycle 172
Test Scenario: 2.4. Target Reception of I/O Cycles with Legal & Illegal Byte Enables 172
Test Scenario: 2.5. Target Ignores Reserved Commands 173
Test Scenario: 2.6. Target Receives Configuration Cycles....173
Test Scenario: 2.7. Target Receives I/O Cycles with Address & Data Parity Errors 173
Test Scenario: 2.8 Target Receives Configuration Cycles with Address & Data Parity Errors....173
Test Scenario: 2.9. Target Receives Memory Cycles 174
Test Scenario: 2.10. Target Receives Memory Cycles with Address & Parity Errors....174
Test Scenario: 2.11. Target Receives Fast Back-to-Back Cycles 174
Test Scenario: 2.12. Target Performs Exclusive Address Cycles....174
Test Scenario: 2.13. Target Receives Cycles with irdy Used for Data Stepping....175



Notes:



PCI SIG Protocol Checklists

June 1999, ver. 1.1

Checklists

Tables 1 through 8 list the applicable PCI SIG protocol requirements from the *PCI Compliance Checklist, Revision 2.1*. A check mark in the Yes column indicates that the `pci_c` function meets the requirement. Checklists not applicable to the Altera `pci_c` function are not listed, and table entries annotated with N.A. represent non-applicable PCI SIG requirements.

Table 1. Component Configuration

CO#	Requirement	pci_c	
		Yes	No
1	Does each PCI resource have a configuration space based on the 256 byte template defined in section 6.1, with a predefined 64-byte header and a 192-byte device-specific region?	✓	
2	Do all functions in the device support the vendor ID, device ID, command, status, header type, and class code fields in the header?	✓	
3	Is the configuration space available for access at all times?	✓	
4	Are writes to reserved registers or read-only bits completed normally and the data discarded?	✓	
5	Are reads to reserved or unimplemented registers, or bits, completed normally and a data value of 0 returned?	✓	
6	Is the vendor ID a number allocated by the PCI SIG?	✓	
7	Does the header type field have a valid encoding?	✓	
8	Do multi-byte transactions access the appropriate registers and are the registers in “little endian” order?	✓	
9	Are all read-only register values within legal ranges? For example, the interrupt pin register must only contain values 0 - 4.	✓	
10	Is the class code in compliance with the definition in appendix D?	✓	
11	Is the predefined header portion of configuration space accessible as bytes, words, and DWORDs?	✓	
12	Is the device a multi-function device?		✓
13	If the device is multi-function, are configuration space accesses to unimplemented functions ignored?		N.A

Table 2. Component Configuration Space Summary

Location	Name	Required/Optional	pci_c	
			N/A	Yes
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, bit 7 must be set to a 1.		✓
0Fh	BIST	Optional.	✓	
10h-27h	BAR0	Optional.		✓
28h-2Bh	BAR1-BAR5	Optional.		✓
2Ch-2Dh	Cardbus CIS pointer	Optional.	✓	
2Eh-2Fh	Subsystem vendor ID	Optional.		✓
30h-33h	Subsystem ID	Optional.		✓
34h-3Bh	Expansion ROM base address	Required for devices/functions that have expansion ROM.		✓
3Ch	Reserved			✓
3Dh	Interrupt line	Required by devices/functions that use an interrupt pin.		✓
3Eh	Interrupt pin	Required by devices/functions that use an interrupt pin.		✓
3Fh	Min_Gnt	Optional.		✓

Table 3. Device Control summary

DC#	Required/Optional	pci_c	
		Yes	No
1	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).	✓	
2	Is the device/function disabled after the assertion of PCI `rstn`? (Devices in boot code are exempt.)	✓	

Table 4. Command Register Summary

Bit	Name	Required/Optional	pci_c	
			Yes	No
0	I/O space	Required if device/function has registers mapped into I/O space.	✓	
1	Memory space	Required if device/function responds to memory space accesses.	✓	
2	Bus master	Required.	✓	
3	Special cycles	Required for devices/functions that can respond to special cycles.		✓
4	Memory write and invalidate	Required for devices/functions that generate Memory Write and Invalidate cycles.	✓	
5	VGA palettesnoop	Required for VGA or graphical devices/functions that snoop VGA palette.		✓
6	Parity error response	Required.	✓	
7	Wait cycle control	Optional.	✓	
8	`serrn` enable	Required if device/function has `serrn` pin.	✓	
9	Fast back-to-back enable	Required if master device/function can support fast back-to-back cycles among different targets.		✓
10-15	Reserved			

Table 5. Device Status

DS#	Requirement	pci_c	
		Yes	No
1	Do all implemented read/write bits in the status reset to 0?	✓	
2	Are read/write bits set to a 1 exclusively by the device/function?	✓	
3	Are read/write bits reset to a 0 when PCI `rstn` is asserted?	✓	
4	Are read/write bits reset to a 0 by writing a 1 to the bit?	✓	

Table 6. Status Register Summary

Bit	Name	Required/Optional	pci_c	
			Yes	No
0-4	Reserved	Required.		
5	66-MHz capable	Required for 66-MHz capable devices.	✓	
6	UDF supported	Optional.		✓
7	Fast back-to-back capable	Optional.		✓
8	Data parity detected	Required.	✓	
9-10	`devsel` timing	Required.	✓	
11	Signaled target abort	Required for devices/functions that are capable of signaling target abort.	✓	
12	Received target abort	Required.	✓	
13	Received master abort	Required.	✓	
14	Signaled system error	Required for devices/functions that are capable of asserting `serrn`.	✓	
15	Detected parity error	Required unless exempted per section 3.7.2.	✓	

Table 7. Component Master Checklist (Part 1 of 2)

MP#	Requirement	pci_c	
		Yes	No
1	All sustained tri-state signals are driven high for one clock before being tri-stated. (section 2.1)	✓	
2	Interface under test (IUT) always asserts all byte enables during each data phase of a memory write and invalidate cycle. (section 3.1.1)	N.A.	
3	IUT always uses linear burst ordering for memory write and invalidate cycles. (section 3.1.1)	✓	
4	IUT always drives `irdyn` when data is valid during a write transaction. (section 3.2.1)	✓	
5	IUT only transfers data when both `irdyn` and `trdyn` are asserted on the same rising clock edge. (section 3.2.1)	✓	
6	Once the IUT asserts `irdyn`, it does not change `framen` until the current data phase completes. (section 3.2.1)	✓	
7	Once the IUT asserts `irdyn`, it does not change `irdyn` until the current data phase completes. (section 3.2.1)	✓	
8	IUT never uses reserved burst ordering (`ad[1..0]` = “01”). (section 3.2.2)	✓	
9	IUT never uses reserved burst ordering (`ad[1..0]` = “11”). (section 3.2.2)	✓	
10	IUT always ignores the configuration command unless `idsel` is asserted and `ad[1..0]` is “00”. (section 3.2.2)	✓	
11	The IUT’s address lines are driven to stable values during every address and data phase. (section 3.2.4)	✓	
12	The IUT’s `cben[3..0]` output buffers remain enabled from the first clock of the data phase through the end of the transaction. (section 3.3.1)	✓	
13	The IUT’s `cben[3..0]` lines contain valid byte enable information during the entire data phase. (section 3.3.1)	✓	
14	IUT never deasserts `framen` unless `irdyn` is asserted or will be asserted. (section 3.3.3.1)	✓	
15	IUT never deasserts `irdyn` until at least one clock after `framen` is deasserted. (section 3.3.3.1)	✓	
16	Once the IUT deasserts `framen`, it never reasserts `framen` during the same transaction. (section 3.3.3.1)	✓	
17	IUT never terminates with master abort once target has asserted `devseln`.	✓	
18	IUT never signals master abort earlier than 5 clocks after `framen` was first sample-asserted. (section 3.3.3.1)	✓	
19	IUT always repeats an access exactly as the original when terminated by retry. (section 3.3.3.2.2)	✓	
20	IUT never starts cycle unless `gntn` is asserted. (section 3.4.1)	✓	
21	IUT always tri-states `cben[3..0]` and `ad[31..0]` within one clock after `gntn` negation when the bus is idle and `framen` is negated. (section 3.4.3)	✓	

Table 7. Component Master Checklist (Part 2 of 2)

MP#	Requirement	pci_c	
		Yes	No
22	IUT always drives `cben[3..0]` and `ad[31..0]` within eight clocks of `gntn` assertion when the bus is idle. (section 3.4.3)	✓	
23	IUT always asserts `irdyn` within eight clocks on all data phases. (section 3.5.2)	✓	
24	IUT always begins lock operation with a read transaction. (section 3.6)	N.A.	
25	IUT always releases LOCK# when access is terminated by target-abort or master-abort. (section 3.6)	N.A.	
26	IUT always deasserts LOCK# for a minimum of one idle cycle between consecutive lock operations. (section 3.6)	N.A.	
27	IUT always uses linear burst ordering for configuration cycles. (section 3.7.4)	✓	
28	IUT always drives `par` within one clock of `cben[3..0]` and `ad[31..0]` being driven. (section 3.8.1)	✓	
29	IUT always drives `par` such that the number of “1”s on `ad[31..0]`, `cben[3..0]`, and `par` equals an even number. (section 3.8.1)	✓	
30	IUT always drives `perrn` (when enabled) active two clocks after data when a data parity error is detected. (section 3.8.2.1)	✓	
31	IUT always drives `perr` (when enabled) for a minimum of 1 clock for each data phase that a parity error is detected. (section 3.8.2.1)	✓	
32	IUT always holds `framen` asserted for the cycle following DUAL command. (section 3.10.1)	N.A.	
33	IUT never generates a dual cycle when the upper 32-bits of the address are zero. (section 3.10.1)	N.A.	

Table 8. Component Target Checklist (Part 1 of 2)

TP#	Requirement	pci_c	
		Yes	No
1	All sustained tri-state signals are driven high for one clock before being tri-stated. (section 2.1)	✓	
2	IUT never reports `perrn` until it has claimed the cycle and completed a data phase. (section 2.2.5)	✓	
3	IUT never aliases reserved commands with other commands. (section 3.1.1)	N.A.	
4	32-bit addressable IUT treats the dual command as reserved. (section 3.1.1)	N.A.	
5	Once IUT has asserted `trdyn`, it does not change `trdyn` until the data phase completes. (section 3.2.1)	✓	
6	Once IUT has asserted `trdyn`, it does not change `devseln` until the data phase completes. (section 3.2.1)	✓	
7	Once IUT has asserted `trdyn`, it does not change `stopn` until the data phase completes. (section 3.2.1)	✓	
8	Once IUT has asserted `stopn`, it does not change `stopn` until the data phase completes. (section 3.2.1)	✓	
9	Once IUT has asserted `stopn`, it does not change `trdyn` until the data phase completes. (section 3.2.1)	✓	
10	Once IUT has asserted `stopn`, it does not change `devseln` until the data phase completes. (section 3.2.1)	✓	
11	IUT only transfers data when both `irdyn` and `trdyn` are asserted on the same rising clock edge. (section 3.2.1)	✓	
12	IUT always asserts `trdyn` when data is valid on a read cycle. (section 3.2.1)	✓	
13	IUT always signals target-abort when unable to complete the entire I/O access as defined by the byte enables. (section 3.2.2)	N.A.	
14	IUT never responds to reserved encodings. (section 3.2.2)	✓	
15	IUT always ignores a configuration command unless `idsel` is asserted and `ad[31..0]` is "00". (section 3.2.2)	✓	
16	IUT always disconnects after the first data phase when reserved burst mode is detected. (section 3.2.2)	N.A.	
17	The IUT's `ad[31..0]` lines are driven to stable values during every address and data phase. (section 3.2.4)	✓	
18	The IUT's `cben[3..0]` output buffers remain enabled from the first clock of the data phase through the end of the transaction. (section 3.3.1)	✓	
19	IUT never asserts `trdyn` during a turn-around cycle on a read. (section 3.3.1)	✓	
20	IUT always deasserts `trdyn`, `stopn`, and `devseln` the clock following the completion of the last data phase. (section 3.3.3.2)	✓	
21	IUT always signals disconnect when a burst crosses the resource boundary. (section 3.3.3.2)	N.A.	

Table 8. Component Target Checklist (Part 2 of 2)

TP#	Requirement	pci_c	
		Yes	No
22	IUT always deasserts `stopn` the cycle immediately following `framen` being deasserted. (section 3.3.3.2.1)	✓	
23	Once the IUT has asserted `stopn`, it does not deassert `stopn` until `framen` is negated. (section 3.3.3.2.1)	✓	
24	IUT always deasserts `trdyn` before signaling target-abort. (section 3.3.3.2.1)	N.A.	
25	IUT never deasserts `stopn` and continues the transaction. (section 3.3.3.2.1)	✓	
26	IUT always completes an initial data phase within 16 clocks. (section 3.5.1.1)	✓	
27	IUT always locks a minimum of 16 bytes. (section 3.6)	N.A.	
28	IUT always issues `devseln` before any other response. (section 3.7.1)	✓	
29	Once IUT has asserted `devseln`, it does not deassert `devseln` until the last data phase has competed except to signal target-abort. (section 3.7.1)	✓	
30	IUT never responds to special cycles. (section 3.7.2)	✓	
31	IUT always drives `par` within one clock of `cben[3..0]` and `ad[31..0]` being driven. (section 3.8.1)	✓	
32	IUT always drives `par` such that the number of “1”s on `ad[31..0]`, `cben[3..0]`, and `par` equals an even number. (section 3.8.1)	✓	

PCI SIG Test Scenarios

Tables 9 through 24 list the applicable PCI SIG test scenarios from the *Compliance Checklist, Revision 2.1*. A check mark in the Yes column indicates that the `pci_c` function meets the requirement. Checklist items that are not applicable are indicated with N.A.

Table 9. Test Scenario: 1.1. PCI Device Speed (As Indicated by devsel) Tests

#	Requirement	pci_c	
		Yes	No
1	Data transfer after write to fast memory target.	✓	
2	Data transfer after read from fast memory target.	✓	
3	Data transfer after write to medium memory target.	✓	
4	Data transfer after read from medium memory target.	✓	
5	Data transfer after write to slow memory target.	✓	
6	Data transfer after read from slow memory target.	✓	
7	Data transfer after write to subtractive memory target.	✓	
8	Data transfer after read from subtractive memory target.	✓	
9	Master abort bit set after write to slower than subtractive memory target.	✓	
10	Master abort bit set after read from slower than subtractive memory target.	✓	
11	Data transfer after write to fast I/O target.	✓	
12	Data transfer after read from fast I/O target.	✓	
13	Data transfer after write to medium I/O target.	✓	
14	Data transfer after read from medium I/O target.	✓	
15	Data transfer after write to slow I/O target.	✓	
16	Data transfer after read from slow I/O target.	✓	
17	Data transfer after write to subtractive I/O target.	✓	
18	Data transfer after read from subtractive I/O target.	✓	
19	Master abort bit set after write to slower than subtractive I/O target.	✓	
20	Master abort bit set after read from slower than subtractive I/O target.	✓	
21	Data transfer after write to fast configuration target.	✓	
22	Data transfer after read from fast configuration target.	✓	
23	Data transfer after write to medium configuration target.	✓	
24	Data transfer after read from medium configuration target.	✓	
25	Data transfer after write to slow configuration target.	✓	
26	Data transfer after read from slow configuration target.	✓	
27	Data transfer after write to subtractive configuration target.	✓	
28	Data transfer after read from subtractive configuration target.	✓	
29	Master abort bit set after write to slower than subtractive configuration target.	✓	
30	Master abort bit set after read from slower than subtractive configuration target.	✓	

Table 10. Test Scenario: 1.2. PCI Bus Target Abort Cycles (Part 1 of 2)			
#	Requirement	pci_c	
		Yes	No
1	Target abort bit set after write to fast memory target.	✓	
2	IUT does not repeat the write transaction.	✓	
3	IUT’s target abort bit set after read from fast memory target.	✓	
4	IUT does not repeat the read transaction.	✓	
5	Target abort bit set after write to medium memory target.	✓	
6	IUT does not repeat the write transaction.	✓	
7	IUT’s target abort bit set after read from medium memory target.	✓	
8	IUT does not repeat the read transaction.	✓	
9	Target abort bit set after write to slow memory target.	✓	
10	IUT does not repeat the write transaction.	✓	
11	IUT’s target abort bit set after read from slow memory target.	✓	
12	IUT does not repeat the read transaction.	✓	
13	Target abort bit set after write to subtractive memory target.	✓	
14	IUT does not repeat the write transaction.	✓	
15	IUT’s target abort bit set after read from subtractive memory target.	✓	
16	IUT does not repeat the read transaction.	✓	
17	Target abort bit set after write to fast I/O target.	✓	
18	IUT does not repeat the write transaction.	✓	
19	IUT’s target abort bit set after read from fast I/O target.	✓	
20	IUT does not repeat the read transaction.	✓	
21	Target abort bit set after write to medium I/O target.	✓	
22	IUT does not repeat the write transaction.	✓	
23	IUT’s target abort bit set after read from medium I/O target.	✓	
24	IUT does not repeat the read transaction.	✓	
25	Target abort bit set after write to slow I/O target.	✓	
26	IUT does not repeat the write transaction.	✓	
27	IUT’s target abort bit set after read from slow I/O target.	✓	
28	IUT does not repeat the read transaction.	✓	
29	Target abort bit set after write to subtractive I/O target.	✓	
30	IUT does not repeat the write transaction.	✓	
31	IUT’s target abort bit set after read from subtractive I/O target.	✓	
32	IUT does not repeat the read transaction.	✓	
33	Target abort bit set after write to fast configuration target.	✓	

Table 10. Test Scenario: 1.2. PCI Bus Target Abort Cycles (Part 2 of 2)

#	Requirement	pci_c	
		Yes	No
34	IUT does not repeat the write transaction.	✓	
35	IUT’s target abort bit set after read from fast configuration target.	✓	
36	IUT does not repeat the read transaction.	✓	
37	Target abort bit set after write to medium configuration target.	✓	
38	IUT does not repeat the write transaction.	✓	
39	IUT’s target abort bit set after read from medium configuration target.	✓	
40	IUT does not repeat the read transaction.	✓	
41	Target abort bit set after write to slow configuration target.	✓	
42	IUT does not repeat the write transaction.	✓	
43	IUT’s target abort bit set after read from slow configuration target.	✓	
44	IUT does not repeat the read transaction.	✓	
45	Target abort bit set after write to subtractive configuration target.	✓	
46	IUT does not repeat the write transaction.	✓	
47	IUT’s target abort bit set after read from subtractive configuration target.	✓	
48	IUT does not repeat the read transaction.	✓	

Table 11. Test Scenario: 1.3. PCI Bus Target Retry Cycles

#	Requirement	pci_c	
		Yes	No
1	Data transfer after write to fast memory target.	✓	
2	Data transfer after read from fast memory target.	✓	
3	Data transfer after write to medium memory target.	✓	
4	Data transfer after read from medium memory target.	✓	
5	Data transfer after write to slow memory target.	✓	
6	Data transfer after read from slow memory target.	✓	
7	Data transfer after write to subtractive memory target.	✓	
8	Data transfer after read from subtractive memory target.	✓	
9	Data transfer after write to fast I/O target.	✓	
10	Data transfer after read from fast I/O target.	✓	
11	Data transfer after write to medium I/O target.	✓	
12	Data transfer after read from medium I/O target.	✓	
13	Data transfer after write to slow I/O target.	✓	
14	Data transfer after read from slow I/O target.	✓	
15	Data transfer after write to subtractive I/O target.	✓	
16	Data transfer after read from subtractive I/O target.	✓	
17	Data transfer after write to fast configuration target.	✓	
18	Data transfer after read from fast configuration target.	✓	
19	Data transfer after write to medium configuration target.	✓	
20	Data transfer after read from medium configuration target.	✓	
21	Data transfer after write to slow configuration target.	✓	
22	Data transfer after read from slow configuration target.	✓	
23	Data transfer after write to subtractive configuration target.	✓	
24	Data transfer after read from subtractive configuration target.	✓	

Table 12. Test Scenario: 1.4. PCI Bus Single Data Phase Disconnect Cycles			
#	Requirement	pci_c	
		Yes	No
1	Data transfer after write to fast memory target.	✓	
2	Data transfer after read from fast memory target.	✓	
3	Data transfer after write to medium memory target.	✓	
4	Data transfer after read from medium memory target.	✓	
5	Data transfer after write to slow memory target.	✓	
6	Data transfer after read from slow memory target.	✓	
7	Data transfer after write to subtractive memory target.	✓	
8	Data transfer after read from subtractive memory target.	✓	
9	Data transfer after write to fast I/O target.	✓	
10	Data transfer after read from fast I/O target.	✓	
11	Data transfer after write to medium I/O target.	✓	
12	Data transfer after read from medium I/O target.	✓	
13	Data transfer after write to slow I/O target.	✓	
14	Data transfer after read from slow I/O target.	✓	
15	Data transfer after write to subtractive I/O target.	✓	
16	Data transfer after read from subtractive I/O target.	✓	
17	Data transfer after write to fast configuration target.	✓	
18	Data transfer after read from fast configuration target.	✓	
19	Data transfer after write to medium configuration target.	✓	
20	Data transfer after read from medium configuration target.	✓	
21	Data transfer after write to slow configuration target.	✓	
22	Data transfer after read from slow configuration target.	✓	
23	Data transfer after write to subtractive configuration target.	✓	
24	Data transfer after read from subtractive configuration target.	✓	

Table 13. Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles (Part 1 of 3)

#	Requirement	pci_c	
		Yes	No
1	Target abort bit set after write to fast memory target.	✓	
2	IUT does not repeat the write transaction.	✓	
3	IUT’s target abort bit set after read from fast memory target.	✓	
4	IUT does not repeat the read transaction.	✓	
5	Target abort bit set after write to medium memory target.	✓	
6	IUT does not repeat the write transaction.	✓	
7	IUT’s target abort bit set after read from medium memory target.	✓	
8	IUT does not repeat the read transaction.	✓	
9	Target abort bit set after write to slow memory target.	✓	
10	IUT does not repeat the write transaction.	✓	
11	IUT’s target abort bit set after read from slow memory target.	✓	
12	IUT does not repeat the read transaction.	✓	
13	Target abort bit set after write to subtractive memory target.	✓	
14	IUT does not repeat the write transaction.	✓	
15	IUT’s target abort bit set after read from subtractive memory target.	✓	
16	IUT does not repeat the read transaction.	✓	
17	Target abort bit set after write to fast memory target.	✓	
18	IUT does not repeat the write transaction.	✓	
19	IUT’s target abort bit set after read from fast memory target.	✓	
20	IUT does not repeat the read transaction.	✓	
21	Target abort bit set after write to medium memory target.	✓	
22	IUT does not repeat the write transaction.	✓	
23	IUT’s target abort bit set after read from medium memory target.	✓	
24	IUT does not repeat the read transaction.	✓	
25	Target abort bit set after write to slow memory target.	✓	
26	IUT does not repeat the write transaction.	✓	
27	IUT’s target abort bit set after read from slow memory target.	✓	
28	IUT does not repeat the read transaction.	✓	
29	Target abort bit set after write to subtractive memory target.	✓	
30	IUT does not repeat the write transaction.	✓	
31	IUT’s target abort bit set after read from subtractive memory target.	✓	
32	IUT does not repeat the read transaction.	✓	
33	Target abort bit set after write to fast configuration target.	N.A.	

Table 13. Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles (Part 2 of 3)

#	Requirement	pci_c	
		Yes	No
34	IUT does not repeat the write transaction.	N.A.	
35	IUT’s target abort bit set after read from fast configuration target.	N.A.	
36	IUT does not repeat the read transaction.	N.A.	
37	Target abort bit set after write to medium configuration target.	N.A.	
38	IUT does not repeat the write transaction.	N.A.	
39	IUT’s target abort bit set after read from medium configuration target.	N.A.	
40	IUT does not repeat the read transaction.	N.A.	
41	Target abort bit set after write to slow configuration target.	N.A.	
42	IUT does not repeat the write transaction.	N.A.	
43	IUT’s target abort bit set after read from slow configuration target.	N.A.	
44	IUT does not repeat the read transaction.	N.A.	
45	Target abort bit set after write to subtractive configuration target.	N.A.	
46	IUT does not repeat the write transaction.	N.A.	
47	IUT’s target abort bit set after read from subtractive configuration target.	N.A.	
48	IUT does not repeat the read transaction.	N.A.	
49	IUT’s target abort bit set after read from fast memory target.	✓	
50	IUT does not repeat the read transaction.	✓	
51	IUT’s target abort bit set after read from medium memory target.	✓	
52	IUT does not repeat the read transaction.	✓	
53	IUT’s target abort bit set after read from slow memory target.	✓	
54	IUT does not repeat the read transaction.	✓	
55	IUT’s target abort bit set after read from subtractive memory target.	✓	
56	IUT does not repeat the read transaction.	✓	
57	IUT’s target abort bit set after read from fast memory target.	✓	
58	IUT does not repeat the read transaction.	✓	
59	IUT’s target abort bit set after read from medium memory target.	✓	
60	IUT does not repeat the read transaction.	✓	
61	IUT’s target abort bit set after read from slow memory target.	✓	
62	IUT does not repeat the read transaction.	✓	
63	IUT’s target abort bit set after read from subtractive memory target.	✓	
64	IUT does not repeat the read transaction.	✓	
65	Target abort bit set after write to fast memory target.	✓	
66	IUT does not repeat the write transaction.	✓	

Table 13. Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles (Part 3 of 3)

#	Requirement	pci_c	
		Yes	No
67	Target abort bit set after write to medium memory target.	✓	
68	IUT does not repeat the write transaction.	✓	
69	Target abort bit set after write to slow memory target.	✓	
70	IUT does not repeat the write transaction.	✓	
71	IUT’s target abort bit set after read from slow memory target.	✓	
72	IUT does not repeat the write transaction.	✓	

Table 14. Test Scenario: 1.6. PCI Bus Multi-Data Phase Retry Cycles (Part 1 of 2)

#	Requirement	pci_c	
		Yes	No
1	Data transfer after write to fast memory target.	✓	
2	Data transfer after read from fast memory target.	✓	
3	Data transfer after write to medium memory target.	✓	
4	Data transfer after read from medium memory target.	✓	
5	Data transfer after write to slow memory target.	✓	
6	Data transfer after read from slow memory target.	✓	
7	Data transfer after write to subtractive memory target.	✓	
8	Data transfer after read from subtractive memory target.	✓	
9	Data transfer after write to fast I/O target.	✓	
10	Data transfer after read from fast I/O target.	✓	
11	Data transfer after write to medium I/O target.	✓	
12	Data transfer after read from medium I/O target.	✓	
13	Data transfer after write to slow I/O target.	✓	
14	Data transfer after read from slow I/O target.	✓	
15	Data transfer after write to subtractive I/O target.	✓	
16	Data transfer after read from subtractive I/O target.	✓.	
17	Data transfer after write to fast configuration target.	N.A.	
18	Data transfer after read from fast configuration target.	N.A.	
19	Data transfer after write to medium configuration target.	N.A.	
20	Data transfer after read from medium configuration target.	N.A.	
21	Data transfer after write to slow configuration target.	N.A.	
22	Data transfer after read from slow configuration target.	N.A.	

Table 14. Test Scenario: 1.6. PCI Bus Multi-Data Phase Retry Cycles (Part 2 of 2)			
#	Requirement	pci_c	
		Yes	No
23	Data transfer after write to subtractive configuration target.	N.A.	
24	Data transfer after read from subtractive configuration target.	N.A.	
25	Data transfer after memory read multiple from fast target.	✓	
26	Data transfer after memory read multiple from medium target.	✓	
27	Data transfer after memory read multiple from slow target.	✓	
28	Data transfer after memory read multiple from subtractive target.	✓	
29	Data transfer after memory read line from fast target.	✓	
30	Data transfer after memory read line from medium target.	✓	
31	Data transfer after memory read line from slow target.	✓	
32	Data transfer after memory read line from subtractive target.	✓	
33	Data transfer after memory write and invalidate to fast target.	✓	
34	Data transfer after memory write and invalidate to medium target.	✓	
35	Data transfer after memory write and invalidate to slow target.	✓	
36	Data transfer after memory write and invalidate to subtractive target.	✓	

Table 15. Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles (Part 1 of 2)			
#	Requirement	pci_c	
		Yes	No
1	Data transfer after write to fast memory target.	✓	
2	Data transfer after read from fast memory target.	✓	
3	Data transfer after write to medium memory target.	✓	
4	Data transfer after read from medium memory target.	✓	
5	Data transfer after write to slow memory target.	✓	
6	Data transfer after read from slow memory target.	✓	
7	Data transfer after write to subtractive memory target.	✓	
8	Data transfer after read from subtractive memory target.	✓	
9	Data transfer after write to fast I/O target.	✓	
10	Data transfer after read from fast I/O target.	✓	
11	Data transfer after write to medium I/O target.	✓	
12	Data transfer after read from medium I/O target.	✓	
13	Data transfer after write to slow I/O target.	✓	

Table 15. Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles (Part 2 of 2)

#	Requirement	pci_c	
		Yes	No
14	Data transfer after read from slow I/O target.	✓	
15	Data transfer after write to subtractive I/O target.	✓	
16	Data transfer after read from subtractive I/O target.	✓	
17	Data transfer after write to fast configuration target.	N.A.	
18	Data transfer after read from fast configuration target.	N.A.	
19	Data transfer after write to medium configuration target.	N.A.	
20	Data transfer after read from medium configuration target.	N.A.	
21	Data transfer after write to slow configuration target.	N.A.	
22	Data transfer after read from slow configuration target.	N.A.	
23	Data transfer after write to subtractive configuration target.	N.A.	
24	Data transfer after read from subtractive configuration target.	N.A.	
25	Data transfer after memory read multiple from fast target.	✓	
26	Data transfer after memory read multiple from medium target.	✓	
27	Data transfer after memory read multiple from slow target.	✓	
28	Data transfer after memory read multiple from subtractive target.	✓	
29	Data transfer after memory read line from fast target.	✓	
30	Data transfer after memory read line from medium target.	✓	
31	Data transfer after memory read line from slow target.	✓	
32	Data transfer after memory read line from subtractive target.	✓	
33	Data transfer after memory write and invalidate to fast target.	✓	
34	Data transfer after memory write and invalidate to medium target.	✓	
35	Data transfer after memory write and invalidate to slow target.	✓	
36	Data transfer after memory write and invalidate to subtractive target.	✓	

Table 16. Test Scenario: 1.8. Multi-Data Phase & trdyn Cycles (Part 1 of 3)

#	Requirement	pci_c	
		Yes	No
1	Verify that data is written to the primary target when `trdyn` is released after the second rising clock edge and asserted on the third rising clock edge after `framen`.	✓	
2	Verify that data is read from the primary target when `trdyn` is released after the second rising clock edge and asserted on the third rising clock edge after `framen`.	✓	
3	Verify that data is written to the primary target when `trdyn` is released after the third rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
4	Verify that data is read from the primary target when `trdyn` is released after the third rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
5	Verify that data is written to the primary target when `trdyn` is released after the third rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
6	Verify that data is read from the primary target when `trdyn` is released after the third rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
7	Verify that data is written to the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	
8	Verify that data is read from the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	
9	Verify that data is written to the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
10	Verify that data is read from the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
11	Verify that data is written to the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	
12	Verify that data is read from the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	
13	Verify that data is written to the primary target when `trdyn` is released after the third rising clock edge and asserted on the third rising clock edge after `framen`.	✓	
14	Verify that data is read from the primary target when `trdyn` is released after the third rising clock edge and asserted on the third rising clock edge after `framen`.	✓	
15	Verify that data is written to the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
16	Verify that data is read from the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
17	Verify that data is written to the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
18	Verify that data is read from the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
19	Verify that data is written to the primary target when `trdyn` is released after the fifth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	

Table 16. Test Scenario: 1.8. Multi-Data Phase & trdyn Cycles (Part 2 of 3)			
#	Requirement	pci_c	
		Yes	No
20	Verify that data is read from the primary target when `trdyn` is released after the fifth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	
21	Verify that data is written to the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
22	Verify that data is read from the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
23	Verify that data is written to the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	
24	Verify that data is read from the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	
25	Verify that data is read from the primary target when `trdyn` is released after the second rising clock edge and asserted on the third rising clock edge after `framen`.	✓	
26	Verify that data is read from the primary target when `trdyn` released after the third rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
27	Verify that data is read from the primary target when `trdyn` released after the third rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
28	Verify that data is read from the primary target when `trdyn` released after the fourth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	
29	Verify that data is read from the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
30	Verify that data is read from the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	
31	Verify that data is read from the primary target when `trdyn` is released after the second rising clock edge and asserted on the third rising clock edge after `framen`.	✓	
32	Verify that data is read from the primary target when `trdyn` released after the third rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
33	Verify that data is read from the primary target when `trdyn` is released after the third rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
34	Verify that data is read from the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	
35	Verify that data is read from the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
36	Verify that data is read from the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	
37	Verify that data is written to the primary target when `trdyn` is released after the second rising clock edge and asserted on the third rising clock edge after `framen`.	✓	

Table 16. Test Scenario: 1.8. Multi-Data Phase & trdyn Cycles (Part 3 of 3)

#	Requirement	pci_c	
		Yes	No
38	Verify that data is written to the primary target when `trdyn` is released after the third rising clock edge and asserted on the fourth rising clock edge after `framen`.	✓	
39	Verify that data is written to the primary target when `trdyn` is released after the third rising clock edge and asserted on the fifth rising clock edge after `framen`.	✓	
40	Verify that data is written to the primary target when `trdyn` is released after the fourth rising clock edge and asserted on the sixth rising clock edge after `framen`.	✓	
41	Verify that data is written to the primary target when `trdyn` is alternately released for one clock cycle and asserted for one clock cycle after `framen`.	✓	
42	Verify that data is written to the primary target when `trdyn` is alternately released for two clock cycles and asserted for two clock cycles after `framen`.	✓	

Table 17. Test Scenario: 1.9. Bus Data Parity Error Single Cycles

#	Requirement	pci_c	
		Yes	No
1	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT memory write.	✓	
2	Verify that `perrn` is active two clocks after the first data phase (which had odd parity) on an IUT memory read.	✓	
3	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT memory read.	✓	
4	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT I/O write.	✓	
5	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT I/O read.	✓	
6	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT I/O read.	✓	
7	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT configuration write.	✓	
8	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT configuration read.	✓	
9	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT configuration read.	✓	

Table 18. Test Scenario: 1.10. Bus Data Parity Error Multi-Data Phase Cycles			
#	Requirement	pci_c	
		Yes	No
1	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT multi-data phase memory write.	✓	
2	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT multi-data phase memory read.	✓	
3	Verify that the IUT sets the parity error detected bit when odd.	✓	
4	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT dual-address multi-data phase write.	✓	
5	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT dual-address multi-data phase read.	✓	
6	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT dual-address multi-data phase read.	✓	
7	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT configuration multi-data phase write.	N.A.	
8	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT configuration multi-data phase read.	N.A.	
9	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT configuration multi-data phase read.	N.A.	
10	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT memory read multiple data phase.	✓	
11	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT memory read multiple data phase.	✓	
12	Verify that `perrn` is active two clocks after the first data phase (that had odd parity) on an IUT memory read line data phase.	✓	
13	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT memory read line data phase.	✓	
14	Verify that the IUT sets the parity error detected bit when the primary target asserts `perrn` on an IUT memory write and invalidate data phase.	✓	

Table 19. Test Scenario: 1.11. Bus Master Timeout

#	Requirement	pci_c	
		Yes	No
1	Memory write transaction terminates before the fourth data phase is completed.	✓	
2	Memory read transaction terminates before the fourth data phase is completed.	✓	
3	Configuration write transaction terminates before the fourth data phase is completed.	N.A.	
4	Configuration read transaction terminates before the fourth data phase is completed.	N.A.	
5	Memory read multiple transaction terminates before the fourth data phase.	✓	
6	Memory read line transaction terminates before the fourth data phase.	✓	
7	Dual address write transaction terminates before the fourth data phase is completed.	✓	
8	Dual address read transaction terminates before the fourth data phase is completed.	✓	
9	Memory write and invalidate terminates on line boundary.	✓	

Table 20. Test Scenario: 1.13. PCI Bus Master Parking

#	Requirement	pci_c	
		Yes	No
1	IUT drives `ad[31..0]` to stable values within eight PCI clocks of `gntn`.	✓	
2	IUT drives `cben[3..0]` to stable values within eight PCI clocks of `gntn`.	✓	
3	IUT drives `par` one clock cycle after IUT drives `ad[31..0]`.	✓	
4	IUT tri-states `ad[31..0]`, `cben[3..0]`, and `par` when `gntn` is released.	✓	

Table 21. Test Scenario: 1.14. PCI Bus Master Arbitration

#	Requirement	pci_c	
		Yes	No
1	IUT completes transaction when deasserting `gntn` coincides with asserting `framen`.	✓	

Table 22. Test Scenario: 2.1. Target Reception of an Interrupt Cycle			
#	Requirement	pci_c	
		Yes	No
1	IUT generates interrupts when programmed.	N.A.	
2	IUT clears interrupts when serviced (may include driver-specific actions).	N.A.	

Table 23. Test Scenario: 2.2. Target Reception of Special Cycle			
#	Requirement	pci_c	
		Yes	No
1	The devsel signal is not asserted by the IUT after a special cycle.	N.A.	
2	IUT receives encoded special cycle.	N.A.	

Table 24. Test Scenario: 2.3. Target Detection of Address & Data Parity Error for Special Cycle			
#	Requirement	pci_c	
		Yes	No
1	IUT reports address parity errors via serr.	N.A.	
2	IUT reports data parity errors via serr.	N.A.	
3	IUT keeps serr active for at least one clock cycle.	N.A.	

Table 25. Test Scenario: 2.4. Target Reception of I/O Cycles with Legal & Illegal Byte Enables			
#	Requirement	pci_c	
		Yes	No
1	IUT asserts trdy following second rising edge from framen on all legal BE''.	N.A.	
2	IUT terminates with target abort for each illegal BE''.	N.A.	
3	IUT asserts stopn.	N.A.	
4	IUT de-asserts stopn after framen deassertion.	N.A.	

Table 26. Test Scenario: 2.5. Target Ignores Reserved Commands			
#	Requirement	pci_c	
		Yes	No
1	IUT does not respond to reserved commands.	✓	
2	Initiator detects master abort for each transfer.	✓	
3	IUT does not respond to 64-bit cycle (dual address).		✓

Table 27. Test Scenario: 2.6. Target Receives Configuration Cycles			
#	Requirement	pci_c	
		Yes	No
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.	N.A.	

Table 28. Test Scenario: 2.7. Target Receives I/O Cycles with Address & Data Parity Errors			
#	Requirement	pci_c	
		Yes	No
1	IUT reports address parity errors via `serr` during I/O read/write cycles.	✓	
2	IUT reports data parity errors via `perr` during I/O write cycles.	✓	

Table 29. Test Scenario: 2.8. Target Receives Configuration Cycles with Address & Data Parity Errors			
#	Requirement	pci_c	
		Yes	No
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.	✓	

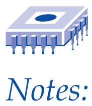
Table 30. Test Scenario: 2.9. Target Receives Memory Cycles			
#	Requirement	pci_c	
		Yes	No
1	IUT completes single memory read and write cycles appropriately.	✓	
2	IUT completes memory read line cycles appropriately.	✓	
3	IUT completes memory read multiple cycles appropriately.	✓	
4	IUT completes memory 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.	N.A.	

Table 31. Test Scenario: 2.10. Target Receives Memory Cycles with Address & Parity Errors			
#	Requirement	pci_c	
		Yes	No
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.	✓	

Table 32. Test Scenario: 2.11. Target Receives Fast Back-to-Back Cycles			
#	Requirement	pci_c	
		Yes	No
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 a second write selecting the IUT.	N.A.	
4	IUT responds to a memory write followed by a memory read with a read selecting the IUT.	N.A.	

Table 33. Test Scenario: 2.12. Target Performs Exclusive Address Cycles			
#	Requirement	pci_c	
		Yes	No
1	IUT responds to exclusive access by initiator and accepts lock.	N.A.	
2	IUT responds with a retry when second initiator attempts an access.	N.A.	
3	IUT responds to access releasing lock by initiator.	N.A.	
4	IUT responds to access by second initiator.	N.A.	

Table 34. Test Scenario: 2.13. Target Receives Cycles with irdy Used for Data Stepping			
#	Requirement	pci_c	
		Yes	No
1	IUT responds appropriately with a wait state inserted on phase 1 of 3 data phases.	✓	
2	IUT responds appropriately with a wait state inserted on phase 2 of 3 data phases.	✓	
3	IUT responds appropriately with a wait state inserted on phase 3 of 3 data phases.	✓	
4	IUT responds appropriately with a wait state inserted on all of 3 data phases.	✓	



Notes: