pci_b & pcit1 MegaCore Function
User Guide

June 1999

pci_b & pcit1 MegaCore Function User Guide
June 1999

A-UG-PCI-02
P25-04562-00

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.

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

I.S. EN ISO 9001

About this User Guide

June 1999

User Guide Contents

This user guide describes the Altera® `pci_b` and `pcit1`, including the specifications of the functions and how to use them in your designs. 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: ***1998 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. 2

Notes:

Introduction

Contents

June 1999

Notes:

June 1999, ver. 2

As programmable logic device (PLD) densities grow to over a 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 APEX® and FLEX® devices easily implement PCI applications, while leaving ample room for your custom logic. The devices are supported by the Altera Quartus™ and MAX+PLUS® II development systems, which allow 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 E2920 product series. Combined with Altera's APEX and 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_b` and `pcit1` functions.

Table 1. Altera PCI MegaCore Functions

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

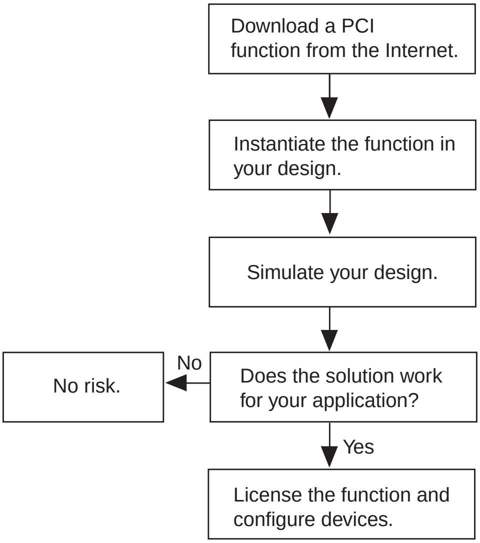
For more information on these MegaCore functions, refer to the following documents:

- *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_c 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 Quartus or 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.

Figure 1. OpenCore Design Flow



Altera Devices

The PCI MegaCore functions have been optimized and targeted for Altera PCI-compliant APEX and FLEX devices. APEX 20K devices offer complete system-level integration on a single device. The APEX MultiCore™ architecture delivers the ultimate in design flexibility and efficiency for high-performance System-on-a-Programmable Chip™ applications. With densities ranging from 100,000 to 1,000,000 gates, the APEX 20K architecture integrates look-up-table (LUT) logic, product-term logic, and memory into a single architecture, eliminating the need for multiple devices, saving board space, and simplifying the implementation of complex designs.

In the APEX MultiCore architecture, embedded system blocks (ESBs) and logic array blocks (LABs) are combined into MegaLAB™ structures. Each APEX 20K ESB can be configured as product-term logic, enabling APEX 20K devices to achieve unmatched integration efficiency, as LUT logic or as memory. The ESB can be configured as dual-port RAM, with a wide range of RAM widths and depths, or ROM in APEX 20K devices, and as content-addressable memory (CAM), a memory technology that accelerates applications requiring fast searches, in APEX 20KE devices.

The FLEX 10K embedded programmable logic device (PLD) family delivers the flexibility of traditional programmable logic with the efficiency and density of gate arrays with embedded memory. FLEX 10K devices feature embedded array blocks (EABs), which are 2 Kbits of RAM that can be configured as 256 × 8, 512 × 4, 1,024 × 2, or 2,048 × 1 blocks. Additionally, the FLEX 10K family offers all the features of programmable logic: ease-of-use, fast and predictable performance, register-rich architecture, and in-circuit reconfigurability (ICR). The 3.3-V FLEX 10KA devices and the MAX+PLUS II software combine to provide performance improvements of up to 100% over traditional FLEX 10K devices. Together, these features enable FLEX 10K devices to achieve the fastest high-density performance in the programmable logic market.

The 2.5-V FLEX 10KE devices support efficient implementation of dual-port RAM, and further enhance the performance of the FLEX 10K family. Designed for compliance with the 3.3-V 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 250,000 gates.

Altera's 5.0-V and 3.3-V FLEX 6000 devices deliver the flexibility and time-to-market of programmable logic at prices that are competitive with gate arrays. Featuring the OptiFLEX™ architecture, FLEX 6000 devices provide a flexible, high-performance, and cost-effective alternative to ASICs for high-volume production.

For more information on FLEX 10K and FLEX 6000 devices, refer to the *FLEX 10K Embedded Programmable Logic Family Data Sheet*, the *FLEX 10KE Embedded Programmable Logic Family Data Sheet*, and the *FLEX 6000 Programmable Logic Device Family Data Sheet*.

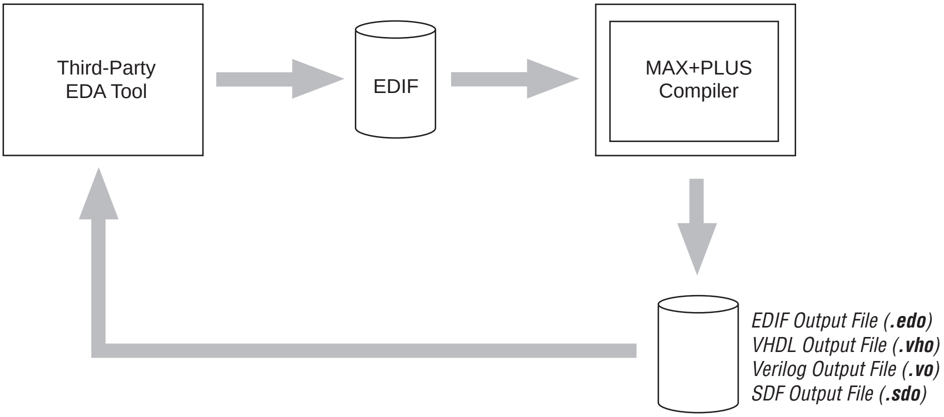
Software Tools

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

The Quartus and MAX+PLUS II software offer 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. Both software programs support 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^{SM}) 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. This testing includes using the PCI functions with a simple memory interface on the Altera PCI prototype board and with different chipsets, such as the Intel 430-FX and 440-FX PCI chipsets, and Intel 21052-AB and 21152-AA PCI-to-PCI bridges. Using the HP E2925A 32-bit, 33-MHz PCI Bus Analyzer and Exerciser in-system, 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.

References

Reference documents for the `pci_b` and `pcit1` functions 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.
- ***1999 Data Book***. Altera Corporation. San Jose, California. May 1999.

Notes:

Getting Started

Contents

June 1999

Notes:

Getting Started

June 1999, ver. 2

Altera PCI MegaCore™ functions provide solutions for integrating 32-bit PCI peripheral devices, including network adapters, graphic accelerator boards, and embedded control modules. The functions are optimized for Altera® APEX™ and 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.

☞ This section describes an example design flow using FLEX 10K devices and the MAX+PLUS® II software. For information on design flows using APEX devices and the Quartus software, contact your local Altera FAE.

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 Tools icon on the home page toolbar.

4. Click the MegaCore Functions link.

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

6. 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 3.x & Windows NT 3.51

For Windows 3.*x* and 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 (Part 1 of 2)

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_b or pcit1>	Contains the MegaCore function files.
****<pci_b or pcit1>**\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_b or pcit1>**\doc**	Contains documentation for the function.
****<pci_b or pcit1>**\examples**	The **\examples** directory has subdirectories containing examples for FLEX device/package combinations. Each subdirectory contains a Graphic Design File (**.gdf**) and an ACF. The **\examples** directory also contains the following subdirectories: ■ **\sim_top**, which contains a GDF and an ACF that can be used to perform functional compilation and simulation of the PCI MegaCore function. ■ **\walkthru**, which contains a sample design you can use to create a PCI design using the MAX+PLUS II software. This sample design familiarizes you with the Altera PCI MegaCore function and describes how to use it in your custom design. In the **\walkthru** directory, there is a **\solution** subdirectory that can be used as a reference as you implement the sample design.

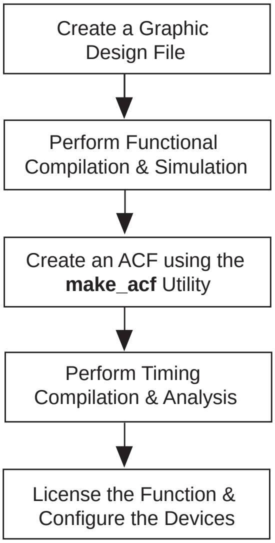
Table 1. PCI MegaCore Directories (Part 2 of 2)

Directory	Description
*****<pci_b or pcit1>***\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.
*****<pci_b or pcit1>***\sim\sig**	Contains the simulation files required by the PCI SIG ***Compliance Checklist, Revision 2.1***.

Walk-Through Overview

This section describes an entire 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 either the `pcit1` or `pci_b` 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 9.24 or higher is installed in the default location (**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 and an application design called **altr_app.tdf**. The **altr_app.tdf** design is a first-in first-out (FIFO) function, written in the Altera Hardware Description Language (AHDL), that is used to write and read data. This design and the PCI MegaCore function comprise the top-level design.
2. Perform functional compilation and simulation to verify that the circuit works correctly.
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 integrates the PCI MegaCore function with your own logic.

☞ 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 function. Choose **User Libraries** (Options menu) and specify the directories **c:\megacore\lib** and **c:\megacore**<*pcit1 or pci_b*>**\examples\walkthru**.
3. Create a directory to hold your design files, e.g., **c:\altr_app**.
4. Create a new GDF named **walkthru.gdf** and save it to your new directory (e.g., **c:\altr_app\walkthru.gdf**).

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

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

7. Set the parameter `BAR0="H"F0000000""`. Double-click on the **Parameters Field** of the symbol to open the **Edit Ports/Parameters** dialog box. If you are using the schematic from the **\solution** directory, you can skip this step.

☞ 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.

After you have entered your design, you are ready to perform functional simulation to verify that your circuit is working correctly.

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_b or pcit1*>**\examples\walkthru**<*target or master*>**.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 simulation shows several cycles that write and read from the local-side FIFO function.

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.

Generate the file **walkthru.acf** by performing the following steps. If you used the **walkthru.gdf** file from the **\solution** directory, you can skip the steps below and simply use the **walkthru.acf** that is also available in the **\solution** directory.

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

   ```
   c:\megacore\bin\make_acf ↵
   ```

2. 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:

   ```
   |XX:YY ↵
   ```

   Where:
   XX is the PCI function (`pci_b` or `pcit1`)
   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:

   ```
   walkthru ↵
   ```

   Type the path and name of the output acf file:

   ```
   c:\altr_app\walkthru.acf ↵
   ```

   Type the path and name of the input acf file:

   ```
   c:\megacore\<pci_b or pcit1>\acf\1030r240.acf ↵
   ```

☞ For a listing of the supported Altera device ACFs, refer to the readme file in **\megacore**<*pci_b or pcit1*>**\doc**.

3. 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., $\mathbf{t_{CO}}$ and $\mathbf{t_{PD}}$.
    - 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 EPROM, which loads the configuration data to the FLEX device during power-up. A Programmer Object File (**.pof**) is used to program the Configuration EPROM. 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 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 EPROM, 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**\\<*pcit1 or pci_b*>**\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 17 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 Models" on page 20.

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

Notes:

MegaCore Overview

June 1999, ver. 2

Features...

This section describes the features of both the `pci_b` and `pcit1` MegaCore™ functions. The `pci_b` function is a parameterized MegaCore function implementing a peripheral component interconnect (PCI) master/target interface. The `pcit1` function is a parameterized MegaCore function implementing a PCI target-only interface.

☞ This section describes an example design flow using FLEX® 10K devices and the MAX+PLUS® II software. For information on design flows using APEX™ devices and the Quartus™ software, contact your local Altera FAE.

- 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 hardware tested using the following hardware and software (see "Compliance Summary" on page 30 for details)
  - FLEX 10K PCI prototype board
  - HP E2925A PCI Bus Exerciser and Analyzer
  - Validated against common PCI chipsets, such as the Intel 430-FX and 440-FX, and Intel 21052-AB and 21152-AA PCI-to-PCI bridges
- Optimized for the APEX 20K, FLEX 10K, and FLEX 6000 architectures
- PCI master features (applies to the `pci_b` function only):
  - Zero-wait-state memory read/write operation (up to 132 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), and memory write and invalidate (MWI)
  - Bus parking
  - Independent master operation allows self configuration capability for host bridge applications
- PCI target features (applies to both the `pci_b` and `pcit1` functions):
  - Zero-wait-state memory read/write (up to 132 Mbytes per second)
  - Parity error detection

...and More Features

  - Up to 6 base address registers (BARs) with adjustable memory size and type
  - Capabilities list pointer support, which includes compact PCI Hot Swap support
  - Expansion ROM BAR support
  - CardBus CIS pointer register
  - Most PCI bus commands are supported; interrupt acknowledge, configuration read/write, memory read/write, I/O read/write, MRM, MRL, and MWI
  - Local side can request a target abort, retry, or disconnect
  - Local-side interrupt
- Configuration registers:
  - Parameterized registers: device ID, vendor ID, class code, revision ID, BAR0 through BAR5, subsystem ID, subsystem vendor ID, interrupt pin, maximum latency, minimum grant, capabilities list pointer, CIS pointer, and expansion ROM BAR
  - Parameterized default or preset base address (available for all 6) and expansion ROM base address
  - Non-parameterized registers: command, status, header type, latency timer, cache line size, and interrupt line

General Description

The `pci_b` MegaCore function (ordering code: PLSM-PCI/B) is a hardware-tested, high-performance, flexible implementation of the 32-bit, 33-MHz PCI master/target interface. The `pcit1` MegaCore function (ordering code: PLSM-PCIT1) is an implementation of the 32-bit, 33-MHz PCI target interface. Because these functions handle the complex PCI protocol and stringent timing requirements internally, designers can focus their engineering efforts on value-added custom development, significantly reducing time-to-market.

Optimized for Altera® APEX 20K, FLEX 10K, and FLEX 6000 architectures, the `pci_b` and `pcit1` functions support 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 functions to support unlimited cycles of zero-wait-state memory-burst transactions, thus achieving 132 Mbytes per second throughput, which is the theoretical maximum for a 32-bit, 33-MHz PCI bus.

In the `pci_b` 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. In addition, the `pci_b` function's separate local master and target data paths allow independent data prefetching and posting. Depending on the application, first-in first-out (FIFO) functions of variable length, depth, and type can be implemented in the local logic.

To ensure timing and protocol compliance, the functions have been vigorously hardware tested. See "Compliance Summary" on page 30 for more information on the hardware tests performed.

As parameterized functions, `pci_b` and `pcit1` have configuration registers that can be modified upon instantiation. These features provide scalability, adaptability, and efficient silicon usage. As a result, the same MegaCore functions can be used in multiple PCI projects with different requirements. For example, both functions offer 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_b` function.

Figure 1. pci_b Symbol

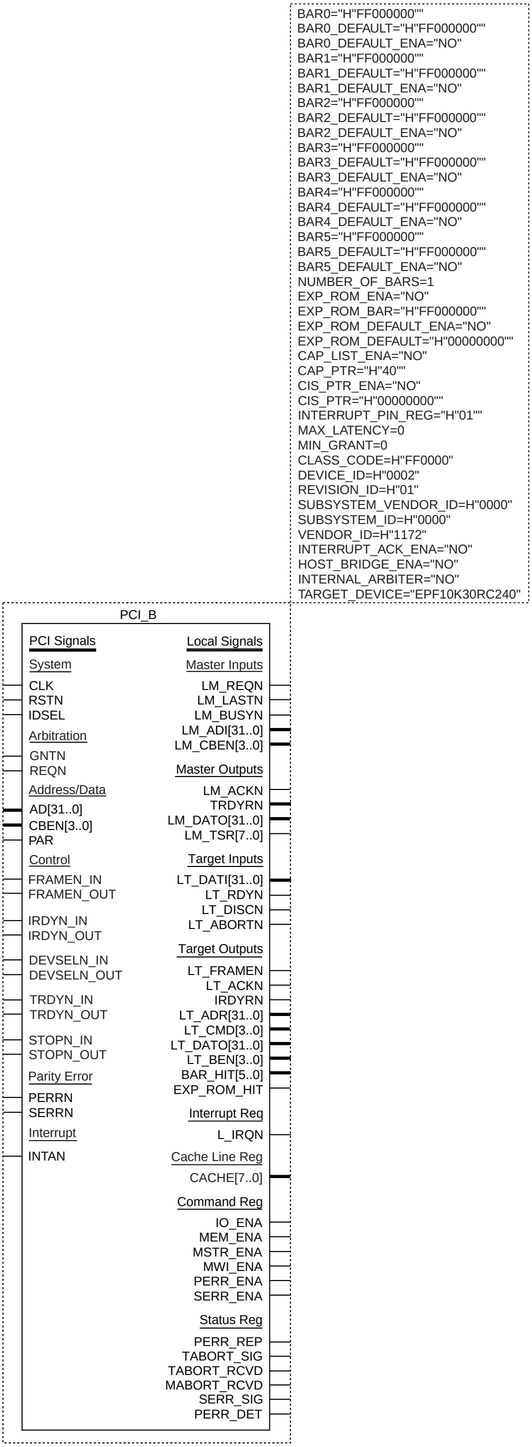
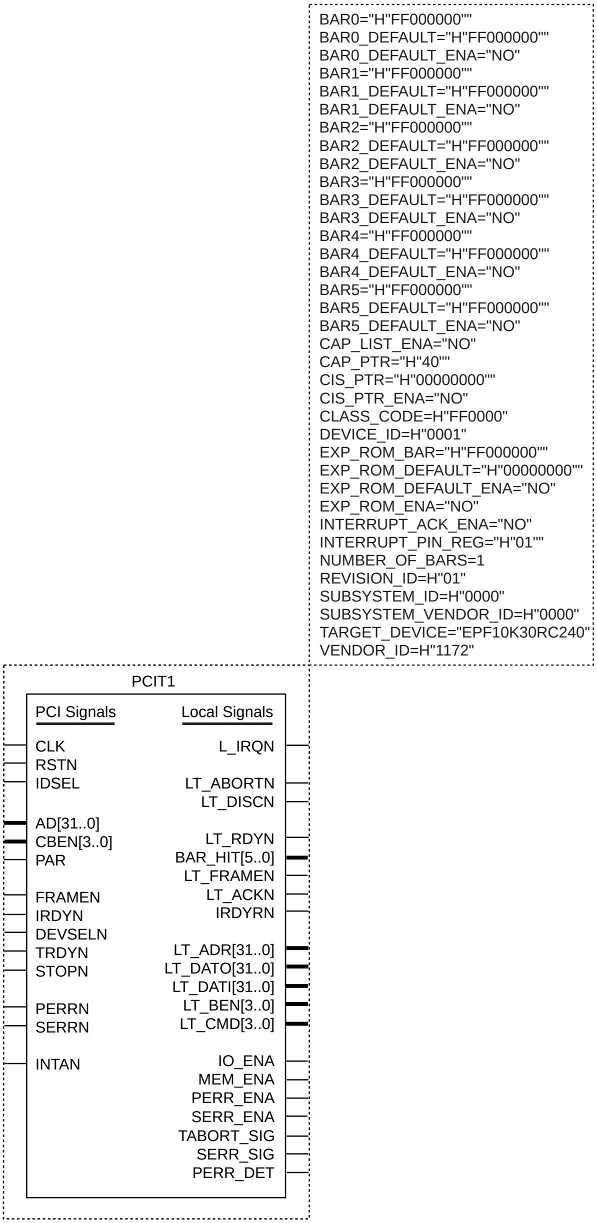


Figure 2 shows the symbol for the `pcit1` function.

Figure 2. pcit1 Symbol



Compliance Summary

The `pci_b` and `pcit1` functions are compliant with the requirements specified in the PCI SIG's ***PCI Local Bus Specification, Revision 2.2*** and ***Compliance Checklist, Revision 2. 1***. The functions are shipped with sample MAX+PLUS II Simulator Channel Files (**.scf**), which can be used to validate the functions in the MAX+PLUS II software. Additionally, functions are shipped with the simulation files required by the PCI SIG ***Compliance Checklist, Revision 2.1***. Consult the **readme** files provided in the **\sim\scf** and **\sim\sig** directories for a complete list and description of the included simulations.

In addition to simulation, Altera has performed extensive hardware testing on the `pci_b` and `pcit1` functions to ensure robustness and PCI compliance. The test platforms included the HP E2925A PCI Bus Excerciser and Analyzer, a PCI prototype board with a FLEX 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_b` and `pcit1` functions operate flawlessly under the most stringent conditions.

In addition to checking for data integrity, the HP E2925A PCI Bus Exerciser and Analyzer was used to ensure that the PCI bus is free of protocol violations. Each iteration of the test program transfers over 6.5 billion data bytes between the host memory and the MegaCore function. The test procedure was completed overnight, thus accounting for hundreds of iterations. The tests were repeated on multiple PCI platforms to ensure compatibility with various chipsets.

PCI Bus Signals

The following PCI bus signals are used by the `pci_b` and `pcit1` functions:

- *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 functions and the PCI bus.

Table 1. PCI Interface Signals (Part 1 of 2)

Name	Type	Direction, *Note (1)*	Polarity	Description
`clk`	Input	Input	–	Clock. The `clk` input provides the reference signal for all other PCI interface signals, except `rstn` and `intan`.
`rstn`	Input	Input	Low	Reset. The `rstn` input initializes the APEX 20K, FLEX 10K, and FLEX 6000 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`, *Note (2)*	Input	Input	Low	Grant. The `gntn` input indicates to the 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`, *Note (2)*	Output	Output	Low	Request. The `reqn` output indicates to the arbiter that the master wants to gain control of the PCI bus to perform a transaction.
`ad[31..0]`	Tri-State	Bidirectional	–	Address/data bus. The `ad[31..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.
`cben[3..0]`	Tri-State	Bidirectional (Input)	Low	Command/byte enable. The `cben[3..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	Bidirectional	–	Parity. The `par` signal is even parity across `ad[31..0]` and `cben[3..0]`. 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.
`idsel`	Input	Input	High	Initialization device select. The `idsel` input is a chip select for configuration transactions.
`framen`, *Note (3)*	STS	Bidirectional (Input)	Low	Frame. The `framen` 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[31..0]` and `cben[3..0]` buses. The `framen` signal remains asserted during the data operation and is deasserted to identify the end of a transaction.

Table 1. PCI Interface Signals (Part 2 of 2)

Name	Type	Direction, *Note (1)*	Polarity	Description
`irdyn`, *Note (3)*	STS	Bidirectional (Input)	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[31..0]` bus. In a read transaction, `irdyn` indicates that the master is ready to accept the data on the `ad[31..0]` bus.
`devseln`, *Note (3)*	STS	Bidirectional (Output)	Low	Device select. Target asserts `devseln` to indicate that the target has decoded its own address and accepts the transaction.
`trdyn`, *Note (3)*	STS	Bidirectional (Output)	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[31..0]` bus. In a write operation, `trdyn` indicates that the target is ready to accept data on the `ad[31..0]` bus.
`stopn`, *Note (3)*	STS	Bidirectional (Output)	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. See Table 8 on page 48 for more details.
`perrn`	STS	Bidirectional (Output)	Low	Parity error. The `perrn` signal indicates a data parity error. The `perrn` signal is asserted one clock following the `par` signal or two clocks following a data phase with a parity error.
`serrn`	Open-Drain	Output	Low	System error. The `serrn` signal indicates system error and address parity error. The `pci_b` function asserts `serrn` if a parity error is detected during an address phase and the required bits in the PCI command register are setup accordingly.
`intan`	Open-Drain	Output	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.

Notes:

(1) If a signal has a different direction for the `pcit1` function than in the `pci_b` function, the direction of the `pcit1` signal is shown in parenthesis and the direction for the `pci_b` function is shown without parenthesis.

(2) This signal is available in the `pci_b` function only.

(3) When implemented in the function, these signals are split into two pins, input, and output. For example, `trdyn` has the input `trdyn_in` and the output `trdyn_out`. Using two pins allows devices that do not meet set-up times for these signals to be used.

The PCI bus, FLEX 10K devices, and FLEX 6000 devices allow IEEE Std. 1149.1 Joint Test Action Group (JTAG) boundary-scan testing. To use JTAG boundary-scan testing, designers should connect the PCI bus JTAG pins with the FLEX 10K or FLEX 6000 device JTAG pins. See Table 2.

Table 2. Optional JTAG Signals

Name	Type	Polarity	Description
`TCK`	Input	High	Test clock. The `TCK` input is used to clock test mode and test data in and out of the device.
`TMS`	Input	High	Test mode select. The `TMS` input is used to control the state of the test access port (TAP) control in the device.
`TDI`	Input	High	Test data. The `TDI` input is used to shift the test data and instruction into the device.
`TDO`	Output	High	Test data. The `TDO` output is used to shift the test data and instruction out of the device.

Target Local-Side Signals

Table 3 summarizes the target interface signals that provide the interface between the MegaCore functions to the local-side peripheral device(s) during target transactions. These signals apply to both the `pci_b` and `pcit1` functions.

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

Name	Direction	Polarity	Description
`lt_dati[31..0]`	Input	–	Local target data bus input. The `lt_dati[31..0]` bus is driven active by the local-side peripheral device during target read transactions.
`lt_rdyn`	Input	Low	Local target ready. The local side asserts `lt_rdyn` to indicate a valid data input during target read, or to indicate that it is ready to accept data 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` directs the `pci_b` or `pcit1` 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_abortn`	Input	Low	Local target abort request. This signal indicates that a local peripheral device has encountered a fatal error and cannot complete the current transaction. Therefore, the local device requests the function to issue a target abort to the PCI master.

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

Name	Direction	Polarity	Description
`lt_discn`	Input	Low	Local target disconnect request. The `lt_discn` input is used to signal a request for either a retry or a disconnect depending on when the signal is asserted during a transaction. Refer to the target termination section for more details. The PCI protocol requires that the PCI target issues a disconnect whenever the transaction exceeds its memory space. In that case, it is the responsibility of the local side to assert `lt_discn`.
`lt_framen`	Output	Low	Local target frame request. The `lt_framen` output is asserted while the function is engaged in a PCI transaction. It is asserted one clock before the function asserts `devseln` (except during a retry, in which it is not asserted) and it is released after the last data phase of the transaction is completed on the PCI bus.
`lt_ackn`	Output	Low	Local target acknowledge. The 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 a wait state.
`irdyrn`	Output	Low	Local target initiator ready register. This signal is a registered output of the PCI `irdyn` signal. Usually, the `irdyrn` signal is used by the local side to monitor the status of the PCI bus data.
`lt_dato[31..0]`	Output	–	Local target data bus output. The `lt_dato[31..0]` bus is driven to the local-side peripheral device during target write transactions.
`lt_adr[31..0]`	Output	–	Local target address output. The `lt_adr[31..0]` bus represents the target memory address for the current local-side data phase. The function increments `lt_adr[31..0]` after a successful data transfer is completed on the local side i.e., `lt_rdyn` and `lt_ackn` are active during the same clock cycle.
`lt_cmd[3..0]`	Output	–	Local target command. The `lt_cmd[3..0]` bus represents the PCI command for the current claimed transaction. The `lt_cmd[3..0]` bus uses the same encoding scheme as the `cben[3..0]` bus.
`lt_ben[3..0]`	Output	Low	Local target byte enable bus. The `lt_ben[3..0]` bus represents the byte enable requests from the PCI master during data phases.

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

Name	Direction	Polarity	Description
`bar_hit[5..0]`	Output	High	Base address register hit. Asserting `bar_hit[5..0]` indicates that the PCI address matches that of a base address register and the PCI function has claimed the transaction. Each bit in `bar_hit[5..0]` is used for the BAR. Therefore, `bar_hit[0]` is used for BAR0. The `bar_hit[5..0]` bus has the same timing as the `lt_framen` signal.
`exp_rom_hit`	Output	High	Expansion ROM base address hit. Asserting this signal indicates that the PCI address matches that of the expansion ROM base address register and the PCI function has claimed the transaction. The `exp_rom_hit` signal has the same timing as the `lt_framen` signal.
`l_irqn`	Input	Low	Local interrupt request. The local-side peripheral device asserts `l_irqn` to signal a PCI bus interrupt. Asserting this signal forces the function to assert the `intan` signal for as long as `l_irqn` is asserted.

Master Local-Side Signals

Table 4 summarizes the `pci_b` master interface signals that provide the interface between the `pci_b` MegaCore function and the local-side peripheral device(s) during master transactions. The `pcit1` function is a target only; therefore, the signals in this section do not apply to it.

Table 4. pci_b Master Signals Interfacing to the Local Side (Part 1 of 3)

Name	Direction	Polarity	Description
`lm_reqn`	Input	Low	Local master request. The local side asserts this signal to request ownership of the PCI bus for a master transaction. The local-side device must supply the PCI bus address and command in the same clock cycle as when `lm_reqn` goes from high to low.
`lm_lastn`	Input	Low	Local master last. This signal is driven by the local side to request that the `pci_b` MegaCore master interface ends the current transaction. When the local side asserts this signal, the `pci_b` 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 4. pci_b Master Signals Interfacing to the Local Side (Part 2 of 3)

Name	Direction	Polarity	Description
`lm_busyn`	Input	Low	Local master busy. The local side asserts this signal to request a wait state for the `pci_b` burst from/to the local side. Asserting this signal causes the `pci_b` MegaCore function to deassert `irdyn` on the PCI bus to request wait states. A local data transfer occurs only if `lm_ackn` is asserted. Therefore, asserting `lm_busyn` causes `lm_ackn` to be deasserted.
`lm_adi[31..0]`	Input	–	Local master address/data bus. This signal is a local-side time multiplexed address/data bus. The local side must drive the transaction address at the same time it asserts `lm_reqn` to request the master transaction. In all other cases, `lm_adi[31..0]` carries data from the local side application for master write transactions. A local side data transfer is complete when `lm_ackn` is asserted. If the local side is unable to transfer data, it must assert `lm_busyn`. This action deasserts `lm_ackn`, indicating that a data transfer did not take place.
`lm_cben[3..0]`	Input	Low	Local master command/byte enable bus. This signal is a local-side time multiplexed command/byte enable bus. The local-side must drive the transaction command at the same time it asserts `lm_reqn` to request the master transaction. In all other cases `lm_cben[3..0]` carries byte enable information. In a burst transaction, it may not be possible to maintain synchronization between data transferred on the PCI bus and local side byte enable signals. Therefore, the `pci_b` MegaCore function only clocks the byte enable signals on the first data phase.
`lm_ackn`	Output	Low	Local master acknowledge. The `pci_b` MegaCore master interface asserts this signal when a local-side data transfer occurs. During a write transaction, the function asserts this signal when it internally latches data from the local side. In a read transaction, the `pci_b` function asserts this signal when it transfers data to the local side. If the local side is not ready to receive/send data, it must assert `lm_busyn`. Therefore, during a master transaction, the `pci_b` function deasserts `lm_ackn` if the `lm_busyn` is asserted or if the PCI target deasserts its `trdyn` signal. The operation of `lm_ackn` is different than the operation of lt_ackn.
`trdyrn`	Output	Low	Local master target read register. This signal is a registered version of the PCI `trdyn` signal. Usually, the signal is used by the local master device to monitor the status of data on the PCI bus.

Table 4. pci_b Master Signals Interfacing to the Local Side (Part 3 of 3)

Name	Direction	Polarity	Description
`lm_dato[31..0]`	Output	–	Local master data output. The `pci_b` function drives data to the local-side application during a master read transaction. A successful data transfer occurs when `pci_b` asserts `lm_ackn`. If the local side is unable to transfer data it must assert `lm_busyn`.
`lm_tsr[7..0]`	Output	–	Local master transaction status register bus. These signals inform the local interface the progress of the transaction. See Table 5 for a detailed description of the bits in this bus.

Table 5 shows definitions for the local master transaction status register outputs.

Table 5. pci_b Local Master Transaction Status Register Bit Definition

Bit Number	Bit Name	Description
0	`tsr_req`	Request. This signal indicates that the `pci_b` function is requesting mastership of the PCI bus, i.e., it is asserting its `reqn` signal.
1	`tsr_gnt`	Grant. This signal is active after the `pci_b` function has detected that `gntn` is asserted, and while `pci_b` is in the transaction address phase.
2	`tsr_dat_xfr`	Data transfer. This signal is active while the `pci_b` function is in data transfer mode. It is active after the address phase and remains active until the turn-around state begins.
3	`tsr_lat_exp`	Latency timer expired. `pci_b` terminated master transaction when the latency timer counter expired and `gntn` is not asserted.
4	`tsr_ret`	Retry detected. This signal indicates that `pci_b` 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.
5	`tsr_disc_wod`	Disconnect without data detected. This signal indicates that the `pci_b` signal terminated the master transaction because the target issued a disconnect without data.
6	`tsr_disc_wd`	Disconnect with data detected. This signal indicates that `pci_b` terminated the master transaction because the target issued a disconnect with data.
7	`tsr_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.

Configuration Space Output Signals

Table 6 shows configuration signals that are useful to local-side applications. For a detailed description of the registers, refer to the ***PCI Local Bus Specification*, Revision 2.2.**

Table 6. Configuration Space Output Signals

Name	Polarity	Description
`cache[7..0]`, *Note (1)*	–	PCI cache line register. The local-side application must use this signal when using the MWI and MRL commands.
`io_ena`	High	I/O space enable. PCI command register bit 0.
`mem_ena`	High	Memory space enable. PCI command register bit 1.
`mstr_ena`, *Note (1)*	High	Master enable. PCI command register bit 2.
`mwi_ena`, *Note (1)*	High	Memory write and invalidate enable. PCI command register bit 4.
`perr_ena`	High	Parity error response enable. PCI command register bit 6.
`serr_ena`	High	System error enable. PCI command register bit 8.
`perr_rep`, *Note (1)*	High	This signal indicates that `perrn` was detected during a master write transaction. PCI status register bit 8.
`tabort_sig`	High	This signal indicates that `pci_b` signaled target abort. PCI status register bit 11.
`tabort_rcvd`, *Note (1)*	High	This signal indicates that `pci_b` received a target abort. PCI status register bit 12.
`mabort_rcvd`, *Note (1)*	High	This signal indicates that `pci_b` received a master abort. PCI status register bit 13.
`serr_sig`	High	Signaled system error. PCI status register bit 14.
`perr_det`	High	This signal indicates that `pci_b` detected a data or address parity error. PCI status register bit 15.

Note:
(1) These signals apply to the `pci_b` function only.

Parameters

Parameters are an innovative feature in Altera's AHDL that allow you to customize MegaCore functions to fit your particular application. In the `pci_b` and `pcit1` functions, parameters allow you to set read-only registers in the configuration space, such as `DEVICE_ID`, control available features within the functions, or optimize a design for a target application. For more information on how the parameters control the configuration space, see "Configuration Registers" on page 56. Table 7 describes the `pci_b` and `pcit1` MegaCore function parameters.

Table 7. PCI MegaCore Function Parameters (Part 1 of 5)

Name	Format	Default Value	Description
`BAR`*n*	Hexadecimal	`H"FF000000"`	Base address register *n*. *n* corresponds to the BAR number and can be from 0 to 5. *Note (1)*
`BARn_DEFAULT_ENA`	String	`"NO"`	Default base address register enable. The `BARn_DEFAULT_ENA` parameter indicates that the user wants to use a default base address at power-up. *n* corresponds to the BAR number and can be from 0 to 5.
`BARn_DEFAULT`	Hexadecimal	`H"FF000000"`	Default base address register. *n* corresponds to the base address register number and can be from 0 to 5. `BARn_DEFAULT` is a 32-bit hexadecimal value that permanently sets the value stored in the corresponding BAR. This parameter is ignored if the corresponding `BARn_DEFAULT_ENA` parameter is not set to `"YES"`. When the corresponding `BARn_DEFAULT_ENA` parameters is set to `"YES"`, the `pci_b` and `pcit1` functions return the value in `BARn_DEFAULT` during a configuration read. To detect a base address register hit, the `pci_b` and `pcit1` functions compare the incoming address to the upper bits of the `BARn_DEFAULT` parameter. The corresponding `BAR`*n* parameter is still used to define the programmable setting of the individual BAR such as address space type and number of decoded bits.
`CAP_LIST_ENA`	String	`"NO"`	Capabilities list enable. The `CAP_LIST_ENA` parameter determines if the capabilities list will be enabled in the configuration space. When this parameter is set to `"YES"`, it sets capabilities list bit (bit4) of the status register and sets the capabilities register to the value of `CAP_PTR`.
`CAP_PTR`	Hexadecimal	`H"40"`	Capabilities pointer. The `CAP_PTR` sets the value stored in the capabilities pointer register. The value set in this pointer should be the address of the first entry of the extended capabilities list is stored. This parameter is ignored if the `CAP_LIST_ENA` parameter is set to `"NO"`.

Table 7. PCI MegaCore Function Parameters (Part 2 of 5)

Name	Format	Default Value	Description
`CIS_PTR_ENA`	String	`"NO"`	CardBus CIS pointer enable. The `CIS_PTR_ENA` parameter enables the CardBus CIS pointer register. When this parameter is set to `"NO"` `pci_b` and `pcit1` return `H"00000000"` during a configuration read to the `CAP_PTR` register.
`CIS_PTR`	Hexadecimal	`H"00000000"`	CardBus CIS pointer. The `CIS_PTR` sets the value stored in the CIS pointer register. The CIS pointer register indicates where the CIS header is located. For more information, refer to the *PCMCIA Specification version 2.10*. `pci_b` and `pcit1` ignore this parameter if `CIS_PTR` is not set to `"YES"`. In other words, if the `CIS_PTR_ENA` parameter is set to `"YES"`, the `pci_b` and `pcit1` return the value in `CIS_PTR` during a configuration read to CIS pointer register. `pci_b` and `pcit1` return `H"00000000"` during a configuration read to CIS when `CIS_PTR_ENA` is set to `"NO"`.
`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_b` or `pcit1` configuration space. The value entered for this parameter must be a valid PCI SIG-assigned class code register value.
`DEVICE_ID`	Hexadecimal	`H"0001"`	Device ID register. This parameter is a 16-bit hexadecimal value that sets the device ID register in the `pci_b` or `pcit1` configuration space. Any value can be entered for this parameter.
`EXP_ROM_BAR_ENA`	String	`"NO"`	Expansion ROM base address register enable. The `EXP_ROM_BAR_ENA` parameter enables the capability for the expansion ROM base address register. If this parameter is set to `"YES"`, `pci_b` and `pcit1` use the value stored in `EXP_ROM_BAR` to set the size and number of bits decoded of the expansion ROM BAR. Otherwise, the expansion ROM BAR is read only and `pci_b` and `pcit1` return `H"00000000"` when the expansion ROM BAR is read.

Table 7. PCI MegaCore Function Parameters (Part 3 of 5)

Name	Format	Default Value	Description
`EXP_ROM_BAR`	Hexadecimal	`H"FF000000"`	Expansion ROM base address register. The `EXP_ROM_BAR` parameter indicates the base address and size information for the expansion ROM. According to the PCI specification, only bits 31 through 11 can be decoded. This parameter works the same way as the `BARn` parameters. If the `EXP_ROM_BAR_ENA` parameter is set to `"NO"`, the `EXP_ROM_BAR` parameter is ignored.
`EXP_ROM_DEFAULT_ENA`	String	`"NO"`	Expansion ROM base address default enable**.** The `EXP_ROM_DEFAULT_ENA` parameter specifies a default address for the expansion ROM base address.
`EXP_ROM_DEFAULT`	Hexadecimal	`H"FF000000"`	Expansion ROM base address default**.** `EXP_ROM_DEFAULT` is the default expansion ROM base address. This parameter is ignored when `EXP_ROM_DEFAULT_ENA` is set to `"NO"`. When `EXP_ROM_DEFAULT_ENA` is set to `"YES"`, the `pci_b` and `pcit1` functions return the value in `EXP_ROM_DEFAULT` during a configuration read. To detect base address hits for the expansion ROM, the `pci_b` and `pcit1` functions compare the input address to the upper bits of `EXP_ROM_DEFAULT`. `EXP_ROM_BAR_ENA` must be set to enable expansion ROM support, and the `EXP_ROM_BAR` parameter setting defines the number of decoded bits.
`HOST_BRIDGE_ENA`, *Note (2)*	String	`"NO"`	This parameter permanently enables the master capability in the `pci_b` function to be used in host bridge applications, which allows the `pci_b` function to generate the required configuration transactions during power-up. If the `pci_b` function is used as a host bridge, the local-side application must be able to perform master transactions at power up. The `pci_b` MegaCore function can generate configuration cycles for other PCI bus agents, including its own target.

Table 7. PCI MegaCore Function Parameters (Part 4 of 5)

Name	Format	Default Value	Description
`INTERNAL_ARBITER`, *Note (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_b` 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.
`INTERRUPT_ACK_ENA`	String	`"NO"`	Interrupt acknowledge enable. The `INTERRUPT_ACK_ENA` parameter enables support for the interrupt-acknowledge command. When set to `"NO"`, the `pci_b` or `pcit1` function ignores the interrupt acknowledge command. When set to `"YES"`, `pci_b` or `pcit1` responds to the interrupt acknowledge command. The `pci_b` and `pcit1` functions treat the interrupt acknowledge command as a regular target memory read. The local side must implement the necessary logic to respond to the interrupt controller.
`INTERRUPT_PIN_REG`	Hexadecimal	`H"01"`	Interrupt pin register. The `INTERRUPT_PIN_REG` parameter indicates the value of the interrupt pin register in the configuration space address location `3DH`. This parameter can be set to two possible values: `H"00"` to indicate that no interrupt support is needed, or `H"01"` to implement `intan`. When the `INTERRUPT_PIN_REG` parameter is set to `H"00"`, `intan` will be stuck at V_{CC} and the `l_irqn` local interrupt request input pin will not be required.
`MAX_LATENCY` *Note (2)*	Hexadecimal	`H"0"`	Maximum latency register. This parameter is an 8-bit hexadecimal value that sets the maximum latency register in the `pci_b` configuration space. This parameter must be set according to the guidelines in the PCI specifications.

Table 7. PCI MegaCore Function Parameters (Part 5 of 5)

Name	Format	Default Value	Description
`MIN_GRANT`, *Note (2)*	Hexadecimal	`H"0"`	Minimum grant register. This parameter is an 8-bit hexadecimal value that sets the minimum grant register in the `pci_b` 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_b` and `pcit1` MegaCore functions sequentially instantiate 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_b` or `pcit1` configuration space.
`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_b` or `pcit1` configuration space. The user can choose a value that uniquely identifies the application.
`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_b` or `pcit1` configuration space. The value for this parameter must be a valid PCI SIG-assigned vender ID number.
`TARGET_DEVICE`, *Note (4)*	String	`EPF10K30RC240`	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_b` or `pcit1` 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. If the `NUMBER_OF_BARS` parameter is less than the maximum number of available BARs, the corresponding `BAR`*n* parameter value is ignored. Each `BAR`*n* parameter is a 32-bit value that controls the BAR options per the definition of a BAR, according to the ***PCI Local Bus Specification, Revision 2.2***. For example, bit 0 of the `BAR`*n* parameter controls the BAR type similar to bit 0 of the BAR. For more details about how these parameters affect the BARs, refer to "Base Address Registers" on page 63.

(2) These parameters apply to the `pci_b` function only.

(3) When the `INTERRUPT_PIN_REG` parameter is set to `H"00"`, `intan` remains at V_{CC} and the `l_irqn` local interrupt request input pin is not required.

(4) For a listing of the supported Altera FLEX devices, refer to the **readme** file for your PCI MegaCore function.

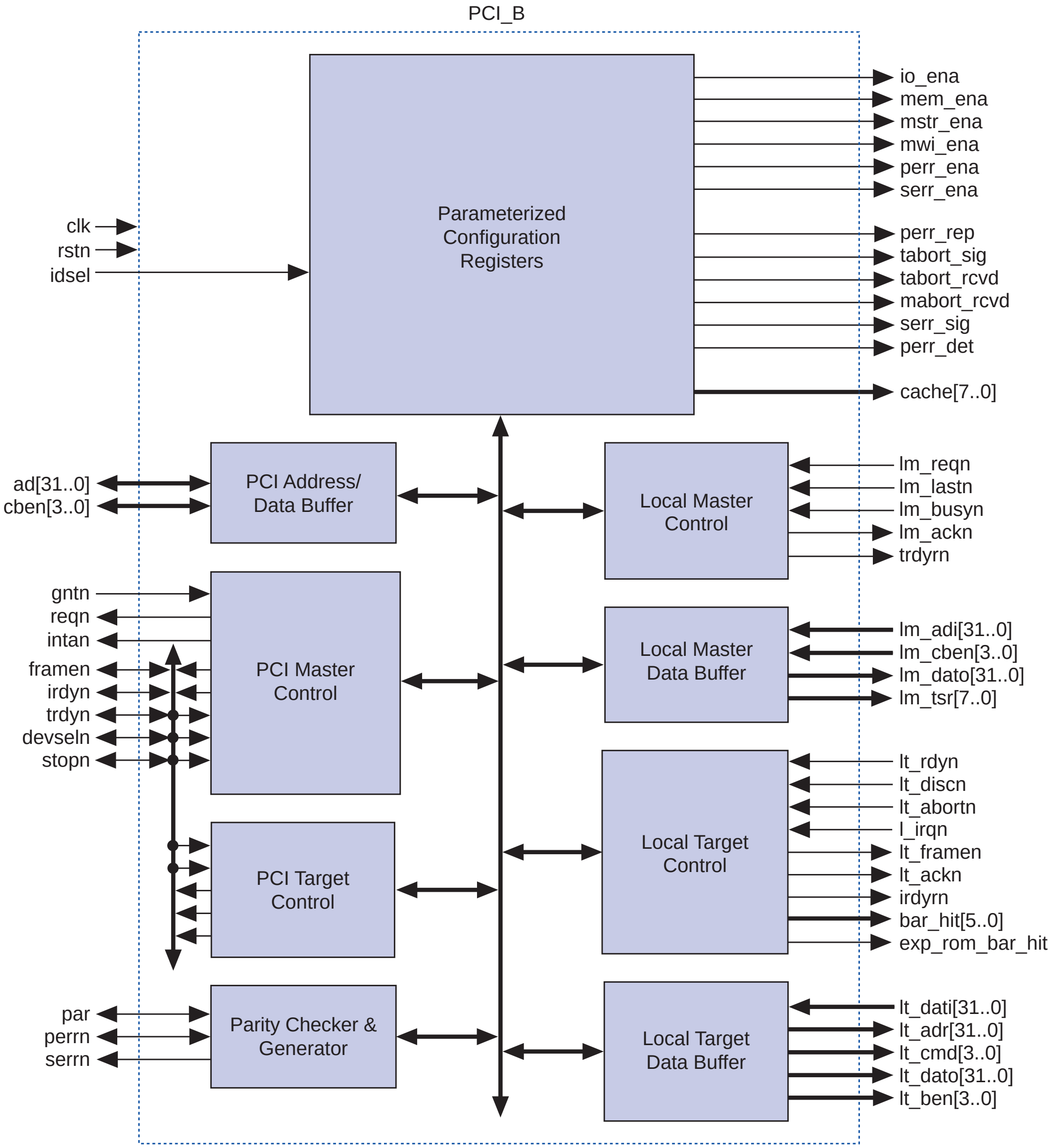
Functional Description

This section provides a general overview of `pci_b` and `pcit1` operations. The `pci_b` function consists of three main elements:

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

Figure 3 shows the `pci_b` functional block diagram.

Figure 3. pci_b Functional Block Diagram

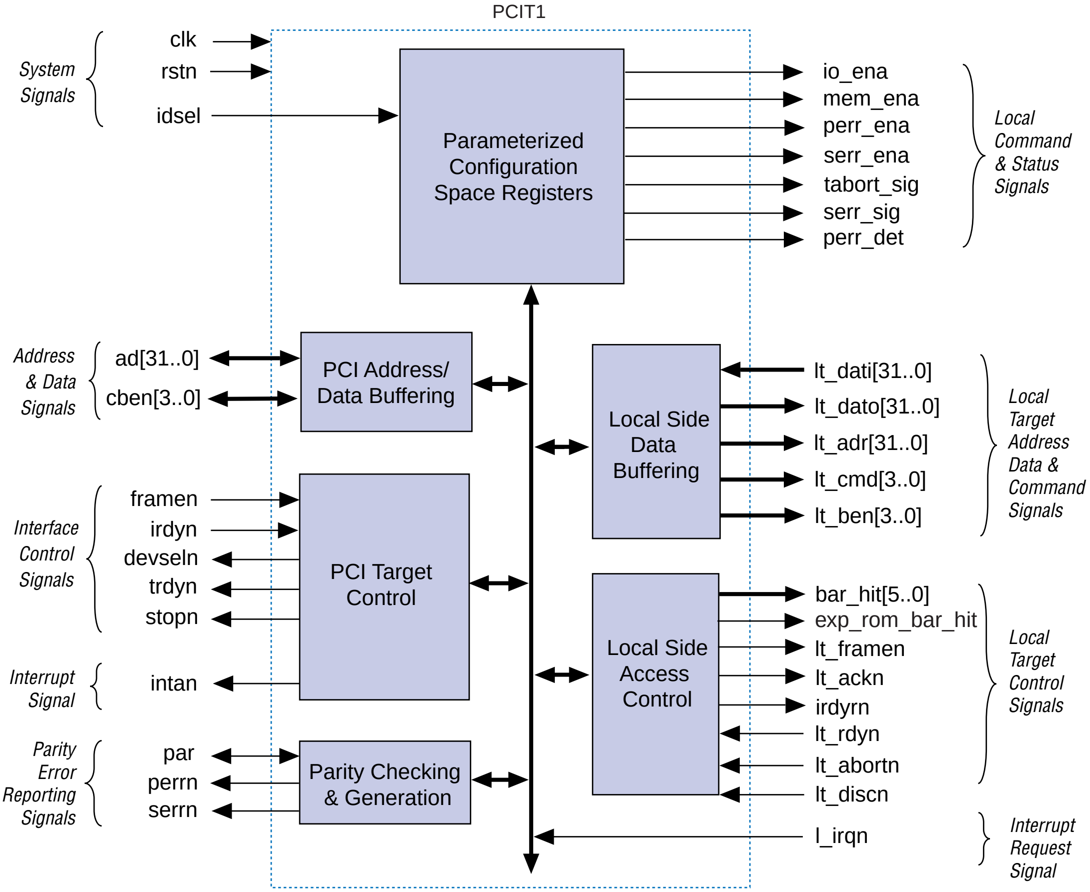


The `pcit1` function consists of two main elements:

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

Figure 4 shows the `pcit1` functional block diagram.

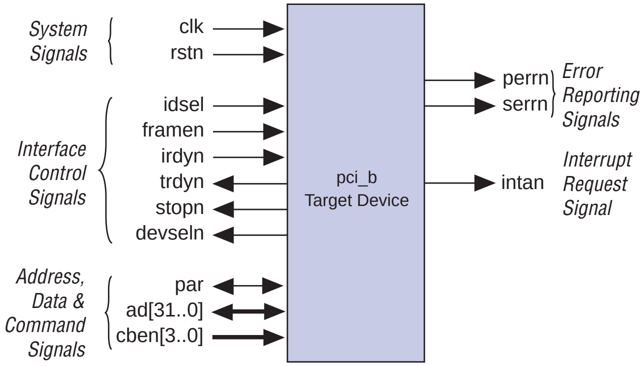
Figure 4. pcit1 Functional Block Diagram



Target Device Signals & Signal Assertion

Figure 5 illustrates the signal directions for a PCI device connecting to the PCI bus in target mode. These signals apply to the `pcit1` function and the `pci_b` 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 5. Target Device Signals



A 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 8 shows the control signal combinations possible on the PCI bus during a PCI transaction. The `pci_b` or `pcit1` function processes the PCI signal assertion from the local side. Therefore, the `pci_b` or `pcit1` 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 84 for more details.

Table 8. Control Signal Combination Transfer

Type	devseln	trdyn	stopn	irdyn
Claim transaction	Assert	Don't care	Don't care	Don't care
Retry, *Note (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, *Note (2)*	De-assert	De-assert	Assert	Don't care
Successful data transfer	Assert	Assert	De-assert	Assert
Target wait state	Assert	De-assert	De-assert	Assert
Master wait state	Assert	Assert	De-assert	De-assert

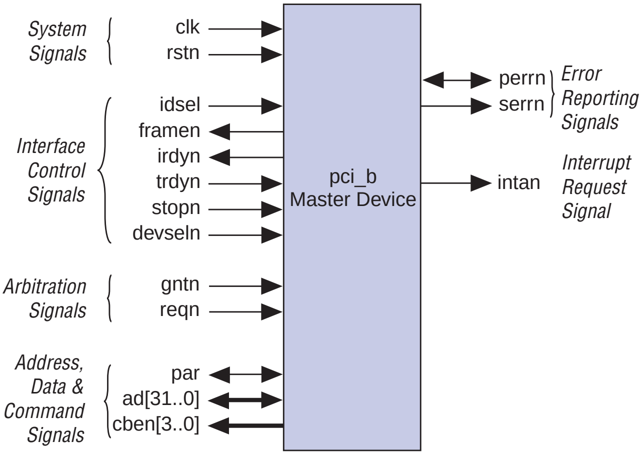
Notes:
(1) A retry occurs before the first data phase.
(2) A device must assert the `devseln` signal for at least one clock before it signals an abort.

The `pci_b` and `pcit1` functions support unlimited burst access cycles. Therefore, they can achieve a throughput of 132 Mbytes per second, the theoretical maximum bandwidth of a 32-bit, 33-MHz PCI bus. 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 tradeoff between bandwidth and increased latency.

Master Device Signals & Signal Assertion

Figure 6 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_b` function operating as a master on the PCI bus. This section applies to the `pci_b` function only.

Figure 6. pci_b Master Device Signals



A `pci_b` master sequence begins when the local side asserts `lm_reqn` to request mastership of the PCI bus. After receiving `gntn` from the PCI bus arbiter and after the bus idle state is detected, the `pci_b` 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.

When the `pci_b` function is ready to present or accept data on the bus, it asserts `irdyn`. At this point, the `pci_b` 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 8 on page 48 shows the possible control signal combinations on the PCI bus during a transaction. The `pci_b` function signals that it is ready to present or accept data on the bus by asserting `irdyn`. At this point, the `pci_b` 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[7..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, latency timer expiration, and `gntn` not being asserted. See "Abnormal Master Transaction Termination" on page 97 for more details.

Specifications

Contents

June 1999

Notes:

June 1999, ver. 2

This section describes the specifications of the `pci_b` and `pcit1` MegaCore functions, including the supported 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_b` and `pcit1` MegaCore functions. The commands supported by the master can be initiated by the `pci_b` function, and the commands supported by the target can be responded to by either the `pci_b` or `pcit1` functions.

Table 1. PCI Bus Commands

cben[3..0] Value	Bus Command Cycle	Master	Target
`0000`	Interrupt acknowledge, *Note (1)*	Yes	Yes
`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, *Note (2)*	Yes	Yes
`1101`	Dual address cycle	Ignored	Ignored
`1110`	Memory read line, *Note (2)*	Yes	Yes
`1111`	Memory write and invalidate, *Note (2)*	Yes	Yes

Notes:

(1) The `INT_ACK_ENA` parameter must be set to `"YES"` to support the interrupt acknowledge command, which is aliased with a memory read.

(2) 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 `lt_cmd[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 functions respond to the standard memory read/write, cache memory read/write, I/O read/write, configuration read/write, and interrupt acknowledge commands. The bus commands are discussed in greater detail in "Target Mode Operation" on page 71 and "Master Mode Operation" on page 88.

In master mode, the `pci_b` 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_b` and `pcit1` functions.

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_b` and `pcit1` functions. The latency timer, cache line size maximum latency, and minimum grant registers are not supported by the `pcit1` function because they are only applicable to a PCI master interface.

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	CardBus CIS Pointer			
2CH	Subsystem ID		Subsystem Vendor ID	
30H	Expansion ROM Base Address Register			
34H	Reserved			Capabilities List
38H	Reserved			
3CH	Maximum Latency	Minimum Grant	Interrupt Pin	Interrupt Line

A write to unused registers completes normally but the data is ignored.

Table 3 summarizes the `pci_b` and `pcit1` 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_b` or `pcit1` 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, *Note (1)*
`0D`	`0D-0D`	1/1	Read/write	`lat_tmr`	Latency timer, *Note (1)*
`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
`28`	`28–2B`	4/4	Read	`cis_ptr`	CardBus CIS pointer
`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	`exp_rom_bar`	Expansion ROM BAR
`34H`	`34–34`	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, *Note (1)*
`3F`	`3F-3F`	1/1	Read	`max_lat`	Maximum latency, *Note (1)*

Note:
(1) These registers are supported by the `pci_b` function only.

Vendor ID Register

Vendor ID is a 16-bit read-only register that identifies the manufacturer of the device (e.g., Altera for the `pci_b` and `pcit1` functions). 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 (e.g., Altera assigned the value of the device ID register for the `pci_b` and `pcit1` functions). The default value of the device ID register is `0003` 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_b` or `pcit1` 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_b` or `pcit1` function respond to the PCI bus memory accesses as a target.
2	`mstr_ena`, *Note (1)*	Read/write	Master enable. When high, `mstr_ena` allows the `pci_b` function to acquire mastership of the PCI bus.
3	Unused	–	–
4	`mwi_ena`, *Note (1)*	Read/write	Memory write and invalidate enable. This bit controls whether the master may generate a MWI command. Although the `pci_b` 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_b` or `pcit1` 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_b` or `pcit1` 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	–	–

Note:

(1) These bits are only supported by the `pci_b` function. In the `pcit1` function, these bits are hardwired to ground.

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

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.
7..5	Unused	–	–
8	`dat_par_rep`, *Note (1)*	Read/write	Data parity reported. When high, `dat_par_rep` indicates that during a read transaction the `pci_b` function asserted the `perrn` output as a master device, or that during a write transaction the `perrn` output was asserted by 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 `perr_rep` output
10..9	`devsel_tim`	Read	Device select timing. The `devsel_tim` bits indicate target access timing of the `pci_b` or `pcit1` function via the `devseln` output. The `pci_b` and `pcit1` functions are designed to be slow target devices, 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_b` or `pcit1` 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 `tabort_sig` output.
12	`targ_abort_rcvd`	Read/write	Target abort. When high, `targ_abort_rcvd` indicates that the current target device transaction has been terminated. This bit is driven to the local side on `targ_abort_rcvd` output.
13	`mstr_abort`, *Note (1)*	Read/write	Master abort. When high, `mstr_abort` indicates that the current master device transaction has been terminated. This bit is driven to the local side on the `mabort_rcvd` output.
14	`serr_set`	Read/write	Signaled system error. When high, `serr_set` indicates that the `pci_b` or `pcit1` function drove the `serrn` output active, i.e., an address phase parity error has occurred. The `pci_b` or `pcit1` 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 `serr_sig` output.
15	`det_par_err`	Read/write	Detected parity error. When high, `det_par_err` indicates that the `pci_b` or `pcit1` function detected either an address or data parity error. Even if parity error reporting is disabled (via `perr_ena`), the `pci_b` or `pcit1` function sets the `det_par_err` bit. This signal is driven to the local side on the `perr_det` output.

Note:

(1) These bits are supported by the `pci_b` function only. In the `pcit1` function, these bits are hardwired to ground.

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_b` or `pcit1` function). For Altera PCI MegaCore functions, the default value of the revision ID register is the revision number of the `pci_b` or `pcit1` 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 `H"FF0000"`. 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, and is supported by the `pci_b` function only. 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, and is supported by the `pci_b` function only. The register defines the maximum amount of time, in PCI bus clock cycles, that the `pci_b` function can retain ownership of the PCI bus. After initiating a transaction, the `pci_b` function decrements its latency timer by one on the rising edge of each clock. The default value of the latency timer register is `H"00"`. 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_b` or `pcit1` function as a single-function device. The default value of the header type register is `H"00"`. 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_b` or `pcit1` 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_b` or `pcit1` function is instantiated. The logic for the unused BARs is reduced automatically by the MAX+PLUS II software when you compile the `pci_b` or `pcit1` function.

Each BAR has its own parameter BAR*n* (where *n* is the BAR number). Each BAR*n* 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 in the 32-bit address space
- 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_b` or `pcit1` 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_b` or `pcit1` memory address space. Only the following two possible values are valid for `pci_b` and `pcit1`: locate memory space any where in the 32-bit address space and locate memory space below 1 Mbyte.
3	`pre_fetch`	Read/write	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 `H"FF000000"` is a legal value for a BAR*n* parameter, but the value `H"FF700000"` is not, because bits 24 and 22 are 1s and bit 23 is 0. As another example, if you set the `BAR0` parameter to `H"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

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_b` or `pcit1` 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 0. 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 `H"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.

In some applications, one or more BARs must be hardwired. The `pci_b` and `pcit1` functions allow you to set default base addresses that can be used to claim transactions without requiring the configuration of the corresponding BARs. To implement this feature, set the `BARn_DEFAULT_ENA` parameter to `"YES"`. In this case, `pci_b` or `pcit1` uses the content of the `BARn_DEFAULT` parameter as the default base address (*n* corresponds to the BAR number and can be from 0 to 5). When using `BARn_DEFAULT`, you must set the corresponding `BAR`*n* parameter appropriately to indicate the BAR settings, such as address space type and number of decoded bits. When `BARn_DEFAULT_ENA` is set to `"NO"`, `BARn_DEFAULT` is ignored.

☞ When you use `BARn_DEFAULT`, the corresponding BARs become read-only. A configuration write to this BAR will proceed normally. However, a configuration read of these registers will return the value in the `BARn_DEFAULT` parameter.

CardBus CIS Pointer Register

The card information structure (CIS) pointer register is a 32-bit read-only register that points to the beginning of the CIS. This optional register is used by devices that have the PCI and CardBus interfaces on the same silicon. By default, the `pci_b` and `pcit1` MegaCore functions do not support this register. To enable support, set the `CIS_PTR_ENA` parameter to `"YES"` and the `CIS_PTR` parameter to the appropriate value. Table 15 shows this register's format. For more information on the CardBus CIS pointer register, refer to the ***PCMCIA Specification version 2.10.***

Table 15. CIS Pointer Register Format

Data Bit	Mnemonic	Read/Write	Definition
0..2	`adr_space_ind`	Read	Address space indicator. The value of these bits indicates that the CIS pointer register is pointing to one of the following spaces: configuration space, memory space, or expansion ROM space.
3..27	`adr_offset`	Read	Address space offset. This value gives the address space's offset indicated by the address space indicator.
31..28	`rom_im`	Read	ROM image. These bits are the uppermost bits of the address space offset when the CIS pointer register is pointing to an expansion ROM space.

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 16. 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 16. 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 17. The default value of the subsystem ID register is `0000` hex. However, designers can change the value by setting the `SUBSYSTEM_ID` parameter.

Table 17. Subsystem ID Register Format

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

Expansion ROM BAR

PCI devices on expansion boards require local EPROMs (or expansion ROMs). The 32-bit expansion ROM BAR handles the expansion ROM's base address and size information. This register functions like a 32-bit BAR except the encoding and use of the bottom bits is different. Bit 0 is read/write and is used to indicate whether the device accepts accesses to its expansion ROM or not. The upper 21 bits correspond to the upper 21 bits of the expansion ROM base address. The size of the address space must not be greater than 16 Mbytes. To enable expansion ROM BAR support in `pci_b` and `pcit1`, set the `EXP_ROM_ENA` parameter to `"YES"` and the `EXP_ROM_BAR` parameter to the appropriate value. When the `EXP_ROM_ENA` parameter is set to `"NO"`, the `EXP_ROM_BAR` parameter is ignored. The expansion ROM BAR in `pcit1` and `pci_b` is formatted according to the ***PCI Local Bus Specification, Revision 2.2***. See Table 18.

In some applications, you may want to hardwire the value of the expansion ROM's base address. To use this feature in `pci_b` and `pcit1`, set the `EXP_ROM_DEFAULT_ENA` parameter to `"YES"`. In this case, the value in the `EXP_ROM_DEFAULT` parameter is used as the expansion ROM's base address. The `EXP_ROM_ENA` parameter must be set to `"YES"` and the `EXP_ROM_BAR` parameter setting defines the expansion ROM's size.

Table 18. Expansion ROM Base Address Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	`exp_rom_ena`	Read/write	Address decode enable. The `exp_rom_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 added to the PCI configuration space to support the capabilities added to PCI after ***PCI Local Bus Specification, Revision 2.1*** was published. The PCI status register supports this optional register when the capabilities list bit (bit 4) is set to indicate that the capabilities pointer is located at offset `H"34"` in the configuration space. The pointer points to the first item in the capabilities list. To enable the capabilities pointer register in `pci_b` and `pcit1`, set the `CAP_LIST_ENA` parameter to `"YES"`. In this case, the value in the `CAP_PTR` parameter is the address of the first item in the capabilities list. See Table 19. The address indicated by the pointer must be `H"40"` or greater, and each capability must be within a DWORD boundary. When you implement the capabilities pointer register in `pci_b` and `pcit1`, the locations at offset `H"40"` and beyond become accessible.

The capabilities pointer register allows the implementation of a compact PCI hot swap register in `pci_b` or `pcit1`. The capabilities list also includes PCI power management, accelerated graphics port (AGP), and others. For a complete list of the items and their ID's, see appendix H in the ***PCI Local Bus Specification, Revision 2.2***.

Table 19. 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 20.

Table 20. 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_b` or `pcit1` function's PCI bus interrupt request line. Both functions support only one interrupt request line: `intan`. The interrupt pin register is controlled by the `INTERRUPT_PIN_REG` parameter, which can be set to only two possible values: `H"01"` to indicate that `pci_b` or `pcit1` implements `intan`, or `H"00"` to indicate that `pci_b` or `pcit1` will not implement an interrupt request. In this case, `intan` is stuck at V_{CC} and the local input signal `l_irqn` is not required. By default, `pci_b` and `pcit1` implement `intan`. See Table 21.

Table 21. 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 applies to the `pci_b` function only. It is an 8-bit read-only register that defines the length of time the `pci_b` 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 22.

Table 22. 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 applies to the `pci_b` function only. It is an 8-bit read-only register that defines the frequency in which the `pci_b` or `pcit1` function would like to gain access to the PCI bus. See Table 23. Designers can set this register with the parameter `MAX_LAT`.

Table 23. 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_b` function in target mode and for the `pcit1` function. The MegaCore functions support the following target transaction types:

- Memory single-cycle target read
- Memory burst target read
- Interrupt acknowledge
- Configuration target read
- Memory single-cycle target write
- Memory burst target write
- Configuration target write
- I/O read
- I/O write

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. The MegaCore function latches the address and command signals on the first clock edge when `framen` is asserted and starts the address decode phase. The MegaCore functions implement slow decode, i.e., the `devseln` signal is asserted three clock cycles after a valid address is presented on the PCI bus. In all operations except configuration read/write and interrupt acknowledge, one of the `bar_hit[5..0]` or `exp_rom_hit` signals is driven high, indicating the BAR range address of the current transaction or the expansion ROM BAR range.

For configuration transactions, the MegaCore function has complete control over the transaction 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 write without interaction from the local-side device.

The `pci_b` and `pcit1` MegaCore functions support the PCI interrupt acknowledge command. This command is treated like a memory read, except during the address phase in which the `ad[31..0]` bus does not contain a valid address, but is driven with stable data. Moreover, these functions do not assert the `bar_hit` or `exp_rom_hit` signals. The local side must implement any special requirements required by the interrupt acknowledge command to respond properly to the interrupt controller.

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 84 describes how to initiate the different types of termination.

☞ The MegaCore functions treat the memory read line and memory read multiple commands as memory read. Similarly, the functions treat 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 transactions. Therefore, the MegaCore function handles them like single-cycle memory commands. Any of the six BARs in the `pci_b` or `pcit1` 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.

Target Read Transactions

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

- Single-cycle read
- Burst read
- Configuration read
- Interrupt acknowledge

For all three 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 drives the address and command on `ad[31..0]` and `cben[3..0]`, correspondingly.

2. Turn-around cycles on the `ad[31..0]` bus occur during the clock immediately following the address phase. During the turn around cycles, the PCI master tri-states the `ad[31..0]` bus but drives correct byte enables on `cben[3..0]` for the first data phase. This process is necessary because the PCI agent driving the `ad[31..0]` bus changes during read cycles.

3. The `pci_b` or `pcit1` function drives `ad[31..0]` with data, but `trdyn` is not asserted.

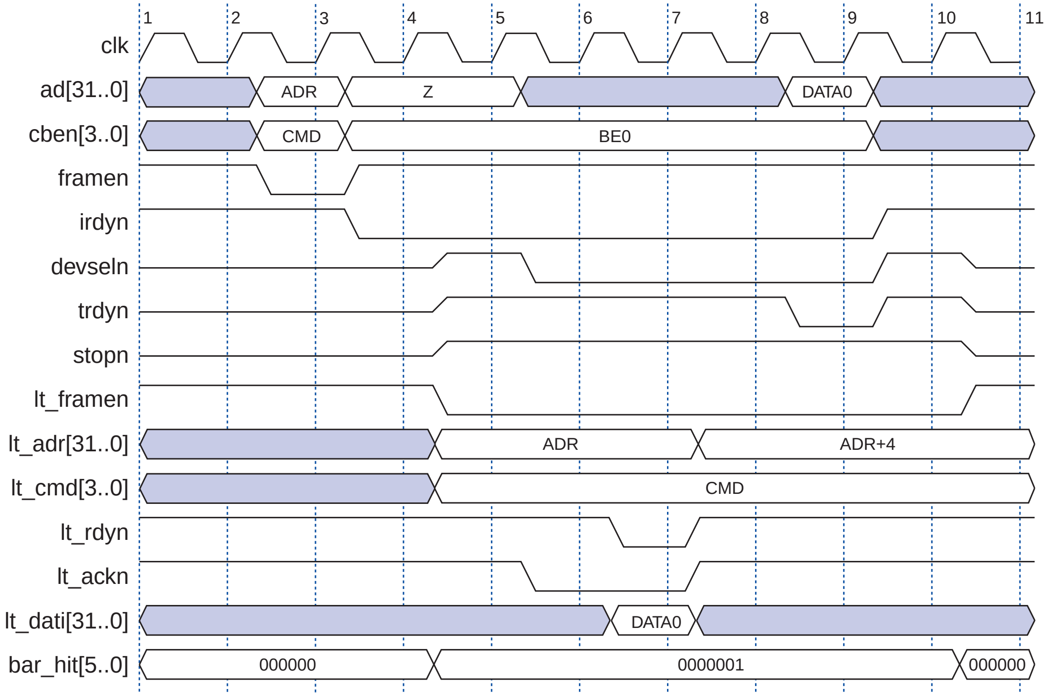
4. One or more data phases follow next, depending on the type of read transaction.

Single-Cycle Read Transaction

Figure 1 shows the waveform for a single-cycle target read transaction. This waveform applies to both single-cycle memory read and I/O read commands. However, if the address specifies a location in the expansion ROM, the `exp_rom_hit` signal is asserted instead of a `bar_hit` signal.

Because `pci_b` and `pcit1` treat the interrupt acknowledge command as a memory read, this waveform also applies to the interrupt acknowledge transaction, except the `bar_hit` and `exp_rom_hit` signals are not asserted.

Figure 1. Single-Cycle Target Read Transaction



The event sequence is summarized in Table 24.

Table 24. Single-Cycle Target Read Sequence

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 `framen` 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 uses clock 3 to decode the address, and if the address falls in the range of one of its BARs, it is treated as an address hit.
4	If the MegaCore function detects an address hit in clock 3, several actions 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 `bar_hit` or `exp_rom_hit` signals and `lt_framen`. During an interrupt acknowledge command, the `bar_hit` and `exp_rom_hit` signals are not asserted. ■ The MegaCore function drives the command on `lt_cmd[3..0]` and address on `lt_adr[31..0]`. ■ The MegaCore function turns on the drivers of `devseln`, `trdyn`, and `stopn`, getting ready to assert `devseln` in clock 5. ■ The PCI master tri-states the `ad[31..0]` bus for the turn-around cycle.
5	The MegaCore function asserts `devseln` 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 `lt_dati[31..0]`. The MegaCore function also enables the output drivers of the `ad[31..0]` bus to ensure that it is not tri-stated for a long time while waiting for valid data.
6	`lt_rdyn` is asserted, indicating that valid data is available on `lt_dati[31..0]`. Because the MegaCore function asserts `lt_ackn` at the same time, indicating that it is ready to receive data from `lt_dati[31..0]`, the MegaCore function registers the data into its internal pipeline.
7	The rising edge of clock 7 registers the valid data from `lt_dati[31..0]`.
8	The MegaCore function drives the valid data that was registered on the rising edge of clock 7. At the same time, the MegaCore function asserts `trdyn`, indicating to the master device that valid data is available on the `ad[31..0]` bus.
9	The MegaCore function deasserts `trdyn` and `devseln` to end the transaction. To satisfy the requirements for sustained tri-state buffers, the MegaCore function drives `devseln`, `trdyn`, and `stopn` high during this clock cycle. Additionally, the MegaCore function tri-states the `ad[31..0]` bus because the cycle is complete.
10	The MegaCore function informs the local-side device that the transaction is complete by deasserting `lt_framen` and resetting the `bar_hit` signals. Additionally, the MegaCore function tri-states `devseln`, `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 clocks to complete. Therefore, the local-side device cannot use more than 12 clocks from `lt_framen` to provide the first data, and no longer than 8 clocks for each subsequent data transfer.

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 and shows the assertion of wait states both by the local side and by the PCI master.

Figure 2. Target Burst Memory Read Transaction

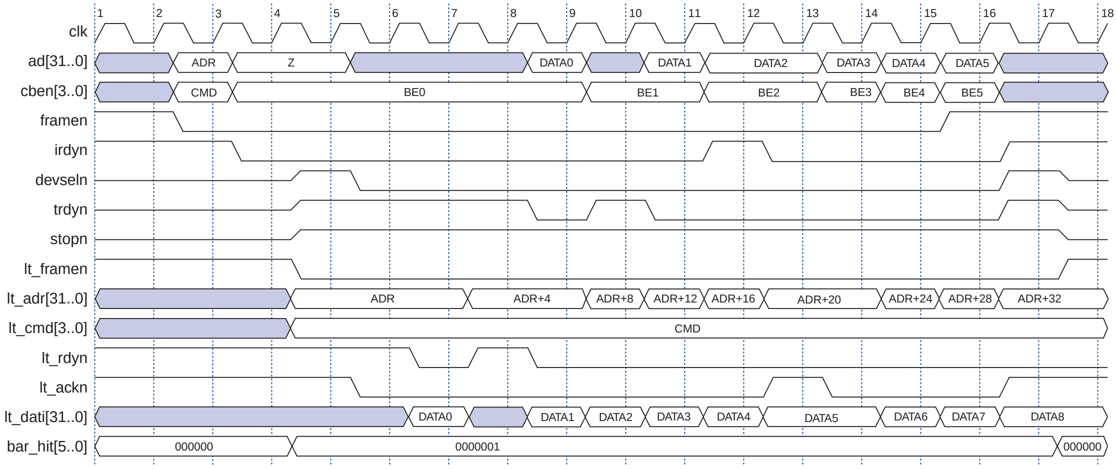


Table 25 describes some of the events during the transaction.

Table 25. Burst Read Events

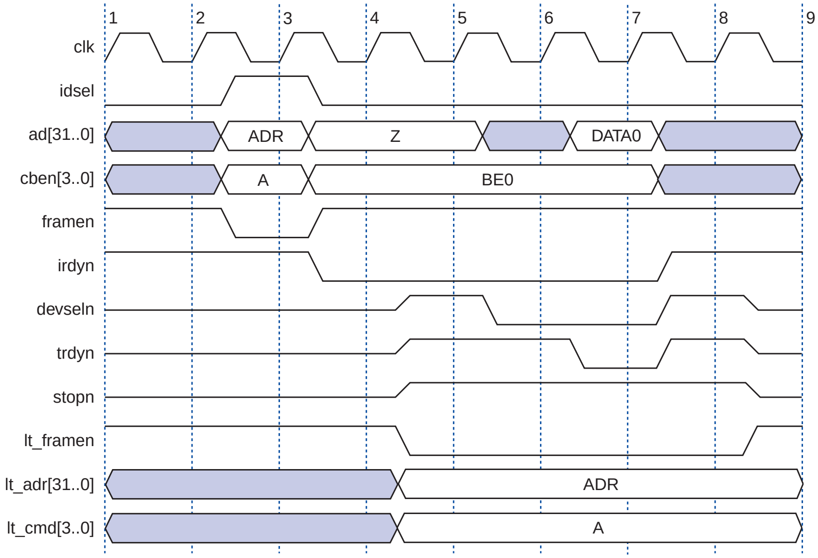
Clock Cycle	Event
6	The local side asserts `lt_rdyn` and the MegaCore function asserts `lt_ackn` indicating a successful data transfer from the local side to the MegaCore function.
7	On the rising edge of clock 7, `DATA0` is registered in the first stage of the MegaCore function internal pipeline. The following events occur during clock 7: ■ The MegaCore function increments `lt_adr[31..0]` by 4 to point to the address of the next word to be transferred. ■ The local side deasserts `lt_rdyn,` indicating that it is not ready to transfer data to the MegaCore function. ■ The MegaCore function asserts `lt_ackn`, indicating that it is ready to receive data from the local-side application.
8	On the rising edge of clock 8, `DATA0` is registered from the first stage of the MegaCore function internal pipeline to the second stage. The following events occur during clock 8: ■ The MegaCore function drives `DATA0` on the PCI bus. Because the internal pipeline has two stages, data requires two clocks to be driven on the PCI side from the local side. ■ The MegaCore function asserts `trdyn` to indicate that valid data is available on the `ad[31..0]` bus. ■ `DATA1` is transferred on the local side because both `lt_rdyn` and `lt_ackn` are asserted. ■ The `lt_adr[31..0]` bus is incremented by 4 to indicate the address of the next word.
9	The MegaCore function deasserts `trdyn` because of the wait state asserted by the local side during clock 7. At the same time, a new word is transferred on the local side and the address on `lt_adr[31..0]` bus is incremented.
10	The MegaCore function asserts `trdyn` and transfers the second word from the local side (i.e., `DATA1`). At the same time, the local side transfers additional data and the MegaCore function increments the address on `lt_adr[31..0]`.
11	The master deasserts `irdyn` to indicate that it is not ready to receive data. At the same time, the MegaCore function continues to receive data from the local side.
12	The wait state asserted by the master during clock 11 is shown by the MegaCore function on the local side by deasserting `lt_ackn` during the same clock. At the same time, the same data that is driven on the PCI bus by the MegaCore function is driven during this clock.
13 to 16	The cycle continues in a similar fashion as described above. The cycle is terminated normally by the master in clock 15 with `framen` going high and `irdyn` going low. The last data is transferred on the rising edge of clock 16.

For a burst-read transaction, the MegaCore function can sustain a maximum of 132 Mbytes per second transfer because it does not impose wait-state requirements. Additionally, the MegaCore function has no upper limit on the size of the burst transfer.

Configuration Read Transaction

Figure 3 shows the timing of a `pci_b` or `pcit1` configuration read transaction. The protocol is identical to the protocol discussed in "Interrupt Acknowledge" on page 73 except for the `idsel` signal, which is active during the address phase of a configuration transaction. Additionally, because the `pci_b` or `pcit1` function does not have to wait for the local side to supply it with data during the configuration read transaction, this transaction requires fewer clock cycles.

Figure 3. Configuration Read Transaction



Target Write Transactions

The `pci_b` and `pcit1` functions support three types of target write transactions:

- Single-cycle write
- Burst target write
- Configuration write

In all target write transactions, the events follow the sequence described below:

1. The address phase occurs when the PCI master asserts `framen` and drives the address and command on `ad[31..0]` and `cben[3..0]`, correspondingly.

2. The MegaCore function decodes the address and determines if the address is within the range of one of its BARs.

3. If the MegaCore function detects a hit, it informs the local side that it will claim the transaction and drive the address and command to the local side.

4. The MegaCore function claims the transaction by asserting `devseln`.

5. The MegaCore function accepts one or more data phases.

Single-Cycle Write Transaction

Figure 4 shows the waveform for a single-cycle target write transaction. This waveform applies to both single-cycle memory write and I/O write commands.

Figure 4. Single-Cycle Target Write Transaction

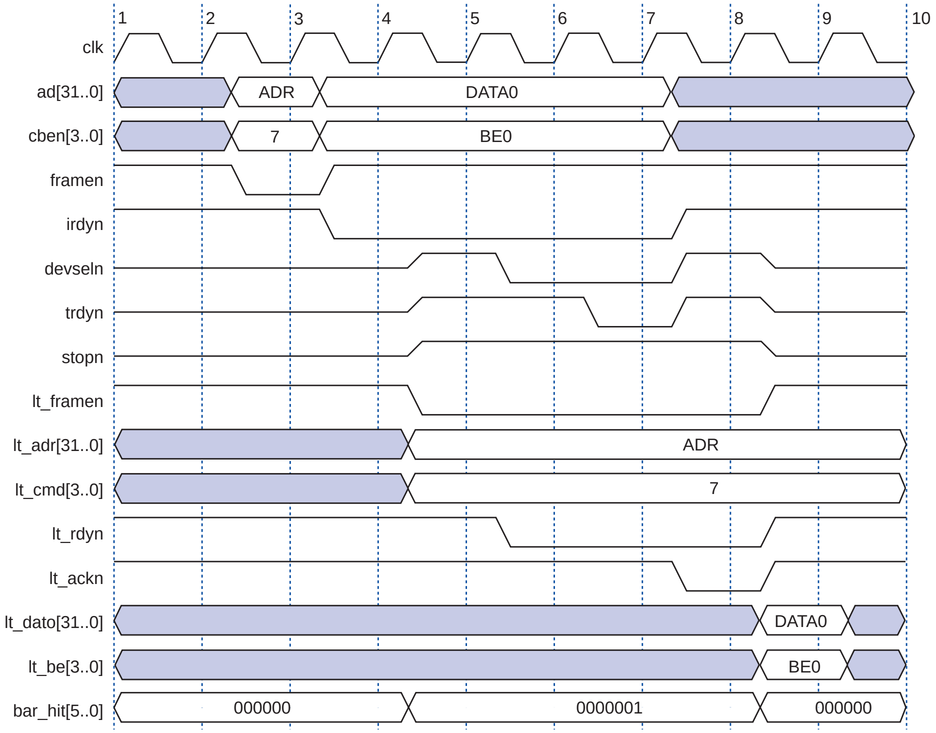


Table 26 describes the events that occur during the transaction.

Table 26. Single-Cycle Target Write Events

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. In clock 3, the master deasserts `framen` and asserts `irdyn` to indicate that only one data phase remains in the transaction. For single-cycle target write transactions, only one data phase occurs during 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, it is treated as an address hit.
4	If the MegaCore function detects an address hit in clock 3, several events follow in clock 4: ■ The MegaCore function informs the local-side device that it will claim the write transaction by asserting one of the `bar_hit` or `exp_rom_hit` signals and `lt_framen`. ■ The MegaCore function drives the command on `lt_cmd[3..0]` and address on `lt_adr[31..0]`. ■ The MegaCore function turns on the drivers of `devseln`, `trdyn`, and `stopn`, getting ready to assert `devseln` in clock 5.
5	The MegaCore function asserts `devseln` to claim the transaction. Figure 4 also shows the local side asserting `lt_rdyn`, indicating that it is ready to receive data from the MegaCore function in clock 6.
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	Data is registered in the MegaCore function internal pipeline on the rising edge of clock 7. Then, the MegaCore function latches the data and byte enables from the PCI bus because both `irdyn` and `trdyn` are asserted. At the same time, the MegaCore function asserts `lt_ackn` to inform the local-side device that valid data will be driven on the `lt_dato[31..0]` bus in clock 8 and the local-side device asserts `lt_rdyn`. Therefore, on the rising edge of clock 9, data is transferred to the local-side device. During a write cycle, data is transferred to the local side on the clock cycle following the one where both `lt_rdyn` and `lt_ackn` are asserted. This process differs from the read transaction in which data is transferred on the same clock that both `lt_rdyn` and `lt_ackn` are asserted.
8	The MegaCore function drives valid data and byte enables on `lt_dato[31..0]` and `lt_ben[3..0]` on clock 8. During the same clock cycle, the MegaCore function deasserts `lt_framen` and the `bar_hit` signal to indicate to the local-side device that the PCI transaction is complete.

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 5 shows the waveform for a typical burst write transaction.

Figure 5. Target Burst Memory Write Transaction

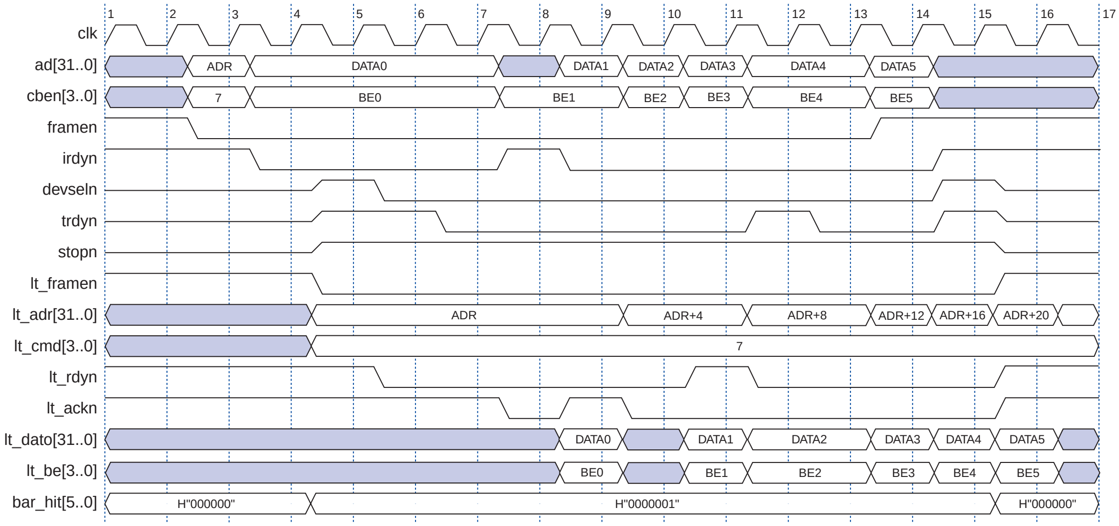


Figure 5 shows the assertion of wait states by the local side and the PCI master. The PCI master inserts a wait state during the second data phase at clock 7 by deasserting `irdyn` for one clock cycle. The `pci_b` or `pcit1` function deasserts `lt_ackn` in clock 8 to indicate the PCI wait state. The local-side device inserts a wait state during the third data transfer by de-asserting `lt_rdyn` during clock 10. The local-side wait state is reflected to the PCI bus with the MegaCore function deasserting `trdyn` during the fifth data phase in clock 11. As also shown in Figure 5, `lt_adr[31..0]` advances by 4 bytes after each successful transfer on the local side.

During burst write transactions, the `lt_rdyn` signal must be asserted before the `trdyn` signal can be asserted.

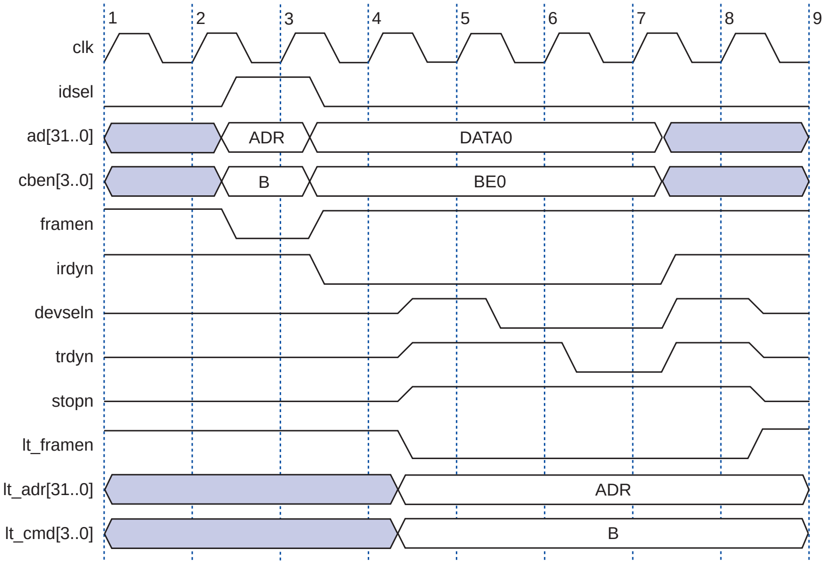
During burst write transactions, the MegaCore function can sustain a maximum of 132 Mbytes per second transfer because it does not impose any wait state requirements. Additionally, the `pci_b` and `pcit1` functions have no upper limit on the size of the burst transfer.

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 terminates 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. Therefore, the local-side device cannot use more than 12 clocks from `lt_framen` to provide the first data, and no longer than 8 clocks for each subsequent data transfer.

Configuration Write Transaction

Figure 6 shows the timing of a `pci_b` or `pcit1` configuration write transaction. The protocol is the same as the protocol discussed in "Single-Cycle Write Transaction" on page 79, except for the `idsel` signal, which is active during the address phase of a configuration transaction.

Figure 6. Configuration Write Transaction



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.

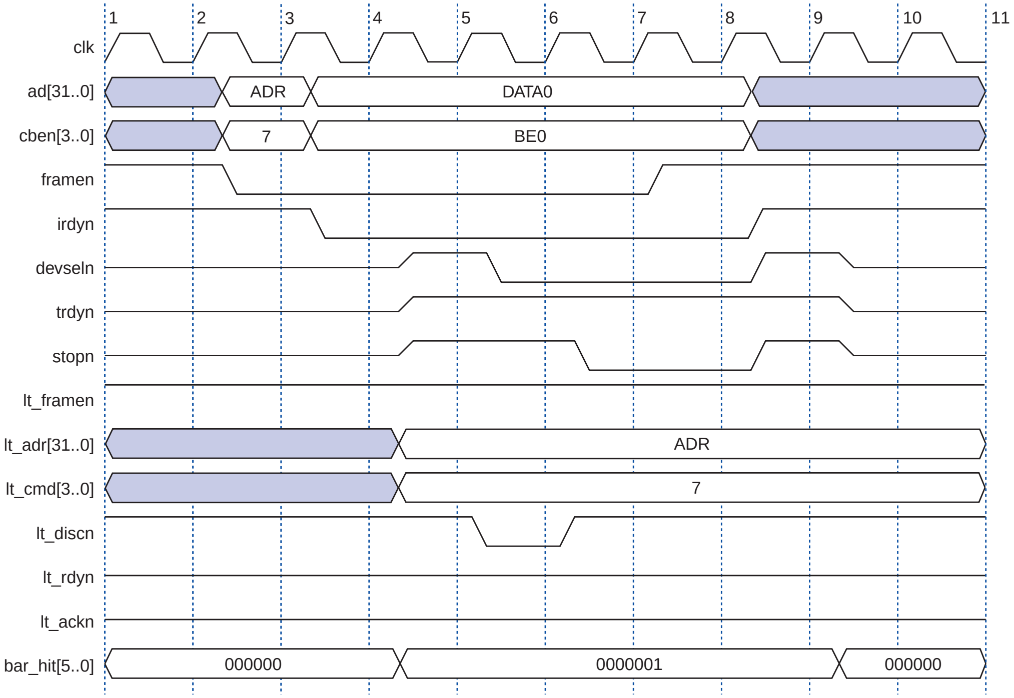
The local-side device can also request a target abort, which indicates that a catastrophic error has occurred in the device. This termination is requested by asserting `lt_abortn` during a target transaction other than a configuration transaction.

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 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_b` or `pcit1` 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 the first DWORD of data has not been received by the `pci_b` or `pcit1` function. Figure 7 shows a write transaction where the MegaCore function issues a retry in response to the local side asserting `lt_discn` during clock 5.

Figure 7. Target Retry

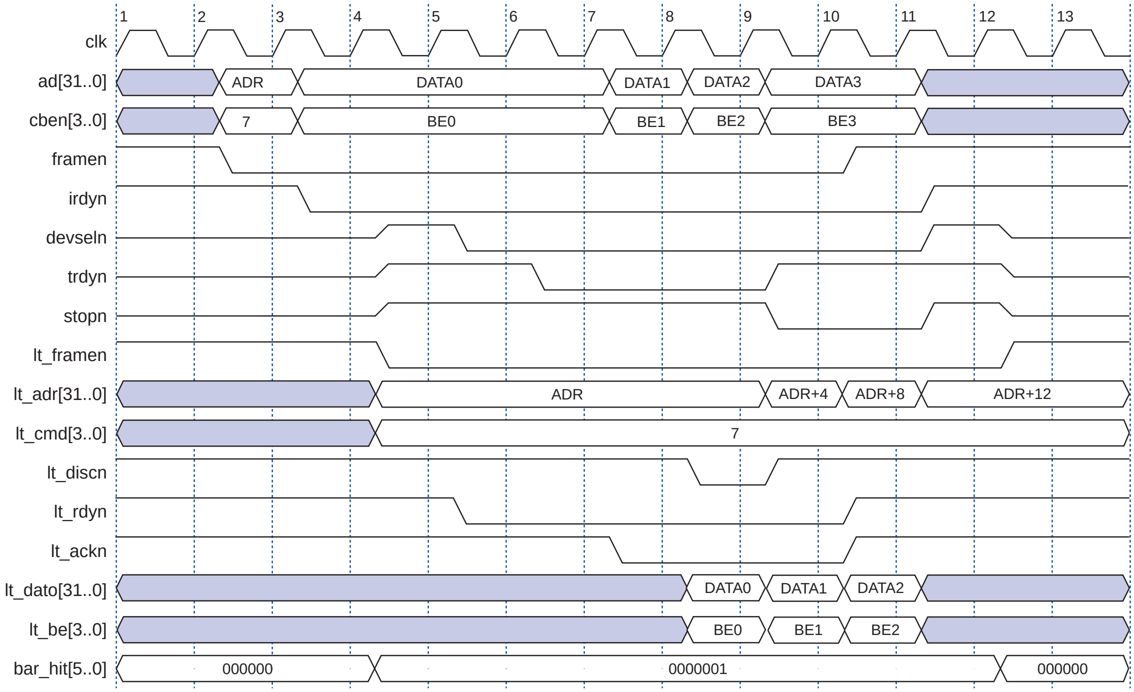


Disconnect

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. Figure 8 shows the MegaCore function issuing a disconnect during a burst write transaction.

☞ The ***PCI Local Bus Specification*** requires that a target device issue 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 address of the current data transfer, and if the transfer exceeds its address range, the local side should request a disconnect by asserting `lt_discn`.

Figure 8. Target Disconnect



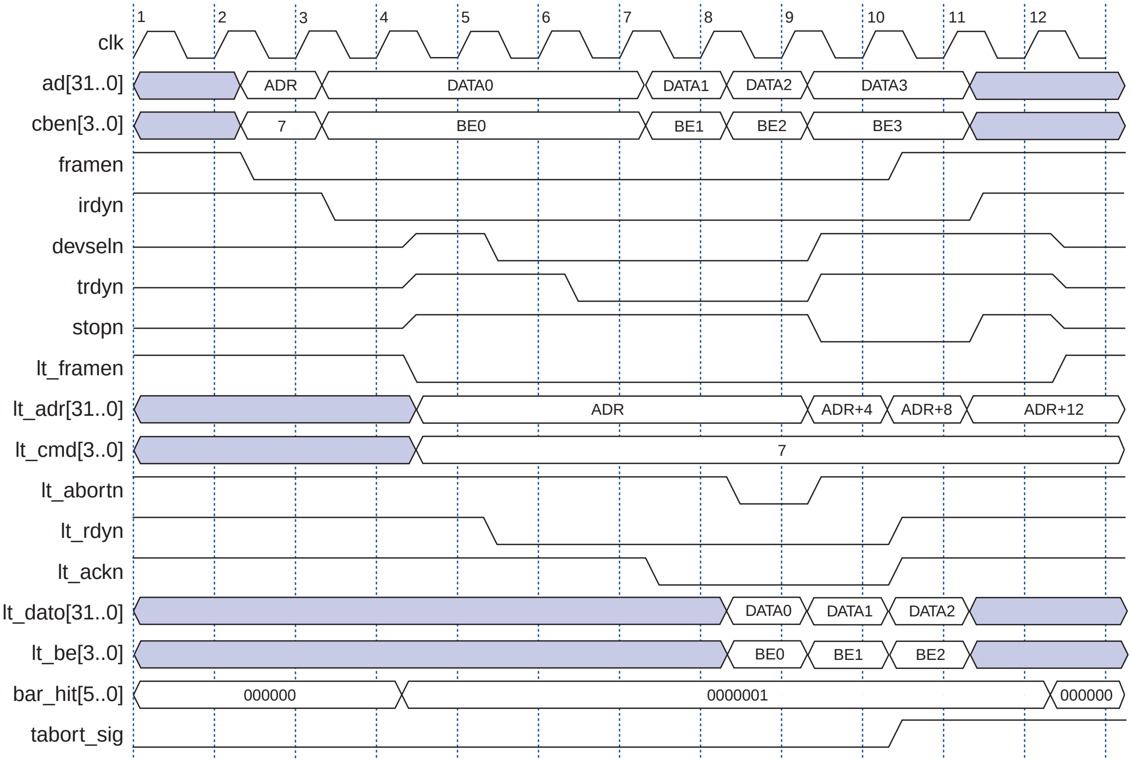
Target Abort

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 requests 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 9 shows the `pci_b` or `pcit1` function issuing an abort during a burst write cycle.

☞ The *PCI Local Bus Specification, Revision 2.2* requires that a target device issue 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 occurs most commonly 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 9. Target Abort



Master Mode Operation

The `pci_b` function supports the following master transaction types (the `pcit1` function supports target transactions only):

- Single-cycle read
- Memory burst read
- Single-cycle write
- Memory burst write

A master operation begins when the local side asserts the `lm_reqn` signal and provides the transaction command on the `lm_cben[3..0]` bus and the PCI address on `lm_adi[31..0]`. The `pci_b` function latches the address and command internally and at the same time asserts `reqn` to request mastership of the PCI bus. When the PCI bus arbiter grants the bus to the `pci_b` function by asserting `gntn`, `pci_b` begins the transaction with the address phase.

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

During a transaction, the `pci_b` function outputs data on `lm_dato[31..0]` and inputs data on `lm_adi[31..0]`. During a single-cycle read/write transaction, the local side provides the `pci_b` function with the byte enable values on `lm_cben[3..0]`. During burst transactions, the local-side application must ensure that `lm_cben[3..0]` is `B"0000""`.

A data transfer between the local side and the `pci_b` function in master mode occurs if `lm_ackn` is asserted, which is different than when the `pci_b` function is in target mode. If the local-side application cannot transfer data, it must assert `lm_busyn`. Asserting `lm_busyn` always results in the `pci_b` function deasserting `lm_ackn` to indicate that data is not being transferred between the `pci_b` function and the local side. The `pci_b` function only deasserts `irdyn` in response to `lm_busyn` if it is necessary. For example, if `lm_busyn` is asserted during a burst read transaction and the target also asserts a wait state, the `pci_b` function does not deassert `irdyn`.

> ☞ The local-side device may require a long time to transfer data to/from the `pci_b` function during a burst transaction. The local-side device must ensure that PCI latency rules are not violated while the `pci_b` function waits for data. Therefore, the local-side device must not insert more than 8 wait states before asserting `lm_busyn`.

The `pci_b` function uses the transaction status register outputs (`lm_tsr[7..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_b` master mode operation.

Master Read Transactions

There are two types of `pci_b` master read transactions: single-cycle read transactions and burst memory read transactions. These transactions differ in the following ways:

- The burst transaction transfers more data and is generally longer.
- The `lm_ben[3..0]` bus can only enable specific bytes in the DWORD during single-cycle transactions.
- The local side uses different processes to assert `lm_lastn`.

Single-Cycle Read Transaction

Figure 10 shows the waveform for a single-cycle master read transaction. This waveform applies to the following transactions generated by the `pci_b` function in master mode:

- I/O read transactions
- Configuration read transactions
- Single-cycle memory read transactions

Figure 10. Single-Cycle Master Read Transaction

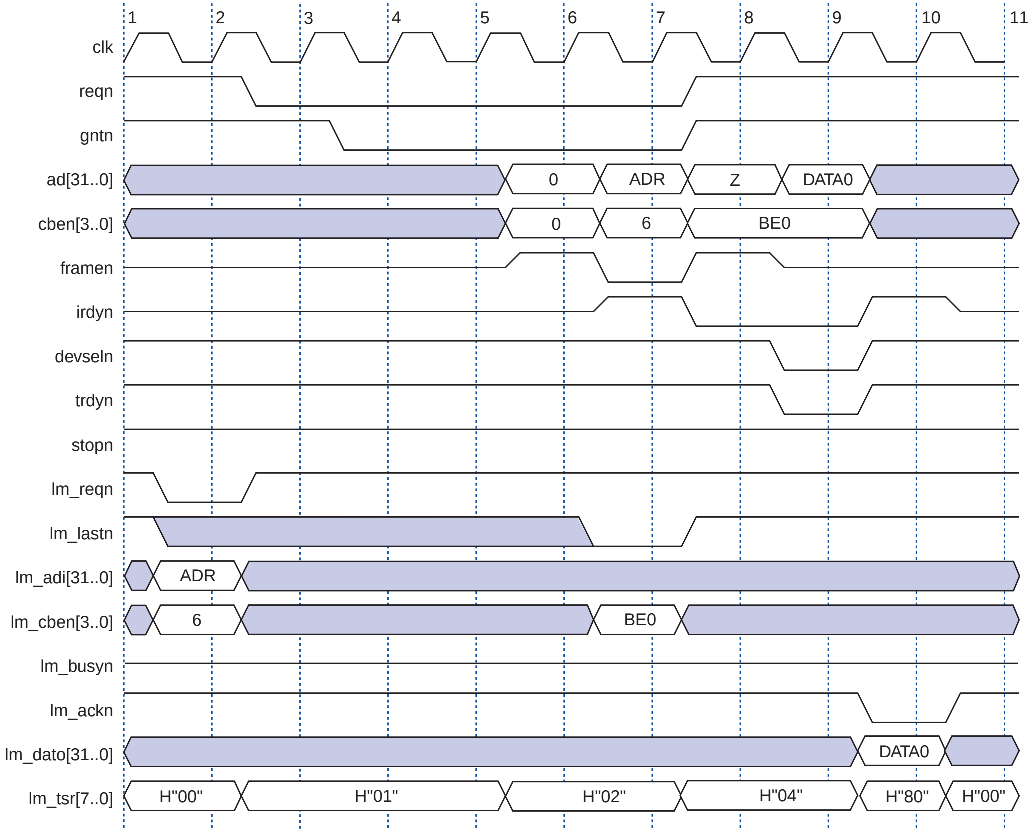


Table 27 shows the sequence of events for a single-cycle master read transaction.

Table 27. Single-Cycle Master Read Transactions (Part 1 of 2)

Clock Cycle	Event
1	The local side asserts `lm_reqn`, drives the address on the `lm_adi[31..0]` bus, and drives the command on `lm_cben[3..0]`. This action informs the `pci_b` function that the local-side application requests a master transaction.
2	The `pci_b` function latches the address and command internally and asserts `reqn` to request mastership of the PCI bus. At the same time, the `pci_b` function asserts `lm_tsr[0]` to indicate to the local side that the `pci_b` master requests the PCI bus.

Table 27. Single-Cycle Master Read Transactions (Part 2 of 2)

Clock Cycle	Event
3	The PCI bus arbiter asserts `gntn` to grant the PCI bus to the `pci_b` function. Although Figure 10 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_b` 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_b` function turns on its output drivers, getting ready to begin the address phase. The `pci_b` function continues to assert its `reqn` signal until it begins the address phase. The `pci_b` function also asserts `lm_tsr[1]` to indicate to the local side that the PCI bus has been granted.
6	The `pci_b` function begins the master read transaction with the address phase. At the same time, `lm_tsr[1]` remains asserted. During this clock cycle, the local side must provide the byte enables for the transaction on `lm_cben[3..0]`. The local side must assert `lm_lastn` during this clock cycle or earlier to ensure that the cycle is a single-cycle read transaction. If `lm_lastn` is not asserted during this clock cycle or earlier and the transaction is a memory transaction, the transaction must have at least two data phases. ☞ In I/O and configuration transactions, the `pci_b` function automatically performs single-cycle transactions and ignores `lm_lastn`. It is sufficient for the local side to assert `lm_lastn` for a single clock cycle on or before clock 6 to ensure that the transaction has only one data phase.
7	The `pci_b` function tri-states the `ad[31..0]` bus for the PCI bus turn-around cycle. Also, the `pci_b` function deasserts `framen` and asserts `irdyn` to inform the target that this data phase is the one in the transaction. Because this phase is the only data phase in the transaction, this action also informs the target that the cycle is a single-cycle transaction. By asserting `irdyn`, the `pci_b` function informs the target that it is ready to receive data. During this clock cycle, the `pci_b` function also asserts `lm_tsr[2]` to inform the local side that it is in data transfer mode. The `pci_b` function asserts `irdyn` on the first data phase of a read transaction, independent of the state of `lm_busyn`.
8	The target claims the transaction by asserting `devseln`. In this case, the target performs a medium address decode. During the same clock cycle, the target asserts `trdyn` to inform the `pci_b` function that it is ready to transfer data. Because the `pci_b` function has already asserted `irdyn`, a data phase is completed on the rising edge of clock 9.
9	The data is output on the local side and the `pci_b` function asserts `lm_ackn` to inform the local side that valid data is available on the `lm_dato[31..0]` bus. The `pci_b` function also asserts `lm_tsr[7]` in the same clock to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock. Because this transaction is single-cycle, the `pci_b` function also deasserts `irdyn` and tri-states the `cben[3..0]` bus for the PCI bus turn-around cycle.
10	The `pci_b` function performs a turn-around cycle for `irdyn` by tri-stating it. The `lm_tsr[7..0]` bus does not show signals are asserted, indicating that the transaction ended normally and the `pci_b` function has no further activity in master mode. Also, the rising edge of clock 10 transfers the data from the `pci_b` function to the local side, deasserting `lm_ackn`, because the `pci_b` function is finished transferring data.

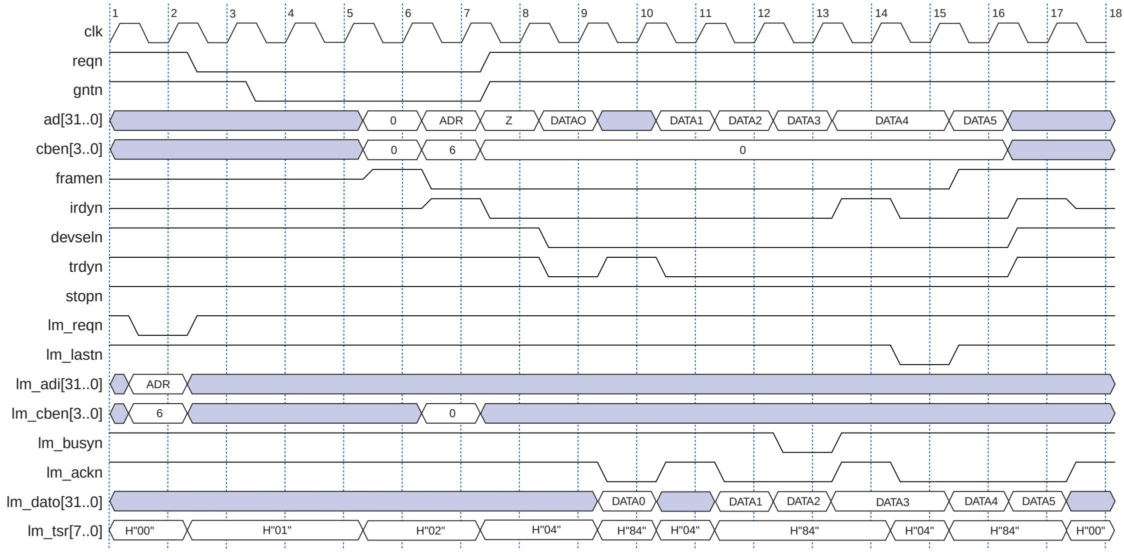
Burst Memory Read Transaction

Figure 11 shows the waveform for a master burst memory read transaction. This waveform applies to the following transactions generated by the `pci_b` function in master mode:

- Memory burst-read transaction
- Memory read multiple transaction
- Memory read line transaction

The `pci_b` 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 11. Master Burst Memory Read Transaction



The sequence of events in Figure 11 is the same as Figure 10. However, Figure 11 has more than one data phase, and wait states exist on the local side as well as on the PCI master side.

In Figure 11, the PCI target asserts a wait state during clock 9. During clock 10, the local side reflects that wait state by deasserting `lm_ackn` and informing the local side that it does not have valid data on the `lm_dato[31..0]` bus.

The local side asserts `lm_busyn` during clock 12, indicating to the `pci_b` function that the local side cannot receive data in clock 13. In response, the `pci_b` function deasserts `irdyn` on the PCI side to inform the PCI target that it is not ready to receive data. Additionally, in clock 13 the `pci_b` function deasserts `lm_ackn` to inform the local side that a data transfer did not take place.

☞ In a burst read transaction, the `pci_b` function asserts wait states on the PCI bus in response to local-side wait states only when necessary. Additionally, the `pci_b` function asserts wait states on the local side in response to PCI target wait states only when necessary.

The local side asserts `lm_lastn` during clock 14. This assertion guarantees to the local side that two more data phases will occur, at most: one during clock 14 and another during clock 15. In Figure 11, the last data phase takes place during clock 15. If `irdyn` was deasserted during clock 15, only one additional data phase takes place after `lm_lastn` is asserted.

☞ It is sufficient for the local side to assert `lm_lastn` for one clock cycle to end the transaction. Asserting `lm_lastn` for more than one clock cycle has no effect on the `pci_b` master interface.

Master Write Transactions

The `pci_b` function has two types of master write transactions: single-cycle write transactions and burst memory write transactions. These transactions differ in the following ways:

- The burst transaction transfers more data and is generally longer.
- The `lm_ben[3..0]` bus can only be used to enable specific bytes in the DWORD in single-cycle transactions.
- The local side asserts `lm_lastn` differently in each transaction.

Single-Cycle Write Transaction

Figure 12 shows a single-cycle master write transaction. This waveform applies to the following transactions generated by `pci_b` in master mode:

- I/O write transactions
- Configuration write transactions
- Single-cycle memory write transactions

Figure 12. Single-Cycle Master Write Transaction

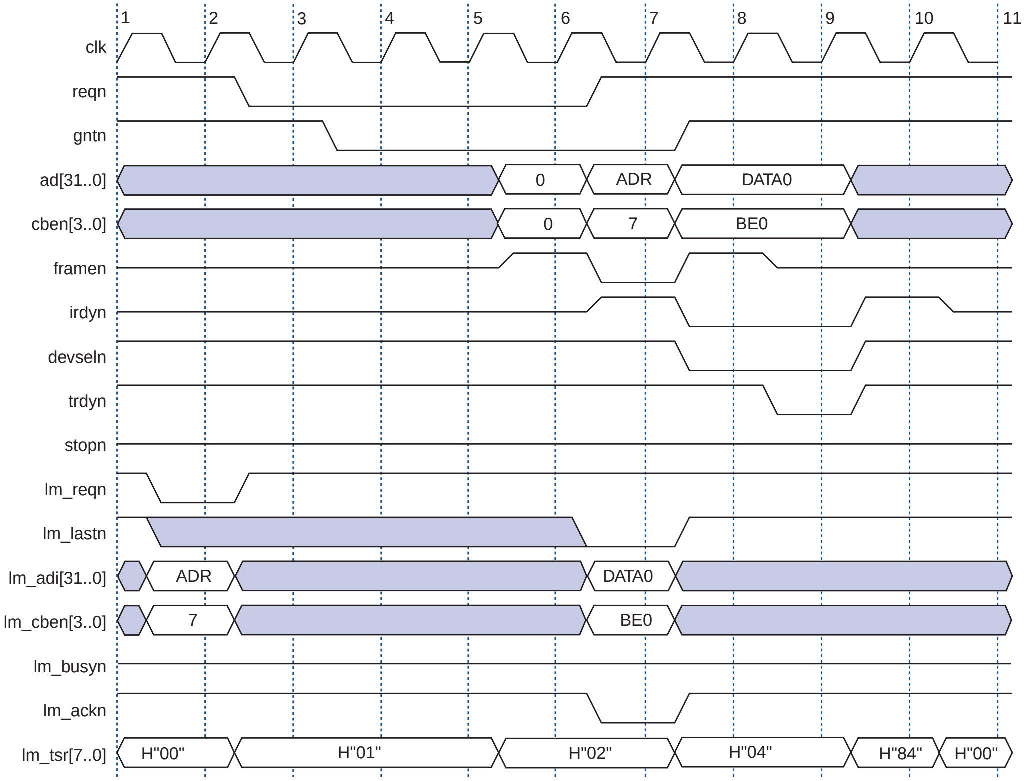


Table 28 shows the sequence of events for the single-cycle master write transaction.

Table 28. Single-Cycle Master Write Transaction Events

Clock Cycle	Event
1	The local side asserts `lm_reqn`, drives the address on the `lm_adi[31..0]` bus, and drives the command on `lm_cben[3..0]`. This action informs the `pci_b` function that the local-side application requests a master transaction.
2	The `pci_b` function latches the address and command internally, and asserts `reqn` to request the PCI bus. At the same time, the `pci_b` function asserts `lm_tsr[0]` to indicate to the local side that the `pci_b` master requests the PCI bus.
3	The PCI bus arbiter asserts `gntn` to grant the PCI bus to the `pci_b` function. Although Figure 12 shows that the grant occurs immediately and the PCI bus is idle at the time, this situation may not apply in an actual transaction on the PCI bus. The `pci_b` function waits until `gntn` is asserted and the PCI bus is idle before it proceeds. A PCI bus idle state occurs when both `framen` and `irdyn` are deasserted.
5	The `pci_b` function turns its output drivers on and begins the address phase. The `pci_b` function continues asserting its `reqn` signal until the function enters the address phase. The `pci_b` function asserts `lm_tsr[1]` to inform the local side that the PCI bus has been granted.
6	The `pci_b` function begins the master write transaction with the address phase. At the same time, `lm_tsr[1]` remains asserted. During this clock cycle, the local side must provide the byte enables for the transaction on `lm_cben[3..0]`. The local side must also assert `lm_lastn` during this clock cycle or earlier to inform the `pci_b` function that there is only one data phase in this transaction. This situation exists because the local side did not transfer any data prior to asserting `lm_lastn`. In I/O and configuration transactions, the `pci_b` function ignores `lm_lastn` and performs single-cycle transactions automatically. It is sufficient for the local side to assert `lm_lastn` for a single clock on or before clock 6 to ensure that the transaction only has one data phase.
7	The `pci_b` function deasserts `framen` and asserts `irdyn` to inform the target that this data phase is the last one in the transaction and valid data exists on the `ad[31..0]` bus. The `pci_b` function asserts `lm_tsr[2]` to inform the local side that it is in data transfer mode. Additionally, the target claims the transaction by asserting `devseln`.
8	The target asserts `trdyn` to inform the `pci_b` function that it is ready to transfer data. Because the `pci_b` function has already asserted `irdyn`, a data phase is completed on the rising edge of clock 9.
9	The `pci_b` function asserts `lm_tsr[7]` to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock cycle. Because this transaction is single-cycle, the `pci_b` function also deasserts `irdyn` and tri-states the `cben[3..0]` and `ad[31..0]` buses for the PCI bus turn-around cycle.
10	The `pci_b` function performs a turn-around cycle for `irdyn` by tri-stating it. The `lm_tsr[7..0]` bus does not show asserted signals, indicating that the transaction ended normally and the `pci_b` function has completed its actions in master mode.

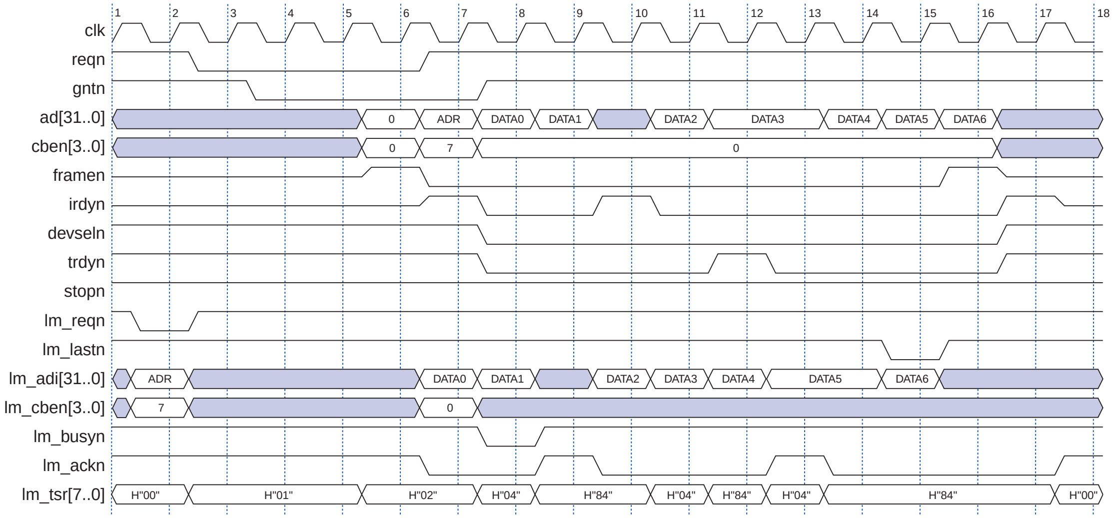
Burst Memory Write Transaction

Figure 13 shows the waveform for a master burst memory write transaction. This waveform applies to the following transactions generated by the `pci_b` function in master mode:

- Memory burst read transaction
- Memory write and invalidate transaction

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

Figure 13. Master Burst Memory Write Transaction



The sequence of events in Figure 13 is the same as in Figure 12. However, in Figure 13 more than one data phase is shown and wait states are shown on the local side as well as on the PCI side. Table 29 shows additional events for the burst memory write transaction.

Table 29. Master Burst Memory Write Transaction Events

Clock Cycle	Event
7	The local side asserts `lm_busyn` during clock 7. This action indicates to the `pci_b` function that the local side is unable to transfer data in clock 8.
8	The `pci_b` function deasserts `lm_ackn` during clock 8 to inform the local side that it will not transfer data on the rising edge of clock 9.
9	The `pci_b` function deasserts `irdyn` during clock 9 because it does not have valid data to transfer to the PCI target.
11	The PCI target asserts a wait state during clock 11.
12	The `pci_b` function deasserts `lm_ackn` during clock 12 to inform the local side that it is unable to receive data.
14	The local side asserts `lm_lastn` during clock 14 at the same time that DATA6 is transferred from the local side to the `pci_b` function.
15	The `pci_b` function signals the last data phase in clock 15.
16	The last data phase ends when the `pci_b` function transfers DATA6 on the rising edge of clock 16.

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_b` 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_b` function adheres to this rule; it ends the transaction when the latency timer expires and `gntn` is no longer asserted. In that case, `pci_b` asserts `lm_tsr[3]` (`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_b` function detects a retry condition (see "Retry" on page 84 for details), it ends the cycle and asserts `lm_tsr[4]` 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 repeat 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 85. When the `pci_b` function ends the transaction because of a disconnect without data, it asserts `lm_tsr[5] (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 85. When the `pci_b` function ends the transaction because of a disconnect with data, it asserts `lm_tsr[6]` (`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 86. When the `pci_b` function ends the transaction because of a target abort, it asserts the `targ_abort_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_b` 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_b` 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.

Notes:

Examples

Contents

June 1999

Notes:

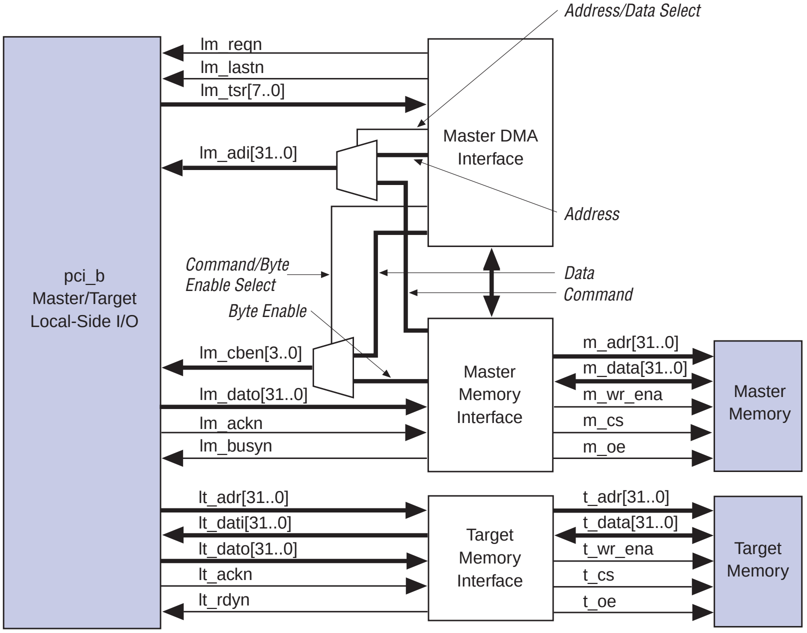
June 1999, ver. 2

The applications for the PCI bus interface have increased as the demand for high-performance and high-bandwidth I/O functions has increased. This section describes two typical designs that implement a PCI interface using the Altera® `pci_b` and `pcit1` MegaCore™ functions.

Example 1: Unintelligent Local Side

The first example shows the `pci_b` function connecting with an unintelligent local side. The example also describes the signals that are used to communicate between the different design blocks. See Figure 1.

Figure 1. Interface Logic to an Unintelligent Local Side



Master DMA Interface

The master DMA interface generates the control signals to initiate a PCI transaction and monitors the progress of the transaction. The master DMA interface asserts `lm_reqn` when the local side requests ownership of the PCI bus and asserts `lm_lastn` when the local side requests the `pci_b` master function to end the current transaction. The master DMA interface also drives `lm_adi[31..0]` and `lm_cben[3..0]` during the address phase. The master DMA interface monitors `lm_tsr[7..0]` to identify the status of the transaction.

Master Memory Interface

The master memory interface buffers the data transfer between the `pci_b` master function and the master memory. The interface monitors `lm_ackn` to determine if a local side data transfer is complete and asserts `lm_busyn` when the local side is unable to transfer data.

Target Memory Interface

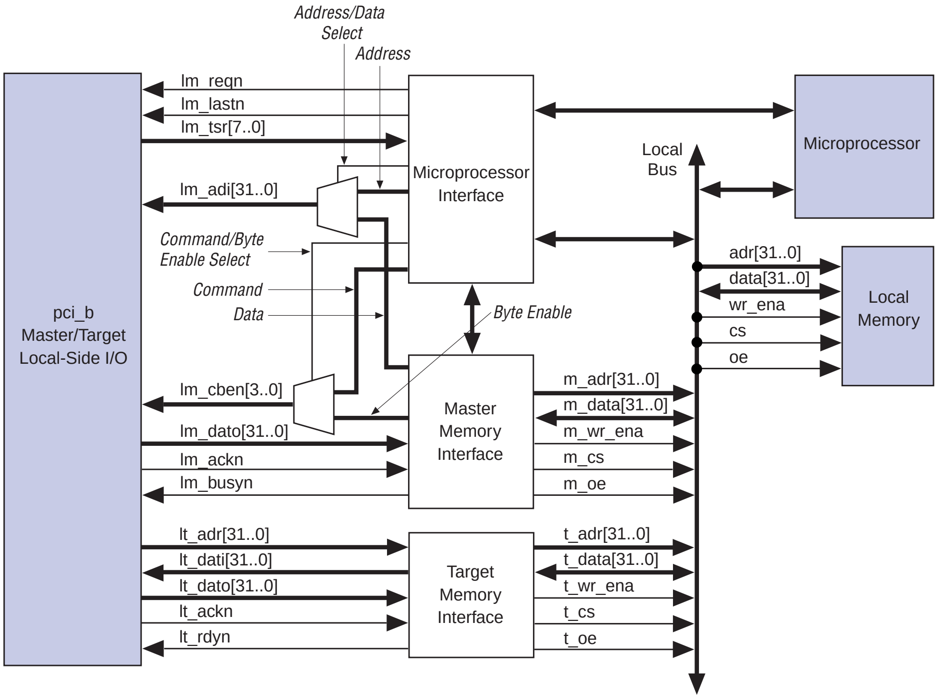
The target memory interface buffers the data transfer between the `pci_b` target function and the target memory. The interface asserts `lt_rdyn` to indicate valid data during a target read, or to indicate it is ready to accept data during a target write. It also monitors `lt_ackn` to acknowledge when the `pci_b` target function has driven valid data out during a target write, or when the `pci_b` target function is ready to accept data during a target read.

The Altera `pcit1` MegaCore function operates in the same manner as the `pci_b` target function.

Example 2: Intelligent Host

The next example shows the `pci_b` function connecting with an intelligent host using a local bus. The example also describes the signals that communicate between the different design blocks. See Figure 2.

Figure 2. Interface Logic to a Local-Side Microprocessor



Microprocessor Interface

The microprocessor interface generates the control signals to initiate a PCI transaction or a local bus transaction, and monitors the transaction's progress. The microprocessor interface also arbitrates ownership of the local bus and verifies whether the `pci_b` function is performing a transaction on the local bus. If the microprocessor interface wants ownership of the PCI bus, it asserts `lm_reqn` and sends the request to the `pci_b` master function. After the microprocessor interface has been granted ownership of the PCI bus or the local bus, it generates the control signals to begin the read/write transactions.

Master Memory Interface

The master memory interface controls the data transfer between the `pci_b` master function and the local memory. The master memory interface monitors `lm_ackn` to determine whether a local-side data transfer is complete and asserts `lm_busyn` when the local side is unable to transfer data.

Target Memory Interface

The target memory interface controls the data transfer between the `pci_b` target function and the local memory. The target memory interface asserts `lt_rdyn` to indicate valid data during a target read, or that it is ready to accept data during a target write. In addition, the interface monitors `lt_ackn` to acknowledge that the `pci_b` target function has driven valid data out during a target write, or to indicate when the `pci_b` target function is ready to accept data during a target read.

The Altera `pcit1` MegaCore function operates in a manner similar to the `pci_b` target function.

PCI SIG Protocol Checklists

Contents

June 1999

Notes:

PCI SIG Protocol Checklists

June 1999, ver. 2

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_b` or `pcit1` function meets the requirement. Checklists not applicable to the Altera `pci_b` or `pcit1` functions are not listed, and table entries annotated with N.A. represent non-applicable PCI SIG requirements.

Table 1. Component Configuration

CO#	Requirement	pci_b		pcit1	
		Yes	No	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.		N.A.	

Table 2. Component Configuration Space Summary

Location	Name	Required/Optional	pci_b		pcit1	
			N/A	Support	N/A	Support
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	Base address registers	One or more required for any address location.		✓		✓
28h-2Bh	Cardbus CIS pointer	Optional.		✓		✓
2Ch-2Dh	Subsystem vendor ID	Optional.		✓		✓
2Eh-2Fh	Subsystem ID	Optional.		✓		✓
30h-33h	Expansion ROM base address.	Required for devices/functions that have expansion ROM.		✓		✓
34h	Capabilities pointer	Optional.		✓		✓
35h-3Bh	Reserved.			✓		✓
3Ch	Interrupt line	Required by devices/functions that use an interrupt pin.		✓		✓
3Dh	Interrupt pin	Required by devices/functions that use an interrupt pin.		✓		✓
3Eh	Min_Gnt	Optional.		✓	✓	
3Fh	Max_Lat	Optional		✓	✓	

Table 3. Device Control summary

DC#	Required/Optional	pci_b		pcit1	
		Yes	No	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_b		pcit1	
			Yes	No	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 palette snoop	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_b		pcit1	
		Yes	No	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_b		pcit1	
			Yes	No	Yes	No
0-3	Reserved	Required.				
4	Capabilities list	Required for devices/functions that support the capabilities list.	✓		✓	
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.	✓		N.A.	
13	Received master abort	Required.	✓		N.A.	
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_b	
		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)	✓	
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 never changes `framen` until the current data phase completes. (section 3.2.1)	✓	
7	Once the IUT asserts `irdyn`, it never changes `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_b	
		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) *(1)*		✓
25	IUT always releases LOCK# when access is terminated by target-abort or master-abort. (section 3.6) *(1)*		✓
26	IUT always deasserts LOCK# for a minimum of one idle cycle between consecutive lock operations. (section 3.6) *(1)*		✓
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) *(2)*		✓
33	IUT never generates a dual cycle when the upper 32-bits of the address are zero. (section 3.10.1) *(2)*		✓

Notes:

(1) The lock function is not supported.

(2) The dual address command is not supported.

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

TP#	Requirement	pci_b		pcit1	
		Yes	No	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)	✓		✓	
4	32-bit addressable IUT treats the dual command as reserved. (section 3.1.1)	✓		✓	
5	Once IUT has asserted `trdyn`, it never changes `trdyn` until the data phase completes. (section 3.2.1)	✓		✓	
6	Once IUT has asserted `trdyn`, it never changes `devseln` until the data phase completes. (section 3.2.1)	✓		✓	
7	Once IUT has asserted `trdyn`, it never changes `stopn` until the data phase completes. (section 3.2.1)	✓		✓	
8	Once IUT has asserted `stopn`, it never changes `stopn` until the data phase completes. (section 3.2.1)	✓		✓	
9	Once IUT has asserted `stopn`, it never changes `trdyn` until the data phase completes. (section 3.2.1)	✓		✓	
10	Once IUT has asserted `stopn`, it never changes `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)	✓		✓	
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)	✓		✓	
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)	✓		✓	

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

TP#	Requirement	pci_b		pcit1	
		Yes	No	Yes	No
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)	✓		✓	
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 never deasserts `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)	✓		✓	
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) *(2)*	N.A.		N.A.	
28	IUT always issues `devseln` before any other response. (section 3.7.1)	✓		✓	
29	Once IUT has asserted `devseln`, it never deasserts `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 2.2*. A check mark in the Yes column indicates that the `pci_b` or `pcit1` function meets the requirement. Checklist items that are not applicable are indicated with N.A.

☞ Refer to the **readme** files in the **\sim\sig** directory of each MegaCore function for the descriptions of the Simulator Channel Files (**.scf**) that correspond to the PCI SIG test scenarios.

Table 9. Test Scenario: 1.1 PCI Device Speed (as indicated by devsel) Tests

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

Table 10. Test Scenario: 1.2 PCI Bus Target Abort Cycles (Part 1 of 2)

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

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

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

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

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

Table 12. Test Scenario: 1.4 PCI Bus Single Data Phase Disconnect Cycles

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

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

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

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

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

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

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

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

#	Requirement	pci_b	
		Yes	No
22	Data transfer after read from slow configuration slave.	✓	
23	Data transfer after write to subtractive configuration slave.	✓	
24	Data transfer after read from subtractive configuration slave.	✓	
25	Data transfer after memory read multiple from fast slave.	✓	
26	Data transfer after memory read multiple from medium slave.	✓	
27	Data transfer after memory read multiple from slow slave.	✓	
28	Data transfer after memory read multiple from subtractive slave.	✓	
29	Data transfer after memory read line from fast slave.	✓	
30	Data transfer after memory read line from medium slave.	✓	
31	Data transfer after memory read line from slow slave.	✓	
32	Data transfer after memory read line from subtractive slave.	✓	
33	Data transfer after memory write and invalidate to fast slave.	✓	
34	Data transfer after memory write and invalidate to medium slave.	✓	
35	Data transfer after memory write and invalidate to slow slave.	✓	
36	Data transfer after memory write and invalidate to subtractive slave.	✓	

Table 15. Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles (Part 1 of 2)

#	Requirement	pci_b	
		Yes	No
1	Data transfer after write to fast memory slave.	✓	
2	Data transfer after read from fast memory slave.	✓	
3	Data transfer after write to medium memory slave.	✓	
4	Data transfer after read from medium memory slave.	✓	
5	Data transfer after write to slow memory slave.	✓	
6	Data transfer after read from slow memory slave.	✓	
7	Data transfer after write to subtractive memory slave.	✓	
8	Data transfer after read from subtractive memory slave.	✓	
9	Data transfer after write to fast I/O slave.	✓	
10	Data transfer after read from fast I/O slave.	✓	
11	Data transfer after write to medium I/O slave.	✓	
12	Data transfer after read from medium I/O slave.	✓	

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

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

Table 16. Test Scenario: 1.8 Multi-Data Phase & trdyn Cycles (Part 1 of 2)			
#	**Requirement**	**pci_b**	
		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`.	✓	
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`.	✓	

Table 16. Test Scenario: 1.8 Multi-Data Phase & trdyn Cycles (Part 2 of 2)			
#	Requirement	pci_b	
		Yes	No
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`.	✓	
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_b	
		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_b**	
		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. *(2)*		✓
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. *(2)*		✓
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. *(2)*		✓
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.	✓	
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.	✓	
9	Verify that the IUT sets the parity error detected bit when odd parity is detected on an IUT configuration multi-data phase read.	✓	
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_b	
		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.		✓
4	Configuration read transaction terminates before the fourth data phase is completed.		✓
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. *(2)*		✓
8	Dual address read transaction terminates before the fourth data phase is completed. *(2)*		✓
9	Memory write and invalidate terminates on line boundary.	✓	

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

#	Requirement	pci_b	
		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_b	
		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_b		pcit1	
		Yes	No	Yes	No
1	IUT generates interrupts when programmed.	✓		✓	
2	IUT clears interrupts when serviced (may include driver-specific actions).	✓		✓	

Table 23. Test Scenario: 2.2. Target Reception of Special Cycle

#	Requirement	pci_b		pcit1	
		Yes	No	Yes	No
1	The `devsel` signal is not asserted by the IUT after a special cycle.	✓		✓	
2	IUT receives encoded special cycle.		✓		✓

Table 24. Test Scenario: 2.3. Target Detection of Address & Data Parity Error for Special Cycle

#	Requirement	pci_b		pcit1	
		Yes	No	Yes	No
1	IUT reports address parity errors via `serr`.	✓		✓	
2	IUT reports data parity errors via `serr`.		✓		✓
3	IUT keeps `serr` active for at least one clock cycle.	✓		✓	

Table 25. Test Scenario: 2.4. Target Reception of I/O Cycles with Legal & Illegal Byte Enables

#	Requirement	pci_b		pcit1	
		Yes	No	Yes	No
1	IUT asserts `trdy` following second rising edge from `framen` on all legal `BE''`.		✓		✓
2	IUT terminates with target abort for each illegal `BE''`.		✓		✓
3	IUT asserts `stopn`.		✓		✓
4	IUT de-asserts `stopn` after `framen` deassertion.		✓		✓

Table 26. Test Scenario: 2.5. Target Ignores Reserved Commands

#	Requirement	pci_b		pcit1	
		Yes	No	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_b		pcit1	
		Yes	No	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.		✓		✓

Table 28. Test Scenario: 2.7. Target Receives I/O Cycles with Address & Data Parity Errors

#	Requirement	pci_b		pcit1	
		Yes	No	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_b		pcit1	
		Yes	No	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_b		pcit1	
		Yes	No	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.	✓		✓	

Table 31. Test Scenario: 2.10. Target Receives Memory Cycles with Address & Parity Errors

#	Requirement	pci_b		pcit1	
		Yes	No	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_b		pcit1	
		Yes	No	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.		✓		✓
4	IUT responds to a memory write followed by a memory read with a read selecting the IUT.		✓		✓

Table 33. Test Scenario: 2.12. Target Performs Exclusive Address Cycles

#	Requirement	pci_b		pcit1	
		Yes	No	Yes	No
1	IUT responds to exclusive access by initiator and accepts lock. *(1)*		✓		✓
2	IUT responds with a retry when second initiator attempts an access. *(1)*		✓		✓
3	IUT responds to access releasing lock by initiator. *(1)*		✓		✓
4	IUT responds to access by second initiator. *(1)*		✓		✓

Table 34. Test Scenario: 2.13. Target Receives Cycles with irdy Used for Data Stepping

#	Requirement	pci_b		pcit1	
		Yes	No	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 to tables:

(1) The lock function is not supported.
(2) The dual address command is not supported.

Notes: