

PCI Compiler

User Guide



The PCI Compiler is scheduled for product obsolescence and discontinued support as described in [PDN1410](#). Therefore, Altera does not recommend use of this IP in new designs. For more information about Altera's current IP offering, refer to Altera's [Intellectual Property](#) website.



101 Innovation Drive
San Jose, CA 95134
www.altera.com

Compiler Version:
Document Date:

11.1
October 2011





Copyright © 2011 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



Printed on recycled paper





About PCI Compiler

Introduction	1
Release Information	2
Device Family Support	2
Features	3
Common Features	3
PCI Compiler with MegaWizard Plug-in Manager Flow	4
PCI Compiler with SOPC Builder Flow	4
General Description	5
PCI MegaCore Functions	5
PCI Testbench	6
PCI Compiler with MegaWizard Plug-in Manager Flow	6
PCI Compiler With SOPC Builder Flow	7
Selecting the Appropriate Flow for Your Design	9
PCI Compiler With SOPC Builder Flow	9
PCI Compiler With MegaWizard Plug-in Manager Flow	10
Compliance Summary	10
Performance and Resource Utilization	11
PCI Compiler with MegaWizard Plug-in Manager Flow	11
PCI Compiler with SOPC Builder Flow	13
Installation and Licensing	17
OpenCore Plus Evaluation.....	19
OpenCore Plus Time-Out Behavior.....	19

Section I. PCI Compiler With MegaWizard Plug-In Manager Flow

Chapter 1. Getting Started

Design Flow	1-1
PCI MegaCore Function Design Walkthrough	1-2
Create a New Quartus II Project	1-2
Launch IP Toolbench	1-4
Step 1: Parameterize	1-5
Step 2: Set Up Simulation	1-7
Step 3: Generate	1-7
Simulate the Design	1-9
Simulation in the Quartus II Software	1-11
The Quartus II Simulation Files	1-12
Master Simulation Files	1-13
Target Simulation Files	1-15

Compile the Design	1–16
Program a Device	1–18
PCI Timing Support	1–18
Using the Reference Designs	1–19
pci_mt32 MegaCore Function Reference Design	1–19
Synthesis & Compilation Instructions	1–20
pci_mt64 MegaCore Function Reference Design	1–21
synthesis & Compilation Instructions	1–22

Chapter 2. Parameter Settings

Parameterize PCI Compiler	2–1
PCI MegaCore Function Settings	2–1
Read-Only PCI Configuration Registers	2–2
PCI Base Address Registers (BARs)	2–2
Advanced PCI MegaCore Function Features	2–3
Optional Registers	2–3
Optional Interrupt Capabilities	2–4
Master Features	2–4
Variation File Parameters	2–7

Chapter 3. Functional Description

Functional Overview	3–1
Target Device Signals & Signal Assertion	3–6
Master Device Signals & Signal Assertion	3–9
PCI Bus Signals	3–11
Parameterized Configuration Register Signals	3–15
Local Address, Data, Command, & Byte Enable Signals	3–16
Target Local-Side Signals	3–20
Master Local-Side Signals	3–24
PCI Bus Commands	3–27
Configuration Registers	3–28
Vendor ID Register	3–31
Device ID Register	3–31
Command Register	3–32
Status Register	3–33
Revision ID Register	3–34
Class Code Register	3–35
Cache Line Size Register	3–35
Latency Timer Register	3–36
Header Type Register	3–36
Base Address Registers	3–37
CardBus CIS Pointer Register	3–40
Subsystem Vendor ID Register	3–40
Subsystem ID Register	3–41
Expansion ROM Base Address Register	3–41
Capabilities Pointer	3–42
Interrupt Line Register	3–43

Interrupt Pin Register	3-43
Minimum Grant Register	3-43
Maximum Latency Register	3-44
Target Mode Operation	3-44
Target Read Transactions	3-48
Memory Read Transactions	3-48
I/O Read Transactions	3-61
Configuration Read Transactions	3-62
Target Write Transactions	3-63
Memory Write Transactions	3-63
I/O Write Transactions	3-75
Configuration Write Transactions	3-76
Target Transaction Terminations	3-77
Retry	3-77
Disconnect	3-79
Target Abort	3-86
Additional Design Guidelines for Target Transactions	3-88
Master Mode Operation	3-88
PCI Bus Parking	3-92
Design Consideration	3-92
Master Read Transactions	3-93
Memory Read Transactions	3-93
I/O & Configuration Read Transactions	3-107
Master Write Transactions	3-108
Memory Write Transactions	3-108
I/O & Configuration Write Master Transactions	3-124
Abnormal Master Transaction Termination	3-125
Latency Timer Expires	3-125
Retry	3-125
Disconnect Without Data	3-126
Disconnect with Data	3-126
Target Abort	3-126
Master Abort	3-126
Host Bridge Operation	3-127
Using the PCI MegaCore Function as a Host Bridge	3-127
PCI Configuration Read Transaction from the pci_mt64 Local Master Device to the Internal Configuration Space	3-127
PCI Configuration Write Transaction from the pci_mt64 Local Master Device to the Internal Configuration Space	3-129
64-Bit Addressing, Dual Address Cycle (DAC)	3-131
Target Mode Operation	3-131
64-Bit Address, 64-Bit Data Single-Cycle Target Read Transaction	3-132
Master Mode Operation	3-134
64-Bit Address, 64-Bit Data Master Burst Memory Read Transaction	3-134

Chapter 4. Testbench

General Description	4-1
---------------------------	-----

Features	4-2
PCI Testbench Files	4-2
Testbench Specifications	4-6
Master Transactor (mstr_tranx)	4-7
PROCEDURES and TASKS Sections	4-7
INITIALIZATION Section	4-8
USER COMMANDS Section	4-8
Target Transactor (trgt_tranx)	4-12
FILE IO section	4-13
PROCEDURES and TASKS sections	4-13
Bus Monitor (monitor)	4-14
Clock Generator (clk_gen)	4-14
Arbiter (arbiter)	4-15
Pull Up (pull_up)	4-15
Local Reference Design	4-15
Local Target	4-17
DMA Engine	4-17
Local Master	4-19
lm_lastn Generator	4-19
Prefetch	4-19
LPM RAM	4-19
Simulation Flow	4-20

Section II. PCI Compiler With SOPC Builder Flow

Chapter 5. Getting Started

Design Flow	5-1
PCI Compiler with SOPC Builder Flow Design Walkthrough	5-2
Create a New Quartus II Project	5-3
Set Up the PCI-Avalon Bridge	5-5
Add the Remaining Components to the SOPC Builder System	5-7
Complete the Connections in SOPC Builder	5-8
Generate the SOPC Builder System	5-9
Files Generated by SOPC Builder	5-10
Simulate the Design	5-11
Compile the Design	5-13
Program a Device	5-14
Upgrading Systems from a Previous Version	5-15

Chapter 6. Parameter Settings

System Options-1	6-1
PCI Device Mode	6-1
PCI Target Performance	6-3
PCI Master Performance	6-5
Value of Multiple Pending Reads	6-6

System Options-2	6-9
PCI Bus Speed	6-9
PCI Data Bus Width	6-9
PCI Clock/Reset Settings	6-9
PCI Bus Arbiter	6-10
PCI Configuration	6-11
PCI Base Address Registers	6-11
PCI Read-Only Registers	6-11
Setting the PCI Base Address Register Values	6-11
Manual Setting of the BAR Size & Avalon Base Address	6-14
Avalon Configuration	6-16

Chapter 7. Functional Description

Functional Overview	7-1
PCI-Avalon Bridge Blocks	7-2
Avalon-MM Ports	7-3
Control/Status Register Module	7-5
PCI MegaCore Function	7-5
PCI Bus Arbiter	7-6
Other PCI-Avalon Bridge Modules	7-6
PCI Operational Modes	7-6
PCI Target-Only Peripheral Mode Operation	7-6
PCI Master/Target Peripheral Mode Operation	7-8
PCI Host-Bridge Device Mode Operation	7-10
Performance Profiles	7-11
Target Performance	7-12
Master Performance	7-12
Interface Signals	7-13
PCI Bus Arbiter Signals	7-14
PCI Bus Commands	7-15
PCI Target Operation	7-15
Non-Prefetchable Operations	7-17
Non-Prefetchable Write Operations	7-18
I/O Write Operations	7-19
Non-Prefetchable Read Operations	7-19
Prefetchable Operations	7-21
Prefetchable Write Operations	7-22
Prefetchable Read Operations	7-23
PCI-to-Avalon Address Translation	7-26
PCI Master Operation	7-27
Avalon-To-PCI Read & Write Operation	7-28
Avalon-to-PCI Write Requests	7-31
Avalon-to-PCI Read Requests	7-32
Arbitration Among Pending PCI Master Requests	7-34
Avalon-to-PCI Address Translation	7-35
Ordering of Requests	7-38
Ordering of Avalon-to-PCI Operations	7-39

Ordering PCI-to-Avalon Operations	7-42
PCI Host-Bridge Operation	7-45
Altera-Provided PCI Bus Arbiter	7-45
Interrupts	7-46
Generation of PCI Interrupts	7-46
Reception of PCI Interrupts	7-46
Generation of Avalon-MM Interrupts	7-47
Control & Status Registers	7-47
PCI Interrupt Status Register	7-49
PCI Interrupt Enable Register	7-51
PCI Mailbox Register Access	7-52
Avalon-to-PCI Address Translation Table	7-53
Read-Only Configuration Registers	7-54
Avalon-MM Interrupt Status Register	7-56
Avalon-MM Interrupt Enable Register	7-60
Avalon Mailbox Register Access	7-60

Chapter 8. Testbench

General Description	8-1
Features	8-2
PCI Testbench Files	8-3
Testbench Specifications	8-4
Master Transactor (mstr_tranx)	8-5
PROCEDURES and TASKS Sections	8-5
INITIALIZATION Section	8-6
USER COMMANDS Section	8-7
cfg_rd	8-7
cfg_wr	8-8
mem_wr_32	8-8
mem_rd_32	8-9
mem_wr_64	8-10
mem_rd_64	8-11
io_wr	8-11
io_rd	8-11
Target Transactor (trgt_tranx)	8-12
FILE IO section	8-13
PROCEDURES and TASKS sections	8-13
Bus Monitor (monitor)	8-13
Arbiter (arbiter)	8-14
Pull Up (pull_up)	8-14
Simulation Flow	8-15

Appendix A. Using PCI Constraint File Tcl Scripts

Introduction	A-1
PCI Constraint Files	A-1
Simultaneous Switching Noise (SSN) Considerations	A-2
Additional Options	A-3

Contents

-speed	A-3
-no_compile	A-7
-no_pinouts	A-7
-pin_prefix	A-7
-pin_suffix	A-7
-help	A-8
Upgrading Assignments from a Previous Version of PCI Compiler	A-8
Upgrading PCI Assignments Containing Nondefault PCI Pin Names	A-8

Additional Information

Revision History	Info-i
How to Contact Altera	Info-ii
Typographic Conventions	Info-iii



Introduction

The Altera® PCI Compiler provides many options for creating custom, high-performance PCI bus interface designs. Whether your system's top priority is high bandwidth, high speed, or a combination of features, you can use the PCI Compiler to meet your system requirements.

The PCI Compiler contains the `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore® functions, a Verilog HDL and VHDL testbench, and reference designs. Altera also offers the following development kits as PCI hardware prototyping platforms:

- PCI High-Speed Development Kit, Stratix Professional Edition
- PCI Development Kit, Cyclone II Edition

These kits include a PCI development board, a reference design, software drivers, and a graphical user interface to help you evaluate the PCI solution in a system.

You can create PCI systems using one of the following design flows in the Quartus® II software.

- MegaWizard™ Plug-in Manager flow

This option allows you to choose a specific PCI MegaCore function, specify parameters, generate design files, and manually integrate the parameterized PCI MegaCore function into your overall system.

- SOPC Builder flow

This option allows you to build a complete PCI system—component-by-component—using an automatically-generated system interconnect fabric. The SOPC Builder uses the PCI-Avalon®-Memory-Mapped (Avalon-MM) bridge to connect the PCI bus to the interconnect, allowing you to easily create any system that includes one or more of the Avalon-MM peripherals.

Release Information

Table 1 provides information about this release of the PCI Compiler.

<i>Table 1. PCI Compiler User Guide Release Information</i>	
Item	Description
Version	11.1
Release Date	October 2011
Ordering Codes	IP-PCI/MT64, IP-PCI/T64, IP-PCI/MT32, IP-PCI/T32
Product IDs	pci_mt64 MegaCore function: 0011, pci_t64 MegaCore function: 0025, pci_mt32 MegaCore function: 0022, pci_t32 MegaCore function: 0024
Vendor ID	6AF7

Device Family Support

The MegaCore functions provide either final or preliminary support for target Altera device families:

- **Final support** means the core is verified with final timing models for this device family. The core meets all functional and timing requirements for the device family and can be used in production designs.
- **Preliminary support** means the core is verified with preliminary timing models for this device family. The core meets all functional requirements, but might still be undergoing timing analysis for the device family. It can be used in production designs with caution.
- **HardCopy Compilation** means the core is verified with final timing models for the HardCopy® device family. The core meets all functional and timing requirements for the device family and can be used in production designs.
- **HardCopy Companion** means the core is verified with preliminary timing models for the HardCopy companion device. The core meets all functional requirements, but might still be undergoing timing analysis for HardCopy device family. It can be used in production designs with caution.

Table 2 shows the level of support offered by the User Guide MegaCore functions for each Altera device family.

Table 2. Device Family Support	
Device Family	Support
Arria®GX	Final
Arria II GX	Final
Cyclone®	Final
Cyclone II	Final
Cyclone III	Final
Cyclone III LS	Final
Cyclone IV (E, GX)	Final
HardCopy II	HardCopy Compilation
HardCopy III	Refer to the What's New in Altera IP page of the Altera website.
HardCopy IV (E, GX)	
MAX®II (1)	Final
Stratix®	Final
Stratix GX	Final
Stratix II	Final
Stratix II GX	Final
Stratix III	Final
Stratix IV (E, GX)	Final
Other device families	No support

Note to Table 2:

- (1) MAX II devices are supported by the `pci_mt32` and `pci_t32` MegaCore functions only.

Features

This section summarizes the features of the PCI Compiler.

Common Features

The following list outlines the common features of the PCI Compiler.

- Fully compliant with the PCI Special Interest Group (PCI SIG) *PCI Local Bus Specification, Revision 3.0*
- Supports both 32-bit and 64-bit interfaces
- Supports Master/Target and Target-Only modes

- IP functional simulation models enable simulation of a register transfer level (RTL) model of a PCI MegaCore function in VHDL and Verilog HDL simulators
- OpenCore Plus hardware evaluation feature enables testing of a PCI MegaCore function in hardware prior to purchasing a license
- Configuration registers:
 - Parameterized registers: device ID, vendor ID, class code, revision ID, BAR0 through BAR5, subsystem ID, subsystem-vendor ID, maximum latency, minimum grant, capabilities list pointer, expansion ROM BAR
 - Parameterized default or preset base address (available for all six BARs) and expansion ROM base address
 - Non-parameterized registers: command, status, header type 0, latency timer, cache line size, interrupt pin, interrupt line
- Host bridge application support

PCI Compiler with MegaWizard Plug-in Manager Flow

The following list outlines the features of the PCI Compiler with MegaWizard Plug-in Manager flow.

- IP Toolbench wizard-driven interface makes it easy to generate a custom variation of a PCI MegaCore function
- PCI target features:
 - Capabilities list pointer support
 - Expansion ROM BAR support
 - Local-side requests for target abort, retry, or disconnect
 - Local-side interrupt requests
- PCI master features (`pci_mt64` and `pci_mt32` only):
 - Allows on-chip arbitration logic
 - Allows disabling latency timer
- 64-bit PCI features (`pci_mt64` and `pci_t64` only):
 - 64-bit addressing support as both master and target
 - Initiates 64-bit addressing, using dual-address cycle (DAC)
 - Initiates 64-bit memory transactions
 - Dynamically negotiates 64-bit transactions and automatically multiplexes data on the local 64-bit data bus

PCI Compiler with SOPC Builder Flow

The following list outlines the features of the PCI Compiler with SOPC Builder flow.

- SOPC Builder ready
- PCI complexities, such as retry and disconnect are handled by the PCI/Avalon Bridge logic and transparent to the user

- Hard-coded (fixed) or run-time configurable (dynamic) Avalon-to-PCI address translation
- Hard-coded or automatic PCI-to-Avalon address translation
- Separate Avalon Memory-mapped (Avalon-MM) slave ports for PCI bus access (PBA) and control register access (CRA)
- Support for Avalon-MM burst mode
- Option for independent or common PCI and Avalon clock domains
- Option to increase PCI read performance by increasing the number of pending reads and maximum read burst size.
- Internal Arbiter in Host Bridge and Target/Master mode

General Description

This section provides a general description of the following:

- PCI MegaCore Functions
- PCI Testbench
- PCI Compiler with MegaWizard Plug-in Manager Flow
- PCI Compiler with SOPC Builder Flow

PCI MegaCore Functions

The PCI MegaCore functions are hardware-tested, high-performance, flexible implementations of PCI interfaces. These functions handle the PCI protocol and timing requirements internally. The back-end interface is designed for easy integration, allowing you to focus your engineering efforts on value-added custom development to significantly reduce time-to-market.

Optimized for Altera devices, the PCI MegaCore functions support configuration, I/O, and memory transactions. The small size of the functions, combined with the high density of Altera's devices, provides ample resources for custom local logic to accompany the PCI interface. The high performance of Altera's devices also enables these functions to support unlimited cycles of zero wait state memory-burst transactions. These functions can operate at either 33- or 66-MHz PCI bus clock speeds, allowing them to achieve up to 132 Megabytes per second (MBytes/s) throughput in a 32-bit 33-MHz PCI bus system and up to 528 MBytes/s throughput in a 64-bit 66-MHz PCI bus system.

In the `pci_mt64` and `pci_mt32` functions, the master and target interfaces can operate independently, allowing maximum throughput and efficient usage of the PCI bus. For instance, while the target interface is accepting zero wait state burst write data, the local logic may simultaneously request PCI bus mastership, thus minimizing latency.

To ensure timing and protocol compliance, the PCI MegaCore functions have been rigorously hardware tested. Refer to [“Compliance Summary” on page 10](#) for more information on the hardware tests performed.

PCI Testbench

The PCI testbench, provided in Verilog HDL and VHDL, facilitates the design and verification of systems that implement any of the PCI MegaCore functions. You can build a PCI behavioral simulation environment by using components of the PCI testbench, the IP functional simulation model of your PCI MegaCore function variation, and the rest of your Verilog HDL or VHDL design.

PCI Compiler with MegaWizard Plug-in Manager Flow

With this flow, you design to a low-level interface that allows custom PCI transaction design. Because you are designing the logic to interface to the PCI MegaCore function, you have more control of individual module functionality.



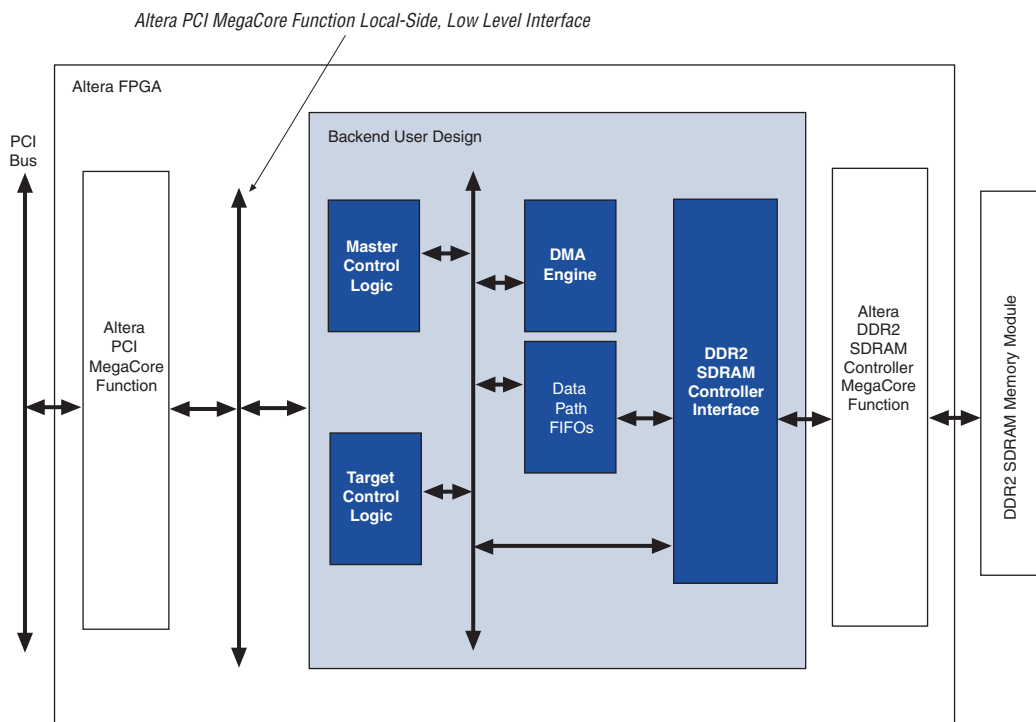
This flow is recommended for users who have previously designed with the PCI Compiler or whose highest priority is to minimize design latency.

For example, if you are designing a PCI-to-DDR2 SDRAM controller interface you need to do the following:

- Specify the PCI MegaCore function parameters.
- Design the ‘back end’ user design, including master control logic, target control logic, data path first-in first-out (FIFO) buffers, and direct memory access (DMA) engine.
- Design the DDR2 SDRAM controller interface.
- Specify the DDR2 SDRAM MegaCore function parameters.
- Design internal PCI and DDR2 SDRAM logic blocks.
- Write RTL code that connects the PCI and DDR2 SDRAM blocks.

Figure 1 shows a PCI-to-DDR2 SDRAM controller interface design using the PCI Compiler with MegaWizard Plug-in Manager flow; shaded areas represent user-customized blocks.

Figure 1. PCI-to-DDR2 SDRAM Design Using the PCI Compiler With MegaWizard Flow



For more information about the PCI Compiler with MegaWizard flow, refer to [Chapter 1, Getting Started](#).

PCI Compiler With SOPC Builder Flow

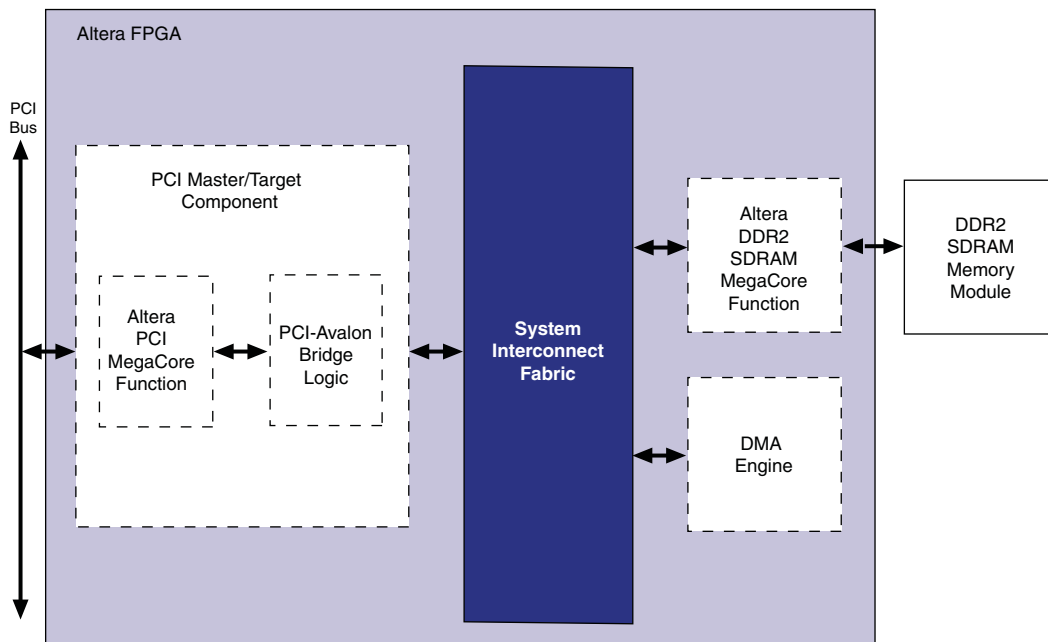
With this flow, you specify system components and choose system options from a rich set of features, and the SOPC Builder then automatically generates the interconnect logic and simulation environment. Thus, you define and generate a complete system in dramatically less time than manually integrating separate IP blocks.



This flow is recommended for users who are new to the PCI Compiler or whose highest priority is to minimize design time.

For example, [Figure 2](#) shows the PCI-to-DDR2 SDRAM design using the PCI Compiler with SOPC Builder flow; the dashed-lines indicate pre-existing components that are added to the design via the SOPC Builder graphical user interface (GUI). When comparing [Figure 1](#) with [Figure 2](#), you can see that the PCI Compiler with SOPC Builder flow option requires far less user customization.

Figure 2. PCI-to-DDR2 SDRAM Design Using the PCI Compiler With SOPC Builder Flow



For more information about the PCI Compiler with SOPC Builder flow, refer to [Chapter 5, Getting Started](#).

For more information about SOPC Builder, refer to [volume 4](#) of the *Quartus II Handbook*.

Selecting the Appropriate Flow for Your Design

Table 3 summarizes the guidelines for selecting a particular flow over another. In most cases, the PCI Compiler with SOPC Builder flow is the appropriate choice.

<i>Table 3. PCI Compiler Parameterization Flow Selection Guidelines</i>	
SOPC Builder Flow	MegaWizard Plug-in Manager Flow
• You would like to quickly integrate multiple system blocks. • You are creating a new PCI design. • You have limited PCI bus protocol experience.	• You are migrating a design that uses a previous version of PCI Compiler. • You require features that are not supported with the SOPC Builder flow.

PCI Compiler With SOPC Builder Flow

This section lists the advantages and disadvantages of the PCI Compiler with the SOPC Builder flow.

Advantages

- Dramatically faster time-to-market
- Requires minimal PCI bus protocol design expertise
- Very short learning curve
- Access to rich feature set
- Uses simple and flexible GUI to create complete PCI system within hours
- Predesigned 'back end' and 'local side' interconnect
- Uses an automatically-generated simulation environment
- Create custom components and integrate them by using the component wizard
- All components are automatically interconnected

Disadvantages

- Does not allow you to customize PCI transaction behavior
- Some applications may have excessive overhead in size and performance

PCI Compiler With MegaWizard Plug-in Manager Flow

This section lists the advantages and disadvantages of the PCI Compiler with MegaWizard Plug-in Manager flow.

Advantages

- More control of the system feature set
- Can design directly from the PCI interface to peripheral devices
- Can access local-side interface to reduce clock cycles and achieve higher bandwidth

Disadvantages

- Requires manual integration of system modules
- Cannot easily use existing SOPC Builder peripherals
- Requires a register transfer level (RTL) file for each instantiation
- Requires significant knowledge of the PCI bus protocol

Compliance Summary

The MegaCore functions are compliant with the requirements specified in the *PCI SIG PCI Local Bus Specification, Revision 3.0* and *Compliance Checklist, Revision 3.0*.

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

- PCI-SIG checklist simulations
- Applicable operating rules in Appendix C of the *PCI Local Bus Specification, Revision 3.0*, including:
 - Basic protocol
 - Signal stability
 - Master and target signals
 - Data phases
 - Arbitration
 - Latency
 - Device selection
 - Parity
- Local-side interface functionality
- Corner cases of the PCI and local-side interface, such as random wait state insertion

In addition to simulation, Altera performed extensive hardware testing on the functions to ensure robustness and PCI compliance. The test platforms include the Agilent E2928A PCI Bus Exerciser and Analyzer, an Altera PCI development board with a device configured with a PCI MegaCore function and a reference design, and PCI bus agents such as a

host bridge, Ethernet network adapter, and video card. The Altera PCI MegaCore functions were tested on the Stratix EP1S25F1020C5 and EP1S60F1020C6 devices. Hardware testing ensures that the PCI MegaCore functions operate flawlessly under the most stringent conditions.

During hardware testing with the Agilent E2928A PCI Bus Exerciser and Analyzer, various tests were performed to guarantee robustness and strict compliance. These tests included the following:

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

The tests generate random transaction types and parameters at the PCI and local sides. The Agilent E2928A PCI Bus Exerciser and Analyzer simulated random behavior on the PCI bus by randomizing transactions with variable parameters such as the following:

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

The local side also emulated a variety of test conditions in which the PCI MegaCore functions experienced random wait states and terminations. During the tests, the Agilent E2928A PCI Bus Exerciser and Analyzer also acted as a PCI protocol and data integrity checker as well as a logic analyzer to aid in debugging. This testing ensures that the functions operate under the most stringent conditions in your system.



For more information on the Agilent E2928A PCI Bus Exerciser and Analyzer, refer to the Agilent website at www.agilent.com.

Performance and Resource Utilization

This section lists the speed and approximate resource utilization of the PCI MegaCore functions in supported Altera device families.

PCI Compiler with MegaWizard Plug-in Manager Flow

The speed and resource utilization estimates are based on a PCI MegaCore function using one BAR that reserves 1 MByte of memory. Implementing additional BARs generates additional logic in the PCI

MegaCore function. Using different parameter options may result in additional logic generated within the function. Results were generated using the Quartus II software version 11.1.

Table 4 shows PCI MegaCore function resource utilization and performance data for Stratix II devices.

Table 4. PCI MegaCore Function Performance in Stratix II Devices (1)			
PCI Function	Utilization (ALUTs) (2)	I/O Pins	f_{MAX} (MHz)
pci_mt64	1,083	89	> 67
pci_t64	714	87	> 67
pci_mt32	754	50	> 67
pci_t32	448	48	> 67

Notes to Table 4:

- (1) This data was obtained by compiling each of the PCI MegaCore functions (parameterized to use one BAR that reserves 1 MByte of memory) in the Stratix II EP2S60F1020C5 device.
- (2) The Utilization for Stratix II devices is based on the number of adaptive look-up tables (ALUTs) used for the design as reported by the Quartus II software.

Table 5 shows PCI MegaCore function resource utilization and performance for Stratix, Stratix GX, and Cyclone devices.

Table 5. PCI MegaCore Function Performance in Stratix, Stratix GX & Cyclone Devices (1)			
PCI Function	Logic Elements (LEs)	I/O Pins	f_{MAX} (MHz)
pci_mt64	1,378	89	> 67
pci_t64	966	87	> 67
pci_mt32	1007	50	> 67
pci_t32	661	48	> 67

Note to Table 5:

- (1) The PCI MegaCore functions use approximately the same number of LEs for the Stratix, Stratix GX, and Cyclone device families. This data was obtained by compiling each of the PCI MegaCore functions (parameterized to use one BAR that reserves 1 MByte of memory) in the Stratix EP1S60F1020C6 device.

Table 6 shows PCI MegaCore function resource utilization and performance data for Cyclone II devices.

Table 6. PCI MegaCore Function Performance in Cyclone II Devices (1)			
PCI Function	Logic Elements (LEs)	I/O Pins	f_{MAX} (MHz)
pci_mt64	1,219	89	> 67
pci_t64	778	87	> 67
pci_mt32	847	50	> 67
pci_t32	504	48	> 67

Note to Table 6:

- (1) This data was obtained by compiling each of the PCI MegaCore functions (parameterized to use one BAR that reserves 1 MByte of memory) in the Cyclone II EP2C35F672C7 device.

Table 7 shows PCI MegaCore function resource utilization and performance for MAX II devices.

Table 7. PCI MegaCore Function Performance in MAX II Devices (1), (2)			
PCI Function	Logic Elements (LEs)	I/O Pins	f_{MAX} (MHz)
pci_mt32	789	50	> 67
pci_t32	455	48	> 67

Notes to Table 7:

- (1) This data was obtained by compiling each of the PCI MegaCore functions (parameterized to use one BAR that reserves 1 MByte of memory) in the MAX II EPM2210F324C3 device.
- (2) pci_mt64 and pci_t64 MegaCore functions are not supported in MAX II devices.

PCI Compiler with SOPC Builder Flow

The speed and resource utilization estimates are for the supported devices when operating in the PCI Target-Only, PCI Master/Target, and PCI Host-Bridge device modes for each of the application-specific performance settings.



Performance results will vary depending on the user-specified parameters that are built into the system module.

Table 8 lists memory utilization and performance data for Stratix II devices.

Table 8. Memory Utilization & Performance Data for Stratix II Devices (4)									
PCI Device Mode	Performance Setting as: (1)		32-Bit PCI Interface			64-Bit PCI Interface			PCI f_{MAX} (MHz)
	PCI Target	PCI Master	Utilization ALUTs(2)	M4K Memory Blocks (3)	I/O Pins	Utilization ALUTs(2)	M4K Memory Blocks (3)	I/O Pins	
PCI Target-Only	Min	N/A	543	0	48	767	0	87	>67
	Typical	N/A	886	4	48	1,165	6	87	>67
	Max	N/A	1,240	4	48	1,556	68	87	>67
PCI Master/Target	Min	Typical	1,726	6	50	2,393	9	89	>67
	Typical	Typical	1,953	8	50	2,729	123	89	>67
	Max	Typical	2,321	8	50	3,114	12	89	>67
	Min	Max	2,532	9	50	3,665	15	89	>67
	Typical	Max	2,753	11	50	3,989	18	89	>67
	Max	Max	3,149	11	50	4,350	18	89	>67

Notes to Table 8:

- (1) **Min** = Single-cycle transactions
Typical = Burst transactions with a single pending read
Max = Burst transactions with multiple pending reads
- (2) The LE count for Stratix II devices is based on the number of adaptive look-up tables (ALUTs) used for the design as reported by the Quartus II software.
- (3) In some compilations one M512 block was used, but it is not counted.
- (4) The data was obtained by performing compilations on a Stratix II EP2S60F1020C5 device. Each of the device types was parameterized to use one BAR that reserved 1 MByte of memory on the Avalon-MM side. For the PCI Master/Target Peripheral mode, one MByte of memory was reserved on the PCI side.

Table 9 lists memory utilization and performance data for Cyclone II devices.

Table 9. Memory Utilization & Performance Data for Cyclone II Devices (2)

PCI Device Mode	Performance Setting as: (1)		32-Bit PCI Interface			64-Bit PCI Interface			PCI f_{MAX} (MHz)
	PCI Target	PCI Master	Logic Elements (LEs)	M4K Memory Blocks	I/O Pins	Logic Elements (LEs)	M4K Memory Blocks	I/O Pins	
PCI Target-Only	Min	N/A	547	0	48	1,114	0	87	>67
	Typical	N/A	1,113	4	48	1,565	6	87	>67
	Max	N/A	1,605	4	48	2,051	6	87	>67
PCI Master/Target	Min	Typical	2,117	7	50	3,075	9	89	>67
	Typical	Typical	2,319	9	50	3,391	13	89	>67
	Max	Typical	2,806	9	50	3,915	13	89	>67
	Min	Max	3,096	7	50	4,655	9	89	>67
	Typical	Max	3,328	9	50	4,939	13	89	>67
	Max	Max	3,806	9	50	5,454	13	89	>67

Notes to Table 9:

- (1) **Min** = Single-cycle transactions
Typical = Burst transactions with a single pending read
Max = Burst transactions with multiple pending reads
- (2) The data was obtained by performing compilations on a Cyclone II EP2C35F672C7 device. Each of the device types was parameterized to use one BAR that reserved 1 MByte of memory on the Avalon-MM side. For the PCI Master/Target Peripheral mode, one MByte of memory was reserved on the PCI side.

Table 10 lists memory utilization and performance data for Stratix, Stratix GX, and Cyclone devices.

Table 10. Memory Utilization & Performance Data for Stratix, Stratix GX & Cyclone Devices (3) (Part 1 of 2)

PCI Device Mode	Performance Setting as: (1)		32-Bit PCI Interface			64-Bit PCI Interface			PCI f_{MAX} (MHz)
	PCI Target	PCI Master	Logic Elements (LEs)	M512 Memory Blocks (2)	I/O Pins	Logic Elements (LEs)	M512 Memory Blocks (2)	I/O Pins	
PCI Target-Only	Min	N/A	852	0	48	1,186	0	87	>67
	Typical	N/A	1,460	4	48	1,949	6	87	>67
	Max	N/A	1,940	4	48	2,442	6	87	>67

Table 10. Memory Utilization & Performance Data for Stratix, Stratix GX & Cyclone Devices (3) (Part 2 of 2)

PCI Device Mode	Performance Setting as: (1)		32-Bit PCI Interface			64-Bit PCI Interface			PCI f_{MAX} (MHz)
	PCI Target	PCI Master	Logic Elements (LEs)	M512 Memory Blocks (2)	I/O Pins	Logic Elements (LEs)	M512 Memory Blocks (2)	I/O Pins	
PCI Master/Target	Min	Typical	2,715	7	50	3,668	10	89	>67
	Typical	Typical	3,053	9	50	4,187	14	89	>67
	Max	Typical	3,540	9	50	4,682	14	89	>67
	Min	Max	3,728	10	50	5,138	16	89	>67
	Typical	Max	4,059	12	50	5,634	20	89	>67
	Max	Max	4,788	14	50	6,696	22	89	>67

Notes to Table 10:

- (1) **Min** = Single-cycle transactions
Typical = Burst transactions with a single pending read
Max = Burst transactions with multiple pending reads
- (2) In Cyclone devices, memory is implemented in M4K blocks, not M512 blocks.
- (3) The data was obtained by performing compilations on a Cyclone EP1C20F400C7 device. Each of the device types was parameterized to use one BAR that reserved 1 MByte of memory on the Avalon-MM side. For the PCI Master/Target Peripheral mode, one MByte of memory was reserved on the PCI side.

Table 11 lists memory utilization and performance data for MAX II devices.



MAX II devices only support the PCI Target-Only peripheral and the single-cycle performance setting.

Table 11. Memory Utilization & Performance Data for MAX II Devices (2)

PCI Device Mode	Performance Setting as: (1)		32-Bit PCI Interface			PCI f_{MAX} (MHz)
	PCI Target	PCI Master	Logic Elements (LEs)	Memory Blocks	I/O Pins	
PCI Target-Only	Min	N/A	770	0	48	>67

Notes to Table 11:

- (1) **Min** = Single-cycle transactions
- (2) The data was obtained by performing compilations on a MAX II EPM2210F324C3 device. The device type was parameterized to use one BAR that reserved 1 MByte of memory on the Avalon-MM side.

Installation and Licensing

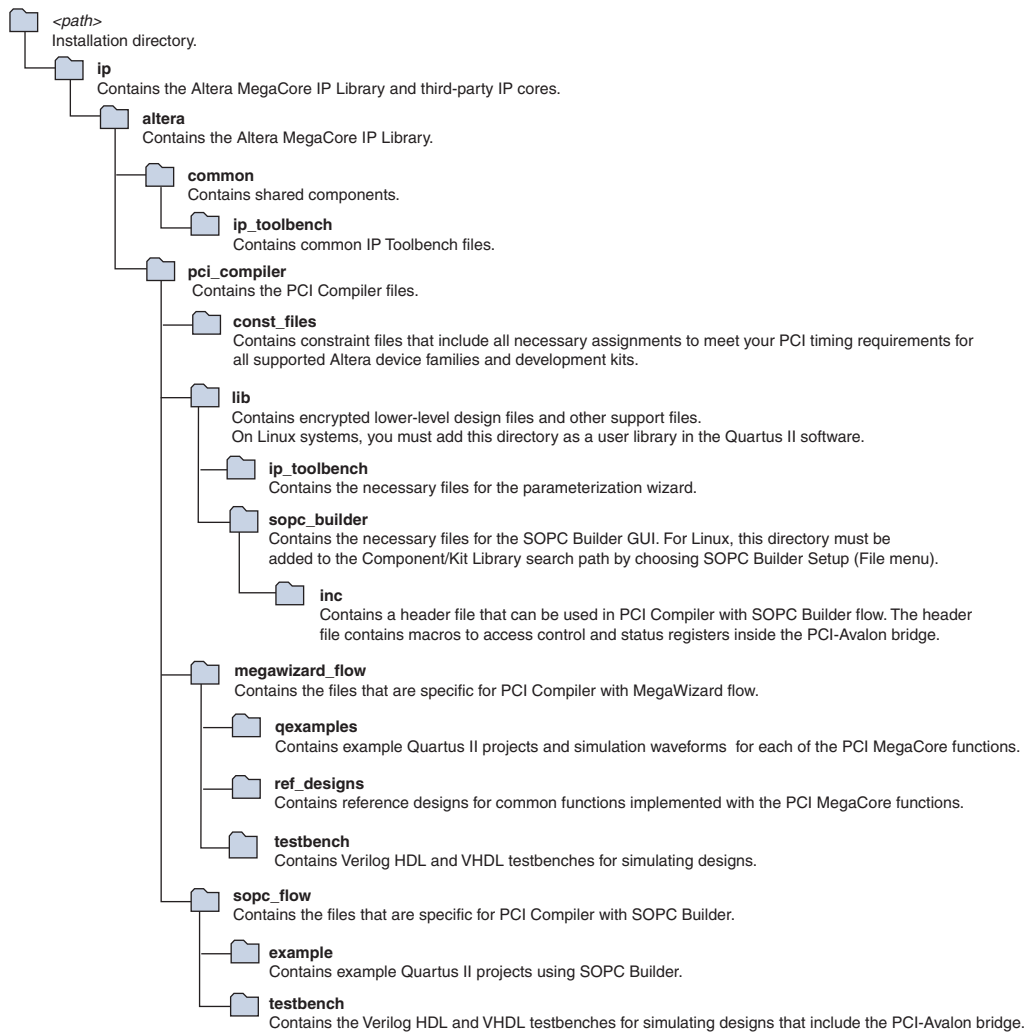


The User Guide is part of the MegaCore IP Library, which is distributed with the Quartus II software and downloadable from the Altera website, www.altera.com.

For system requirements and installation instructions, refer to *Altera Software Installation and Licensing*.

Figure 3 shows the directory structure after you install the PCI Compiler User Guide, where *<path>* is the installation directory. The default installation directory on Windows is **c:\altera\<version>**; on Linux it is **/opt/altera<version>**.

Figure 3. Directory Structure



OpenCore Plus Evaluation

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPPSM megafunction) within your system.
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily.
- Generate time-limited device programming files for designs that include megafunctions.
- Program a device and verify your design in hardware.

You only need to purchase a license for the megafunction when you are completely satisfied with its functionality and performance, and want to take your design to production.

After you purchase a license for PCI Compiler MegaCore function, you can request a license file from the Altera website at www.altera.com/licensing and install it on your computer. When you request a license file, Altera emails you a **license.dat** file. If you do not have Internet access, contact your local Altera representative.



For more information on OpenCore Plus hardware evaluation, refer to *AN 320: OpenCore Plus Evaluation of Megafunctions*.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation supports the following two operation modes:

- *Untethered*—the design runs for a limited time.
- *Tethered*—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely.

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.



For MegaCore functions, the untethered time-out is 1 hour; the tethered time-out value is indefinite.

Your design stops working after the hardware evaluation time expires.



Section I. PCI Compiler With MegaWizard Plug-In Manager Flow

The Altera PCI Compiler provides a complete solution for implementing a conventional PCI interface using Altera devices. It contains the Altera `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore functions, a Verilog HDL and VHDL testbench, and reference designs.

This section includes the following chapters:

- [Chapter 1, Getting Started](#)
- [Chapter 2, Parameter Settings](#)
- [Chapter 3, Functional Description](#)
- [Chapter 4, Testbench](#)

Design Flow

To evaluate a PCI Compiler MegaCore function using the OpenCore Plus feature include these steps in your design flow:

1. Obtain and install the PCI Compiler.
2. Create a custom variation of a PCI MegaCore function using IP Toolbench.



IP Toolbench is a toolbar from which you can quickly and easily view documentation, choose a PCI MegaCore function, specify parameters, and generate all of the files necessary for integrating the parameterized PCI MegaCore function into your design.

3. Implement the rest of your system using the design entry method of your choice.
4. Use the IP Toolbench-generated IP functional simulation model to verify the operation of your design.



For more information on IP functional simulation models, refer to the *Simulating Altera in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

5. Use an Altera-provided PCI constraint file to meet the timing requirements of the PCI specification.



For more information on obtaining and using Altera-provided PCI constraint files in your design, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

6. Use the Quartus II software to compile your design and perform static timing analysis.



You can generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware.

7. Purchase a license for the PCI Compiler.

After you have purchased a license for the PCI Compiler, the design flow involves the following additional steps:

1. Set up licensing.
2. Generate a programming file for the Altera device(s) on your board.
3. Program the Altera device(s) with the completed design.
4. Perform design verification.

PCI MegaCore Function Design Walkthrough

This walkthrough explains how to create a custom variation of a PCI MegaCore function using the Altera PCI IP Toolbench and the Quartus II software. When you finish generating a custom variation of the PCI MegaCore function, you can incorporate it into your overall project.

This walkthrough explains how to create a custom variation of the `pci_mt64` MegaCore function in Verilog HDL. You can also use these procedures for the `pci_mt32`, `pci_t32` and `pci_t64` MegaCore functions, and substitute VHDL for Verilog HDL.

Altera recommends that you use the `pci_mt32` MegaCore function for 32-bit applications. The `pci_mt64` MegaCore function has additional logic and I/O pins which are wasted if used in a 32-bit mode applications.



You can interface the `pci_mt64` MegaCore function with 32-bit agents on the bus. To operate in 32-bit mode only, connect an input pin to the `l_dis_64_extn` signal. This signal disables the 64-bit extension signals if driven low.

This walkthrough consists of these steps:

- Create a New Quartus II Project
- Launch IP Toolbench
- Step 1: Parameterize
- Step 2: Set Up Simulation
- Step 3: Generate

Create a New Quartus II Project

You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity.

To create a new project, follow these steps:

1. Choose **Programs > Altera > Quartus II <version>** (Windows Start menu) to run the Quartus II software. You can also use the Quartus II Web Edition software.
2. Choose **New Project Wizard** (File menu).
3. Click **Next** in the **New Project Wizard: Introduction** (the introduction does not display if you turned it off previously).
4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:

- a. Specify the working directory for your project. This walkthrough uses the directory:

c:\altera\projects

- b. Specify the name of the project. This walkthrough uses **pci_project** for the project name.



The Quartus II software automatically specifies a top-level design entity that has the same name as the project. Do not change it.

5. Click **Next** to close this page and display the **New Project Wizard: Add Files** page.



When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.

6. If you installed the MegaCore IP library in a different directory from where you installed the Quartus II software, add user libraries by following these steps on the **New Project Wizard: Add Files** page:
 - a. Click **User Libraries**.
 - b. Type **<path>\pci_compiler\lib** into the **Library name** box, where **<path>** is the directory in which you installed the PCI Compiler.
 - c. Click **Add to** add the path to the Quartus II project.
 - d. Click **OK** to save the library path in the project.

7. Click **Next** to close this page and display the **New Project Wizard: Family & Device Settings** page.
8. On the **New Project Wizard: Family & Device Settings** page, choose the following:
 - In the **Family** list, choose **Stratix** as the target device family.
 - Under **Target device**, select a **Specific device selected in the 'Available devices' list**.
 - Under **Show in 'Available device' list**, in the **Speed Grade** list, choose **Any**.
 - In the **Available Devices** list, select **EP1S60F1020C5**. These procedures create a design targeting the Stratix device family. You can also use these procedures for other supported device families. MAX II devices are supported by the `pci_mt32` and `pci_t32` MegaCore functions only.
9. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You have finished creating your new Quartus II project.

Launch IP Toolbench

To launch IP Toolbench in the Quartus II software, follow these steps:

1. Start the MegaWizard Plug-In Manager by choosing **MegaWizard Plug-In Manager** (Tools menu). The **MegaWizard Plug-In Manager** dialog box is displayed.
 For more information on MegaWizard Plug-in Manager, refer to Quartus II Help.
2. Specify that you want to create a new custom megafunction variation and click **Next**.
3. Under **Installed Plug-Ins**, expand the **Interfaces>PCI** folder, and click on **PCI** to select the **PCI Compiler v10.1**.
4. Select the output file type for your design; the wizard supports VHDL and Verilog HDL. For this walkthrough, choose **Verilog HDL**.

5. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files using the format *<project path>\<variation name>*. For this walkthrough, specify **c:\altera\projects** for the directory name, and **pci_project.v** for the output file variation name.
6. Click **Next** to launch IP Toolbench for the PCI Compiler.

Step 1: Parameterize

To parameterize your MegaCore function, follow these steps:

1. Click **Step 1: Parameterize** in IP Toolbench to open the **Parameterize - PCI Compiler** dialog box.
 For more information on the parameters you set during this walkthrough, refer to [Chapter 2, Parameter Settings](#).
2. On the **PCI MegaCore Function Settings** page, select the following options:
 - a. Under **Technology**, select **PCI**.
 - b. Under **Application Speed**, turn on **PCI 66-MHz Capable**.
 - c. Select the desired PCI MegaCore function in the PCI MegaCore section. For this walkthrough select **64-Bit Master/Target (pci_mt64)**.
3. Click **Next** to open the **Read-Only PCI Configuration Registers** page. You can modify the values of the read-only PCI configuration registers on this page. For this walkthrough, use the default settings.

4. Click **Next** to open the **Base Address Registers (BARs)** page. This page allows you to configure the PCI base address registers (BARs) that define the address ranges of Memory and I/O write and read requests that your application will claim for the PCI interface.

For this walkthrough, specify these settings:

- a. Ensure that **Implement Only 32 Bit BARs** is selected under **32/64 Bit BARs**.
- b. Click **BAR0 = 1 MBytes (Memory)**.
- c. A window showing default settings for BAR0 displays. For this walkthrough, use the default sliding pointer setting so that **BAR0** reserves 1 MByte (0xFFF00000) of memory.
- d. Click **OK**.
- e. Click **BAR1**.
- f. A window showing the default settings for BAR1 displays. Turn on **Enable**.
- g. Select **I/O** for the type of memory reserved.
- h. Move the sliding pointer so that BAR1 reserves 64 Bytes (0xFFFFFC1) of I/O memory.
- i. Click **OK**.
- j. Select **BAR2 Unused: Click to Configure**.
- k. A window showing default settings of BAR2 displays. Turn on **Enable**.
- l. Move the sliding pointer so that BAR2 reserves 1 MByte (0xFFF00000) of memory.
- m. Click **OK**.
5. Click **Next** to open the **Advanced PCI MegaCore Features** page. For this walkthrough, use the default settings for all options on this page.
6. Click **Finish** to complete the parameterization of your pci_mt64 MegaCore function variation.

Step 2: Set Up Simulation

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model file produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.



Only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis creates a nonfunctional design.



Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.

To generate an IP functional simulation model for your MegaCore function, follow these steps:

1. Click **Step 2: Set Up Simulation** in IP Toolbench.
2. Turn on **Generate Simulation Model**.
3. Choose **Verilog HDL** in the **Language** list.
4. Click **OK**.

Step 3: Generate

Generate your MegaCore function after specifying parameter values and IP functional simulation model options.



Clicking **Quartus II Constraints** displays up-to-date information about PCI Constraint files.



For more information on PCI constraint files, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

To generate your MegaCore function, follow these steps:

1. Click **Step 3: Generate** in IP Toolbench. A summary of files generated to your project directory is displayed.

Table 1–1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL.

Table 1–1. IP Toolbench-Generated Files

Extension	Description
<variation name>.v or .vhd	A MegaCore function variation file that defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<variation name>_bb.v	A Verilog HDL black box file for the MegaCore function variation. Use this file when using a third-party EDA tool to synthesize your design.
<variation name>.bsf	A Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<variation name>.qip	Contains Quartus II project information for your MegaCore function variations.
<variation name>_syn.v	A timing and resource estimation netlist for use in some third-party synthesis tools. This file is generated when the option Generate netlist on the EDA page is turned on.
<variation name>.ppf	This XML file describes the MegaCore pin attributes to the Quartus II Pin Planner. MegaCore pin attributes include pin direction, location, I/O standard assignments, and drive strength. If you launch IP Toolbench outside of the Pin Planner application, you must explicitly load this file to use Pin Planner.
<variation name>.vo or .vho	A Verilog HDL or VHDL IP functional simulation model.
pci_constraints_for_<variation name>.tcl	A tcl script for assigning timing constraints to the MegaCore function.
<variation name>_nativelink.tcl	A tcl script for assigning NativeLink simulation testbench settings to the Quartus project.
<variation name>.html	A MegaCore function report file.

2. After you review the generation report, click **Exit** to close IP Toolbench.



If you generate the MegaCore function instance in a Quartus II project, you are prompted to add the Quartus II IP File (.qip) files to the current Quartus II project. The .qip file is generated by the MegaWizard interface, and contains information about the generated IP core. In most cases, the .qip file contains all of the necessary assignments and information required to process the MegaCore function or system in the Quartus II compiler. The MegaWizard interface generates a single .qip file for each MegaCore function.

You can now integrate your PCI MegaCore function variation into your design and compile.

Simulate the Design

To simulate your design, you use the IP functional simulation models generated by IP Toolbench in conjunction with the Altera-provided PCI testbench. The IP functional simulation model is the .vo or .vho file generated as specified in [“Step 2: Set Up Simulation” on page 1–7](#). These files are generated in the directory you specified in the MegaWizard Plug-In Manager. Compile this IP functional simulation model in your simulation environment as instructed below to perform functional simulation of your PCI MegaCore function variation.

This section of the walkthrough uses the following:

- The IP toolbench-generated PCI testbench in the `c:\altera\projects\pci_project_nativelink\verilog\pci_mt64` directory
- The IP functional simulation model generated as specified in “[Step 2: Set Up Simulation](#)” on page 1–7
- The ModelSim® software
- The generated NativeLink script in the project directory, `c:\altera\projects`

For this walkthrough, follow these steps:

1. On the **EDA Tool Option** page in the Quartus II software (**Tools > Options > EDA Tools Option**), set the location of the ModelSim executable .



If you are using other simulators, set the location of your preferred EDA simulation tool executable. This is a global setting, and needs to be done only once.

2. At the Quartus II Tcl Console, run the following command:
`source pci_top_nativelink.tcl`
3. On the **Simulation** page (**Assignments > EDA Tools Settings > Simulation**), do the following:
 - select ModelSim from the **Tool Name** list
 - select **Compile test bench** under **NativeLink settings**.
4. Perform analysis and synthesis to create the required netlist.
5. Run the simulation.



For more information on simulation using NativeLink, refer to *[Simulating Altera IP in Third-Party Simulation Tools](#)* chapter in volume 3 of the *Quartus II Handbook*.

Simulation in the Quartus II Software

Altera provides Vector Waveform Files (**.vwf**) for each of the PCI MegaCore functions to perform functional simulation in the Quartus II software. The **.vwf** files are provided in the subdirectories at `<path>\pci_compiler\megawizard_flow\qexamples\<PCI MegaCore function>\sim`. For an explanation of the provided **.vwf** files, refer to “[The Quartus II Simulation Files](#)” on page 1–12.

This user guide explains the behavior and usage of the PCI MegaCore functions for the most common PCI transactions. You can use the **.vwf** files to further understand the local-side behavior of a PCI MegaCore function for different PCI bus conditions. In addition, you can modify the provided **.vwf** files to simulate other scenarios of interest.



This procedure demonstrates functional simulation in the Quartus II software of a `pci_mt64` MegaCore function variation. You can also use this procedure for the `pci_mt32`, `pci_t32` and `pci_t64` MegaCore functions.

To perform functional simulation in the Quartus II software, perform these steps:

1. Go to the `<path>\pci_compiler\megawizard_flow\qexamples\pci_mt64` directory.
2. Open the Quartus II project by double-clicking on **pci_top.qpf**.



This Quartus II project contains a PCI MegaCore function variation with the parameter settings required to simulate the included **.vwf** files successfully. For a description of the parameter settings required to simulate the included **.vwf** files, refer to “[The Quartus II Simulation Files](#)” on page 1–12.

3. Choose **Generate Functional Simulation Netlist** (Processing menu).

The Quartus II software may issue several warning messages, including messages indicating that one or more registers are stuck at ground. These warning messages are due to parameter settings and can be ignored.

4. After compilation has finished successfully, choose **Simulator Tool** (Processing Menu).
5. In the **Simulation mode** list, select **Functional**.

6. In the **Simulation input**, specify `<path>\pci_compiler\megawizard_flow\qexamples\pci_mt64\sim\target\cfg_wr_rd.vwf`.
7. Click **Start** to start the simulation.
8. Click **Report** to view the simulation results.

The Quartus II Simulation Files

This section contains information about the Quartus II simulation files supplied with the `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore functions. These simulation files are provided in `.vwf` format.

You can use these simulation files to further understand the local-side behavior of the PCI MegaCore functions for different PCI bus conditions. In addition, you can modify the simulation files to simulate the scenarios of interest.

The simulation files are based on the parameter settings used in `pci_top.v`. There is a separate `pci_top.v` file for each of the four MegaCore functions. The files are located in `<path>\pci_compiler\megawizard_flow\qexamples\<PCI MegaCore function>`. This example design file implements 6 base address registers (BARs), and an expansion ROM BAR, with the following attributes:

- BAR0 reserving 256 Megabytes (MBytes) (memory)
- BAR1 reserving 64 Bytes (I/O)
- BAR2 reserving 16 MBytes (memory)
- BAR3 reserving 1 MByte (memory)
- BAR4 reserving 64 Kilobytes (KBytes) (memory)
- BAR5 reserving 4 KBytes (memory)
- Expansion ROM BAR reserving 1 MByte (memory)

The simulation files contain functional simulation waveforms and should be used after choosing **Generate Functional Simulation Netlist** (Processing menu) for your design.

For more information regarding simulating with `.vwf` files, refer to [“Simulation in the Quartus II Software” on page 1–11](#).

The following sections describe the simulation files provided with the PCI Compiler.

Master Simulation Files

Table 1–2 describes the Quartus II simulation files included in the `<path>\pci_compiler\megawizard_flow\qexamples\pci_mt64\sim\master` directory.

Table 1–2. pci_mt64 Master Simulation Files

Simulation File Name	Description
Master Read	
mubr64	Memory Burst Read, 64-Bit PCI, 64-Bit Local
mubr32_64	Memory Burst Read, 32-Bit PCI, 64-Bit Local
mubr32_32	Memory Burst Read, 32-Bit PCI, 32-Bit Local
mmsr64	Memory Single-Cycle, 64-Bit PCI, 64-Bit Local
mmsr32_32	Memory Single-Cycle, 32-Bit PCI, 32-Bit Local
mubr64_mabrt	Master Abort, 64-Bit PCI, 64-Bit Local
mubr64_tabrt	Target Abort Response, 64-Bit PCI, 64-Bit Local
mubr64_tdisc_wd	Target Disconnect with Data Response, 64-Bit PCI, 64-Bit Local
mubr64_tdisc_wod	Target Disconnect without Data Response, 64-Bit PCI, 64-Bit Local
mubr64_tret	Target Retry Response, 64-Bit PCI, 64-Bit Local
mubr64_lte	Latency Timer Expires, 64-Bit PCI, 64-Bit Local
mior	I/O Read
mcfg_r	Configuration Read
Master Write	
mmbw64	Memory Burst Write, 64-Bit PCI, 64-Bit Local
mmbw32_64	Memory Burst Write, 32-Bit PCI, 64-Bit Local
mmbw32_32	Memory Burst Write, 32-Bit PCI, 64-Bit Local
mmsw32_32	Memory Single-Cycle, 32-Bit PCI, 32-Bit Local
mmbw64_mabrt	Master Abort, 64-Bit PCI, 64-Bit Local
mmbw64_tabrt	Target Abort Response, 64-Bit PCI, 64-Bit Local
mmbw64_tdisc_wd	Target Disconnect with Data Response, 64-Bit PCI, 64-Bit Local
mmbw64_tdisc_wod	Target Disconnect without Data Response, 64-Bit PCI, 64-Bit Local
mmbw64_tret	Target Retry Response, 64-Bit PCI, 64-Bit Local
mmbw64_lte	Latency Timer Expires, 64-Bit PCI, 64-Bit Local
miow	I/O Write
mcfg_w	Configuration Write

Table 1–3 describes the Quartus II simulation files included in the
 <path>\pci_compiler\megawizard_flow\qexamples\
 pci_mt32\sim\master directory.

Table 1–3. pci_mt32 Master Simulation Files	
Simulation File Name	Description
Master Read	
mibr	Memory Burst Read
mmsr	Memory Single-Cycle
mibr_mabrt	Master Abort
mibr_tabrt	Target Abort Response
mibr_tdisc_wd	Target Disconnect with Data Response
mibr_tdisc_wod	Target Disconnect without Data Response
mibr_tret	Target Retry Response
mibr_lte	Latency Timer Expires
mior	I/O Read
mcfr	Configuration Read
Master Write	
mmbw	Memory Burst Write
mmsw	Memory Single-Cycle
mmbw_mabrt	Master Abort
mmbw_tabrt	Target Abort Response
mmbw_tdisc_wd	Target Disconnect with Data Response
mmbw_tdisc_wod	Target Disconnect without Data Response
mmbw_tret	Target Retry Response
mmbw_lte	Latency Timer Expires
miow	I/O Write
mcfgw	Configuration Write

Target Simulation Files

Table 1–4 describes the Quartus II simulation files included in the
`<path>\pci_compiler\megawizard_flow\qexamples\
 <pci_mt64 or pci_t64>\sim\target` directory.

Table 1–4. pci_mt64 & pci_t64 Target Simulation Files

Simulation File Name	Description
Target Read	
tmb64	Memory Burst Read, 64-Bit PCI, 64-Bit Local
tmb32_64	Memory Burst Read, 32-Bit PCI, 64-Bit Local
tms64	Memory Single-Cycle, 64-Bit PCI, 64-Bit Local
tmb64_abrt	Memory Abort, 64-Bit PCI, 64-Bit Local
tmb64_disc_wd	Memory Disconnect with Data, 64-Bit PCI, 64-Bit Local
tmb64_disc_wod	Memory Disconnect without Data, 64-Bit PCI, 64-Bit Local
tmb64_ret	Memory Retry, 64-Bit PCI, 64-Bit Local
cfg_wr_rd	Configuration Write and Read
tior	I/O Read
exp_rom_tmb64	Expansion ROM Memory Burst Read, 64-Bit PCI, 64-Bit Local
Target Write	
tmbw64	Memory Burst Write, 64-Bit PCI, 64-Bit Local
tmbw32_64	Memory Burst Write, 32-Bit PCI, 64-Bit Local
tmsw64	Memory Single-Cycle, 64-Bit PCI, 64-Bit Local
tmbw64_abrt	Memory Abort, 64-Bit PCI, 64-Bit Local
tmbw64_disc_wd	Memory Disconnect with Data, 64-Bit PCI, 64-Bit Local
tmbw64_disc_wod	Memory Disconnect without Data, 64-Bit PCI, 64-Bit Local
tmbw64_ret	Memory Retry, 64-Bit PCI, 64-Bit Local
tiow	I/O Write
exp_rom_tmbw64	Expansion ROM Memory Burst Write, 64-Bit PCI, 64-Bit Local

Table 1–5 describes the Quartus II simulation files included in the `<path>\pci_compiler\megawizard_flow\qexamples\
<pci_mt32 or pci_t32>\sim\target` directory.

Table 1–5. pci_mt32 & pci_t32 Target Directory

Simulation File Name	Description
Target Read	
tibr	Memory Burst Read
tmsr	Memory Single-Cycle
tibr_abrt	Memory Abort
tibr_disc_wd	Memory Disconnect with Data
tibr_disc_wod	Memory Disconnect without Data
tibr_ret	Memory Retry
cfg_wr_rd	Configuration Write and Read
tior	I/O Read
exp_rom_tibr	Expansion ROM Memory Burst Read
Target Write	
tibw	Memory Burst Write
tmsw	Memory Single-Cycle
tibw_abrt	Memory Abort
tibw_disc_wd	Memory Disconnect with Data
tibw_disc_wod	Memory Disconnect without Data
tibw_ret	Memory Retry
tiow	I/O Write
exp_rom_tibw	Expansion ROM Memory Burst Write

Compile the Design

You can use the Quartus II software to compile your design.

Altera provides constraint files to ensure that the PCI MegaCore function achieves PCI specification timing requirements in Altera devices. This walkthrough incorporates a constraint file included with PCI Compiler.



For more information on using Altera-provided constraint files in your design, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

For instructions on compiling your design, refer to Quartus II Help.

For this walkthrough, follow these steps:

1. Open `<path>\pci_example\pci_top.qpf` (the **pci_top** project) in the Quartus II software.



This is the same project you created in “PCI MegaCore Function Design Walkthrough” on page 1–2.

2. Choose **Utility Windows > Tcl Console** (View menu).
3. Source the generated constraint file by typing the following commands at the Quartus II Tcl Console command prompt:

```
source pci_constraints_for_pci_top.tcl ↵
```

```
add_pci_constraints ↵
```



The constraint file uses the following naming convention:

pci_constraints_for_<variation name>.tcl.

4. Monitor the Quartus II Tcl Console to see the actions performed by the script.

To verify the PCI timing assignments in your project, perform the following steps:

1. Choose **Start Compilation** (Processing menu) in the Quartus II software.
2. After compilation, expand the **Timing Analyzer** folder in the **Compilation Report** by clicking the + icon next to the folder name. Note the values in the **Clock Setup**, **tsu**, **th**, and **tco** report sections.

Program a Device

After you have compiled your design, program your targeted Altera device and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the PCI MegaCore function before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model and produce a time-limited programming file.

You can simulate the PCI Compiler MegaCore function in your design and perform a time-limited evaluation of your design in hardware.



For more information on IP functional simulation models, refer to the *Simulating Altera in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

For more information on OpenCore Plus hardware evaluation using the PCI MegaCore functions, refer to “[Compliance Summary](#)” on page 10 and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

For more information on setting up licensing for PCI Compiler, refer to “[PCI Timing Support](#)” on page 1–18.

PCI Timing Support

Designs that use an Altera PCI Compiler MegaCore function must use an Altera-provided PCI constraint file. A PCI constraint file does the following:

- Constrains Quartus II compilations so that your design meets PCI timing requirements
- Specifies the required PCI pin assignments for your board layout

The PCI Compiler generates PCI constraint files in the form of Tcl scripts that allow you to meet the PCI timing requirements in the Quartus II software.

The constraint files use the following naming convention:

pci_constraints_for_<variation name>.tcl

These constraint files have been tested against PCI Compiler 11.1 and Quartus II 11.1 and meet PCI Compiler timing.

To use the constraint file, follow these steps:

1. Open your project in the Quartus II software.

2. In the Quartus II software, choose **Tcl Console** (View > Utility Windows menu).
3. To source the constraint file, type the following in the Quartus II Tcl console:

```
source pci_constraints_for_<variation name>.tcl

add_pci_constraints [-speed "33" | "66"]
[-no_compile] [no_pinouts] [-help]
```



For more information on PCI Compiler constraint files, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

Using the Reference Designs

The following sections outline how to use the reference designs that are packaged with the PCI Compiler

pci_mt32 MegaCore Function Reference Design

The pci_mt32 MegaCore Function Reference Design example illustrates how to interface local logic to the pci_mt32 MegaCore function. The reference design includes a target and a master interface to the pci_mt32 function and the SDRAM memory. The DMA engine is implemented in the local logic to enable the pci_mt32 function to operate as a bus master. The design implements a FIFO interface to solve latency issues when data is transferred between the PCI bus and the SDRAM.

The pci_mt32 MegaCore Function Reference Design requires the Quartus II software.

[Table 1–6](#) describes the directory structure of the pci_mt32 MegaCore Function Reference Design. The directory names are relative to the following path:

```
<path>/pci_compiler/megawizard_flow
/ref_designs/ref_designs/pci_mt32/vhdl
```

where *<path>* is the directory in which you installed the PCI Compiler.

Refer to [Table 1–3](#) the pci_mt32 MegaCore Function Reference Design Directory Structure for more details regarding the directory structure.

Table 1–6. Directory Structure of pci_mt32 MegaCore Reference Design	
Directory Name	Description
chip_top	This directory contains a top-level design file that instantiates the following modules: • pci_mt32 MegaCore function variation file • Local interface logic • SDR SDRAM interface • SDR SDRAM controller
pci_top	This directory contains a pci_mt32 MegaCore function top-level wrapper file. This wrapper file was generated using IP Toolbench with the following parameters selected using the Parameterize - PCI Compiler Wizard : • BAR0 reserves 1MB of memory space • BAR1 reserves 32MB of memory space
pci_local	This directory contains local interface logic files. For more information on these files, refer to FS 12: pci_mt32 MegaCore Function Reference Design.
sdr_intf	This directory contains files for the interface logic between the PCI local interface logic and the SDR SDRAM controller.
sdr_cntrl	This directory contains files for the SDR SDRAM controller.

Synthesis & Compilation Instructions

To compile the pci_mt32 MegaCore Function Reference Design in the Quartus II software, perform the following steps:

1. Create a new project in the Quartus II software, specifying `<path>/pci_compiler/megawizard_flow/ref_designs/pci_mt32/vhdl/chip_top.vhd` as the top-level design file.
2. Add the following directories as user libraries in the Quartus II software:

```
<path>/pci_compiler/lib
```

```
<path>/pci_compiler/megawizard_flow/
ref_designs/pci_mt32/vhdl/chip_top
```

```
<path>/pci_compiler/megawizard_flow
/ref_designs/pci_mt32/vhdl/pci_local
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt32/vhdl/sdr_intf
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt32/vhdl/sdr_cntrl
```



Refer to Quartus II help for information on how to add user libraries in the Quartus II software.

3. Include the following files in your Quartus II project:

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt32/vhdl/chip_top  
/vhdl_components.vhd
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt32/vhdl/pci_top/pci_top.vhd
```

4. Select the appropriate Altera device for your project.

Use an Altera-provided PCI constraint file.



For more information on using PCI constraint files, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

5. Compile your project.

pci_mt64 MegaCore Function Reference Design

The pci_mt64 MegaCore Function Reference Design is an example that shows how to connect the local-side signals of the Altera pci_mt64 MegaCore function to local-side applications when the MegaCore function is used as a master or target on the PCI bus. The reference design consists of the following elements:

- Master control logic
- Target control logic
- DMA engine
- Data path FIFO buffer functions
- SDRAM interface

The pci_mt64 MegaCore Function Reference Design requires the Quartus II software.

[Table 1–7](#) describes the directory structure of the pci_mt64 MegaCore Function Reference Design. The directory names are relative to the following path:

```
<path>/pci_compiler/megawizard_flow
/ref_designs/pci_mt64/vhdl
```

where *<path>* is the directory in which you installed the PCI Compiler.

Table 1–7. Directory Structure of pci_mt64 MegaCore Reference Design

Directory Name	Description
chip_top	This directory contains a top-level design file that instantiates the following modules: • pci_mt64 MegaCore function variation file • Local interface logic • SDR SDRAM interface • SDR SDRAM controller
pci_top	This directory contains a pci_mt64 MegaCore function top-level wrapper file. This wrapper file was generated using IP Toolbench with the following parameters selected using the Parameterize - PCI Compiler Wizard : • BAR0 reserves 1MB of memory space • BAR1 reserves 32MB of memory space
pci_local	This directory contains local interface logic files. For more information on these files, refer to FS 10: pci_mt64 MegaCore Function Reference Design.
sdr_intf	This directory contains files for the interface logic between the PCI local interface logic and the SDR SDRAM controller.
sdr_cntrl	This directory contains files for the SDR SDRAM controller.

synthesis & Compilation Instructions

To compile the pci_mt64 MegaCore Function Reference Design in the Quartus II software, follow these steps:

1. Create a new project in the Quartus II software, specifying the top-level design file as follows:

```
<path>/pci_compiler/megawizard_flow
/ref_designs/pci_mt64/vhdl/chip_top.vhd
```

2. Add the following directories as user libraries in the Quartus II software:

```
<path>/pci_compiler/lib

<path>/pci_compiler/megawizard_flow
/ref_designs/pci_mt64/vhdl/chip_top
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt64/vhdl/pci_local
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt64/vhdl/sdr_intf
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt64/vhdl/sdr_cntrl
```



Refer to Quartus II help for information on how to add user libraries in the Quartus II software.

3. Include the following files in your Quartus II project:

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt64/vhdl/chip_top  
/vhdl_components.vhd
```

```
<path>/pci_compiler/megawizard_flow  
/ref_designs/pci_mt64/vhdl/pci_top/pci_top.vhd
```

4. Select the appropriate Altera device for your project.
5. Use an Altera-provided PCI constraint file for the device you have selected.



For more information on using PCI constraint files, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

6. Compile your project.

This chapter describes the parameters available to configure PCI Compiler, including:

- “PCI MegaCore Function Settings”
- “Read-Only PCI Configuration Registers”
- “PCI Base Address Registers (BARs)”
- “Advanced PCI MegaCore Function Features”
- “Variation File Parameters”

Parameterize PCI Compiler

You can customize the PCI MegaCore functions by changing parameters and specifying optional features using the **Parameterize - PCI Compiler** wizard. Start the wizard by clicking **Step 1: Parameterize** in IP Toolbench.

These parameters allow you to customize the PCI MegaCore functions to meet specific application requirements, such as defining read-only and read/write PCI configuration space. The wizard is also used to enable and parameterize optional features.

For a complete list of parameter names and descriptions found in a generated PCI MegaCore function variation file, refer to “[Variation File Parameters](#)” on page 2–7.

PCI MegaCore Function Settings

The PCI MegaCore functions are capable of operating at clock speeds of up to 66 MHz. Depending on the PCI device speed, the **PCI 66-MHz Capable** option is enabled or disabled on the **PCI MegaCore Function Settings** page of the **Parameterize - PCI Compiler** wizard.

When turned on, the **PCI 66-MHz Capable** option sets bit 5 of the PCI configuration space status register. For more information on the function of this register, refer to “[Configuration Registers](#)” on page 3–28.

Read-Only PCI Configuration Registers

Parameters for read-only PCI configuration space registers are defined on the **Read-Only PCI Configuration Registers** page of the **Parameterize - PCI Compiler** wizard.

The following read-only PCI configuration space register parameters are set on this page:

- Device ID
- Vendor ID
- Revision ID
- Subsystem ID
- Subsystem Vendor ID
- Minimum Grant
- Maximum Latency
- Class Code

The parameters require hexadecimal values. For information on the functionality of the read-only registers, refer to [“Configuration Registers” on page 3–28](#).

PCI Base Address Registers (BARs)

The PCI MegaCore functions implement up to six 32-bit BARs and an expansion ROM BAR. The `pci_mt64` and `pci_t64` MegaCore functions can also implement one 64-bit BAR using either BAR 1 and BAR0, or BAR2 and BAR1.

You must instantiate at least one BAR in your application design. Multiple BARs must be implemented in sequence starting from BAR0. By default, BAR0 is enabled and reserves 1 MByte of memory space.

In addition to allowing normal BAR operation where the system writes the base address value during system initialization, the PCI MegaCore functions allow the base address of any BAR to be hardwired using the **Hardwire BAR** option. When hardwiring a BAR, the BAR address becomes a read-only value supplied to the PCI MegaCore function through the parameter value. System software cannot overwrite a base address register that is hardwired. The value provided for the hardwired BAR is written into the BAR, including the four least significant bits. Thus, you must provide the appropriate value for all of the contents of the BAR.



Use hardwired BARs in closed systems only.

The PCI BAR attributes are defined on the **Base Address Registers (BARs)** page of the **Parameterize - PCI Compiler** wizard.

The `pci_mt64` and `pci_t64` MegaCore functions allow the implementation of 64-bit BARs. When implementing a 64-bit BAR, most systems do not require that all of the upper bits be decoded. The PCI MegaCore functions allow the number of read/write bits on the upper BAR to be defined for specific application needs. For example, if the maximum size of memory in your system is 512 Gigabytes (GBytes), you only need 8 bits of the most significant BAR to be decoded. The acceptable range of read/write bits is between 8 and 32. When the maximum number of read/write bits is set to 32, all bits of the most significant BAR will be decoded.

For more information on the function of BARs, refer to “[Base Address Registers](#)” on page 3–37.

Advanced PCI MegaCore Function Features

Optional registers, interrupt capabilities, and optional master features are set on the **Advanced PCI MegaCore Function Features** page of the **Parameterize - PCI Compiler** wizard.

Optional Registers

The PCI MegaCore functions support two optional read-only registers: the capabilities list pointer register and CIS cardbus pointer register. When these features are used, the values provided in the wizard are stored in these optional registers. When **CompactPCI** technology is selected on the initial page of the wizard, the capabilities list pointer register on the **Advanced PCI MegaCore Function Features** page is automatically turned on with the default value of `0x40`.

Optional Interrupt Capabilities

The PCI MegaCore functions support optional PCI interrupt capabilities. For example, if an application uses the interrupt pin, the interrupt pin register indicates that the interrupt signal (`intan`) is used by storing a value of 0x01 in the interrupt pin register. Turning off **Use Interrupt Pin** on the **Advanced PCI MegaCore Function Features** page results in the interrupt pin register being set to 0x00.

The PCI MegaCore functions also include an option to respond to the interrupt acknowledge command. If **Support Interrupt Acknowledge Command** is turned off, the PCI MegaCore function ignores the interrupt acknowledge command. When **Support Interrupt Acknowledge Command** is turned on, the PCI MegaCore function responds to the interrupt acknowledge command by treating it as a regular target memory read. The local side must implement the logic necessary to respond to the interrupt acknowledge command.

For more information on the capabilities list pointer, CIS cardbus pointer, and interrupt pin registers, refer to [“Configuration Registers” on page 3–28](#).

Master Features

The `pci_mt64` and `pci_mt32` MegaCore functions also provide the following options available in the **Parameterize - PCI Compiler** wizard:

- **Allow Variable Byte Enables During Burst Transactions**
- **Use in Host Bridge Application**
- **Allow Internal Arbitration Logic**
- **Disable Master Latency Timer**
- **Assume ack64n Response**

Enable these features on the **Advanced PCI MegaCore Function Features** page as described in the following sections.

Allow Variable Byte Enables During Burst Transactions

In a default master burst transaction the byte enables accompanying the initial data word provided by the local side are used throughout the master burst transaction. Turning on **Allow Variable Byte Enables During Burst Transactions** allows byte enables to change for successive data words during the transaction. This option affects both burst memory read and burst memory write master transactions. However, use this option only for burst memory write master transactions. Refer to [“Burst Memory Write Master Transaction with PCI Wait State” on page 3–117](#) for more information. For burst memory read master transactions, you must keep the byte enables constant throughout the transaction. Typically the byte enable values are set to 0x00 for burst memory read master transactions.

Use in Host Bridge Application

Turning on the **Use in Host Bridge Application** option allows you to implement a host bridge design using the `pci_mt64` and `pci_mt32` MegaCore functions. For more information on using the `pci_mt64` or `pci_mt32` MegaCore functions in a host bridge application, refer to [“Host Bridge Operation” on page 3–127](#).

Allow Internal Arbitration Logic

Many designs that utilize the `pci_mt64` or `pci_mt32` MegaCore functions as a host bridge implement other central resource functionality in the same FPGA as the PCI interface. Turning on **Allow Internal Arbitration Logic** option allows you to include the PCI bus arbiter in the same FPGA as the PCI MegaCore function.

If the **Allow Internal Arbitration Logic** option is not selected, the `reqn` signal output from the `pci_mt64` and `pci_mt32` functions is implemented with a tri-state buffer, which prevents `reqn` from being connected to internal logic and subsequently to `gntn` without the use of device I/Os. Turning on **Allow Internal Arbitration Logic** removes the tri-state buffer from the `reqn` signal output, allowing the signal to be connected to internal FPGA logic and eliminating the need to use additional device I/O resources or board traces.

Disable Master Latency Timer

Turning on the **Disable Master Latency Timer** option allows you to disable the latency timer time-out feature. If the latency timer time-out is disabled, the master will continue the burst transaction even if the latency timer has expired and the `gntn` signal is removed. This feature is useful in systems in which breaking up long data transfers in small transactions will yield undesirable side effects.



Disabling the **Disable Master Latency Timer** violates the PCI specification and therefore should only be used in embedded applications where the designer can control the entire system configuration. Disabling the master latency timer can also result in increased latency for other master devices in the system. If increased latency for other master devices is unacceptable in your application, this option should not be used.

Assume ack64n Response

This feature provides enhanced master functionality when using the `pci_mt64` MegaCore function in systems where a 64-bit transaction request is always accepted by a 64-bit target asserting `ack64n`. This feature can be used where the bit width of all devices is known, such as in an embedded system, and where all 64-bit targets respond with `ack64n` asserted.

With this option turned on, the `pci_mt64` master supports 64-bit single-cycle write transactions and asserts `irdyn` one clock cycle after `framen` is asserted. For more information, refer to “[64-Bit Single Cycle Memory Write Master Transactions](#)” on page 3–121.

Variation File Parameters

If you do not want to use the IP Toolbench **Parameterize - PCI Compiler** wizard, you can specify Altera PCI MegaCore function parameters directly in the hardware description language (HDL) or graphic design files. [Table 2–1](#) provides parameter names and descriptions.

Table 2–1. PCI MegaCore Function Parameters (Part 1 of 5)

Name	Format	Default Value	Description
DEVICE_ID	Hexadecimal	H"0004"	Device ID register. This parameter is a 16-bit hexadecimal value that sets the device ID register in the configuration space. Any value can be entered for this parameter.
CLASS_CODE	Hexadecimal	H"FF0000"	Class code register. This parameter is a 24-bit hexadecimal value that sets the class code register in the configuration space. The value entered for this parameter must be a valid PCI SIG-assigned class code register value.
MAX_LATENCY (1)	Hexadecimal	H"00"	Maximum latency register. This parameter is an 8-bit hexadecimal value that sets the maximum latency register in the configuration space. This parameter must be set according to the guidelines in the PCI specification.
MIN_GRANT (1)	Hexadecimal	H"00"	Minimum grant register. This parameter is an 8-bit hexadecimal value that sets the minimum grant register in the PCI configuration space. This parameter must be set according to the guidelines in the PCI specification.
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 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 configuration space. Any value can be entered for this parameter.
SUBSYSTEM_VEND_ID	Hexadecimal	H"0000"	Subsystem vendor ID register. This parameter is a 16-bit hexadecimal value that sets the subsystem vendor ID register in the PCI configuration space. The value for this parameter must be a valid PCI SIG-assigned vendor ID number.

Table 2–1. PCI MegaCore Function Parameters (Part 2 of 5)

Name	Format	Default Value	Description
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 configuration space. The value for this parameter can be the Altera vendor ID (1172 Hex) or any other PCI SIG-assigned vendor ID number.
BAR0 (2)	Hexadecimal	H"FFF00000"	Base address register (BAR) zero. When implementing a 64-bit base address register that uses BAR0 and BAR1, BAR0 contains the lower 32-bit address. For more information, refer to “PCI Base Address Registers (BARs)” on page 2–2 .
BAR1 (2)	Hexadecimal	H"FFF00000"	Base address register one. When implementing a 64-bit base address register that uses BAR0 and BAR1, BAR1 contains the upper 32-bit address. When implementing a 64-bit base address register that uses BAR1 and BAR2, BAR1 contains the lower 32-bit address. For more information, refer to “PCI Base Address Registers (BARs)” on page 2–2 .
BAR2 (2)	Hexadecimal	H"FFF00000"	Base address register two. When implementing a 64-bit base address register that uses BAR1 and BAR2, BAR2 contains the upper 32-bit address. For more information, refer to “PCI Base Address Registers (BARs)” on page 2–2 .
BAR3 (2)	Hexadecimal	H"FFF00000"	Base address register three.
BAR4 (2)	Hexadecimal	H"FFF00000"	Base address register four.
BAR5 (2)	Hexadecimal	H"FFF00000"	Base address register five.
EXP_ROM_BAR	String	H"FF000000"	Expansion ROM. This value controls the number of bits in the expansion ROM BAR that are read/write and will be decoded during a memory transaction.

Table 2–1. PCI MegaCore Function Parameters (Part 3 of 5)

Name	Format	Default Value	Description
HARDWIRE_BAR _n	Hexadecimal	H"FF000000"	Hardwire base address register. <i>n</i> corresponds to the base address register number and can be from 0 to 5. HARDWIRE_BAR_n is a 32-bit hexadecimal value that permanently sets the value stored in the corresponding BAR. This parameter is ignored if the corresponding HARDWIRE_BAR_n_ENA bit is not set to 1. When the corresponding HARDWIRE_BAR_n_ENA bits are set to 1, the function returns the value in HARDWIRE_BAR_n during a configuration read. To detect a base address register hit, the function compares the incoming address to the upper bits of the HARDWIRE_BAR_n 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.
HARDWIRE_EXP_ROM	Hexadecimal	H"FF000000"	Hardwire expansion ROM BAR. HARDWIRE_EXP_ROM is the default expansion ROM base address. This parameter is ignored when HARDWIRE_EXP_ROM_ENA is set to 0. When HARDWIRE_EXP_ROM_ENA is set to 1, the function returns the value in HARDWIRE_EXP_ROM during a configuration read. To detect base address hits for the expansion ROM, the functions compare the input address to the upper bits of HARDWIRE_EXP_ROM. HARDWIRE_EXP_ROM_ENA must be set to enable expansion ROM support, and the HARDWIRE_EXP_ROM parameter setting defines the number of decoded bits.

Table 2–1. PCI MegaCore Function Parameters (Part 4 of 5)

Name	Format	Default Value	Description
MAX_64_BAR_RW_BITS	Decimal	8	Maximum number of read/write bits in upper BAR when using a 64-bit BAR. This parameter controls the number of bits decoded in the high BAR of a 64-bit BAR. (Values for this parameter are integers from 8 to 32.) For example, setting this parameter to eight (the default value) allows the user to reserve up to 512 Gigabytes (GBytes). Note: Most systems will not require that all of the upper bits of a 64-bit BAR be decoded. This parameter controls the size of the comparator used to decode the high address of the 64-bit BAR.
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 MegaCore function sequentially instantiates the number of BARs specified by this parameter starting with BAR0. When implementing a 64-bit BAR, two BARs are used; therefore, the NUMBER_OF_BARS parameter should be raised by two.
CAP_PTR	Hexadecimal	H"40"	Capabilities list pointer register. This 8-bit value sets the capabilities list pointer 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 <i>PCMCIA Specification, version 3.0</i> . The functions ignore this parameter if CIS_PTR is not set to 0. In other words, if the CIS_PTR_ENA bit is set to 1, the functions return the value in CIS_PTR during a configuration read to the CIS pointer register. The function returns H"00000000" during a configuration read to CIS when CIS_PTR_ENA is set to 0.
ENABLE_BITS	Hexadecimal	H"00000000"	Feature enable bits. This parameter is a 32-bit hexadecimal value which controls whether various features are enabled or disabled. The bit definition of this parameter is shown in Table 2–2 .

Table 2–1. PCI MegaCore Function Parameters (Part 5 of 5)

Name	Format	Default Value	Description
INTERRUPT_PIN_REG	Hexadecimal	H"01"	Interrupt pin register. This 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 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.
PCI_66MHZ_CAPABLE	String	"YES"	PCI 66-MHz capable. When set to "YES", this parameter sets bit 5 of the status register to enable 66-MHz operation.

Notes to Table 2–1:

- (1) These parameters affect master functionality, therefore, they only affect the pci_mt64 and pci_mt32 MegaCore functions.
- (2) The BAR0 through BAR5 parameters control the options of the corresponding BAR instantiated in the PCI MegaCore function. Use BAR0 through BAR5 for I/O and 32-bit memory space. If you use a 64-bit BAR in pci_mt64 or pci_t64, it must be implemented on either BAR0 and BAR1 or BAR1 and BAR2. Consequently, the remaining BARs can still be used for I/O and 32-bit memory space.

Table 2–2 shows the bit definition for ENABLE_BITS.

Table 2–2. Bit Definition of the ENABLE_BITS Parameter (Part 1 of 5)

Bit Number	Bit Name	Default Value	Definition
5..0	HARDWIRE_BARn_ENA	B"000000"	Hardwire BAR enable. This bit indicates that the user wants to use a default base address at power-up. <i>n</i> corresponds to the BAR number and can be from 0 to 5.
6	HARDWIRE_EXP_ROM_ENA	0	Hardwire expansion ROM BAR enable. This bit indicates that the user wants to use a default expansion ROM base address at power-up.
7	EXP_ROM_ENA	0	Expansion ROM enable. This bit enables the capability for the expansion ROM base address register. If this bit is set to 1, the function uses the value stored in EXP_ROM_BAR to set the size and number of bits decoded in the expansion ROM BAR. Otherwise, the expansion ROM BAR is read only and the function returns H"0000000" when the expansion ROM BAR is read.

Table 2–2. Bit Definition of the ENABLE_BITS Parameter (Part 2 of 5)

Bit Number	Bit Name	Default Value	Definition
8	CAP_LIST_ENA	0	Capabilities list enable. This bit determines if the capabilities list will be enabled in the configuration space. When this bit is set to 1, it sets the capabilities list bit (bit 4) of the status register and sets the capabilities register to the value of CAP_PTR.
9	CIS_PTR_ENA	0	CardBus CIS pointer enable. This bit enables the CardBus CIS pointer register. When this bit is set to 0, the function returns H"00000000" during a configuration read to the CIS_PTR register.
10	INTERRUPT_ACK_ENA	0	Interrupt acknowledge enable. This bit enables support for the interrupt-acknowledge command. When set to 0, the function ignores the interrupt acknowledge command. When set to 1, the function responds to the interrupt acknowledge command. The function treats the interrupt acknowledge command as a regular target memory read. The local side must implement the necessary logic to respond to the interrupt controller.
11	Reserved	0	Reserved.
12	INTERNAL_ARBITER_ENA (1)	0	This bit allows <code>reqn</code> and <code>gntn</code> to be used in internal arbiter logic without requiring external device pins. If the PCI MegaCore function and a PCI bus arbiter are implemented in the same device, the <code>reqn</code> signal should feed internal logic and <code>gntn</code> should be driven by internal logic without using actual device pins. If this bit is set to 1, the tri-state buffer on the <code>reqn</code> signal is removed, allowing an arbiter to be implemented without using device pins for the <code>reqn</code> and <code>gntn</code> signals.

Table 2–2. Bit Definition of the ENABLE_BITS Parameter (Part 3 of 5)

Bit Number	Bit Name	Default Value	Definition
13	SELF_CFG_HB_ENA (1)	0	Host bridge enable. This bit controls the self-configuration host bridge functionality. Setting this bit to 1 causes the <code>pci_mt64</code> and <code>pci_mt32</code> MegaCore functions to power up with the master enable bit in the command register hardwired to 1 and allows the master interface to initiate configuration read and write transactions to the internal configuration space. This feature does not need to be enabled for the <code>pci_mt64</code> or <code>pci_mt32</code> master to initiate configuration read and write transactions to other agents on the PCI bus. Finally, you will still need to connect <code>IDSEL</code> to one of the high order bits of the <code>AD</code> bus as indicated in the <i>PCI Local Bus Specification, version 3.0</i> to complete configuration transactions.
14	LOC_HDAT_MUX_ENA	0	Add internal data steering logic for 32- and 64-bit systems. This bit controls the data and byte enable steering logic that was implemented in the <code>pci_mt64</code> and <code>pci_t64</code> MegaCore functions before version 2.0.0. When this bit is set to 0, only the <code>l_dato[31..0]</code> and <code>l_beno[3..0]</code> buses will contain valid data during a 32-bit master read (when a 64-bit transaction was requested) or a 32-bit target write. Setting this bit to 1 will implement the steering logic, providing 100% backward compatible operation with versions prior to 2.0.0. If starting a new design, Altera recommends adding the data steering logic in the local side application for lower logic utilization and better overall performance.

Table 2–2. Bit Definition of the ENABLE_BITS Parameter (Part 4 of 5)

Bit Number	Bit Name	Default Value	Definition
15	DISABLE_LAT_TMR (1)	1	Disable master latency timer. This bit controls whether the latency timer circuitry will operate as indicated in the <i>PCI Local Bus Specification, version 3.0</i> . When this bit is set to 0, the latency timer circuitry will operate normally and will force the <code>pci_mt64</code> or <code>pci_mt32</code> master to relinquish bus ownership as soon as possible when the latency timer has expired and <code>gntn</code> is not asserted. If this bit is set to 1, the latency timer circuitry is disabled. In this case, the <code>pci_mt64</code> or <code>pci_mt32</code> master will relinquish bus ownership normally when the local side signal <code>lm_lastn</code> is asserted or when the target terminates the PCI transaction with a retry, disconnect, or abort.
16	PCI_64BIT_SYSTEM	0	64-bit only PCI devices. This bit allows enhanced master capabilities when the <code>pci_mt64</code> function is used in systems where a 64-bit master request will always be accepted by a 64-bit target device (target device always responds with <code>ack64n</code> asserted). When this bit is set to 1, the <code>pci_mt64</code> master will: Support 64-bit single-cycle master write transactions Assert <code>irdyn</code> one clock cycle after the assertion of <code>framen</code> for read and write transactions. This option should only be used in embedded applications where the designer controls the entire system configuration. This option does not affect target transactions and does not affect master 32-bit transactions including transactions using the <code>lm_req32n</code>, configuration, and I/O transactions.

Table 2–2. Bit Definition of the ENABLE_BITS Parameter (Part 5 of 5)

Bit Number	Bit Name	Default Value	Definition
17	MW_CBEN_ENA	0	In a standard master burst transaction the byte enables accompanying the initial data word provided by the local side are used throughout the master burst transaction. Turning on Allow Variable Byte Enables During Burst Transactions allows byte enables to change for successive data words during the transaction. This option affects both burst memory read and burst memory write master transactions. However, use this option only for burst memory write master transactions. Refer to “ Burst Memory Write Master Transaction with Variable Byte Enables ” on page 3–119 for more information. For burst memory read master transactions, you must keep the byte enables constant throughout the transaction. Typically the byte enable values are set to 0 for burst memory read master transactions.
31..18	Reserved	0	Reserved.

Note to Table 2–2:

- (1) These parameters affect master functionality and therefore only affect the pci_mt64 and pci_mt32 MegaCore functions.

This chapter contains detailed information on the PCI Compiler and the PCI MegaCore functions, including the following:

- “Functional Overview”
- “PCI Bus Signals”
- “PCI Bus Signals”
- “PCI Bus Commands”
- “Configuration Registers”
- “Target Mode Operation”
- “Master Mode Operation”
- “Host Bridge Operation”
- “64-Bit Addressing, Dual Address Cycle (DAC)”

Functional Overview

This section provides a general overview of `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` functionality. It describes the operation and assertion of master and target signals.

Figures 3–1 through 3–4 show the block diagrams for the `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` functions, respectively. The functions consist of several blocks:

- *PCI bus configuration register space*—implements all configuration registers required by the *PCI Local Bus Specification, Revision 3.0*
- *Parity checking and generation*—responsible for parity checking and generation, as well as assertion of parity error signals and required status register bits
- *Target interface control logic*—controls the operation of the corresponding PCI MegaCore function on the PCI bus in target mode
- *Master interface control logic*—controls the PCI bus operation of the corresponding PCI MegaCore function in master mode (`pci_mt64` and `pci_mt32` MegaCore functions only)
- *Local target control*—controls local-side interface operation in target mode
- *Local master control*—controls the local side interface operation in master mode (`pci_mt64` and `pci_mt32` MegaCore functions only)
- *Local address/data/command/byte enables*—multiplexes and registers all address, data, command, and byte-enable signals to the local side interface.

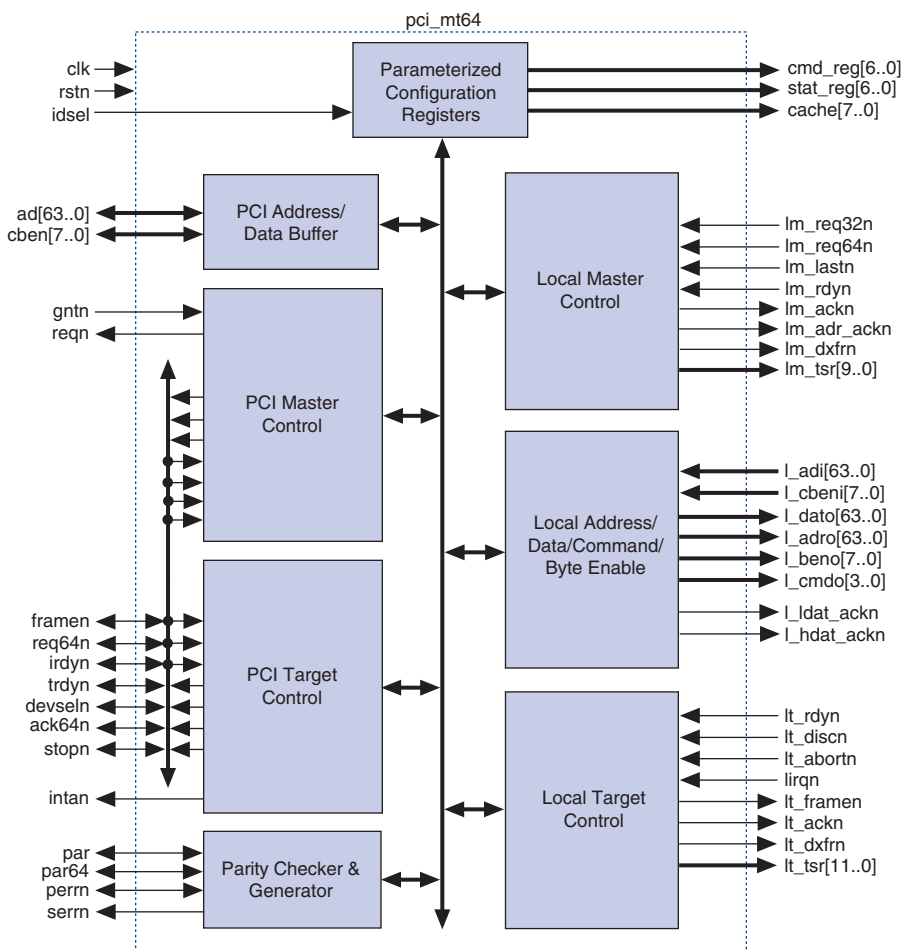
Figure 3–1. *pci_mt64* Functional Block Diagram

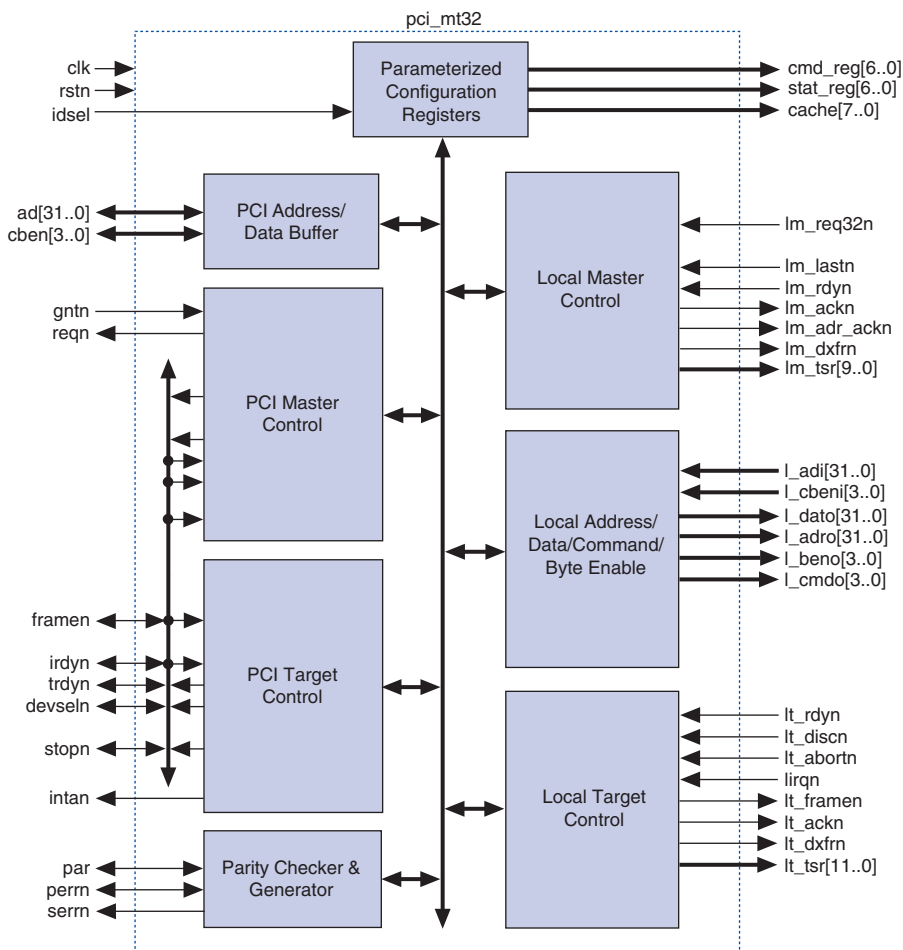
Figure 3–2. *pci_mt32 Functional Block Diagram*

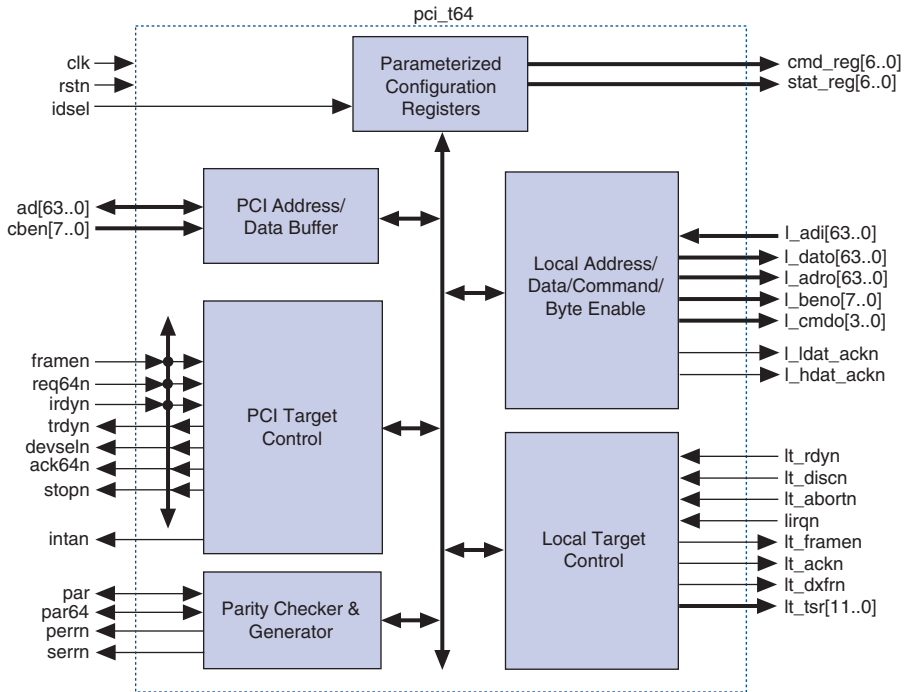
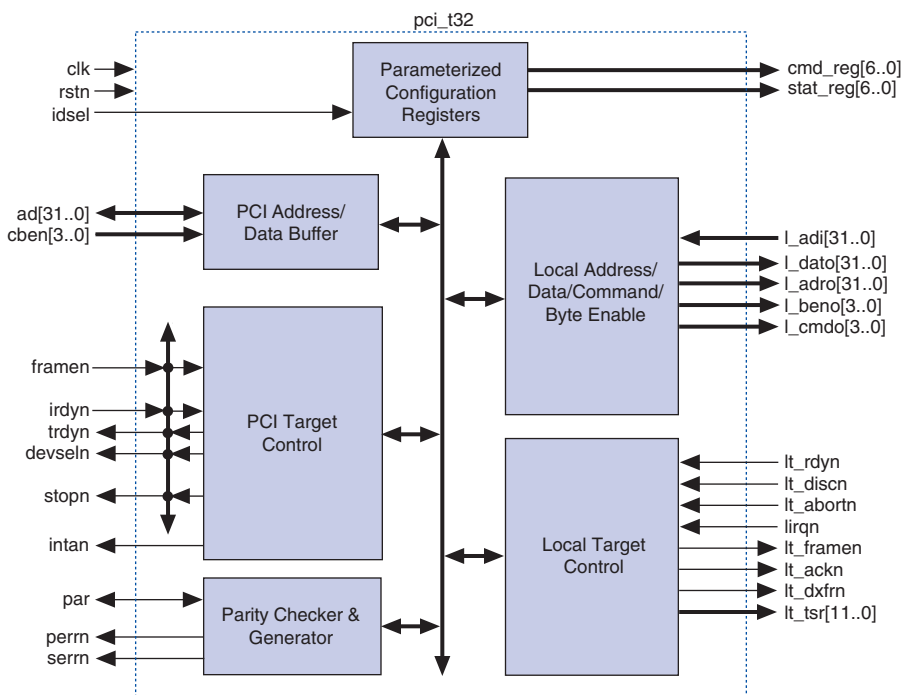
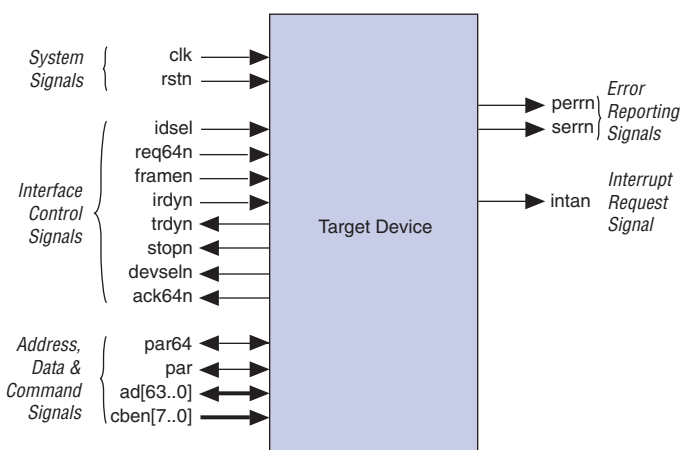
Figure 3–3. pci_t64 Functional Block Diagram

Figure 3–4. pci_t32 Functional Block Diagram

Target Device Signals & Signal Assertion

Figure 3–5 illustrates the signal directions for a PCI device connecting to the PCI bus in target mode. These signals apply to the `pci_mt64`, `pci_t64`, `pci_mt32`, and `pci_t32` functions when they are operating in target mode. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the PCI MegaCore function operating as a target on the PCI bus. The 64-bit extension signals, including `req64n`, `ack64n`, `par64`, `ad[63..32]`, and `cben[7..4]`, are not implemented in the `pci_mt32` and `pci_t32` functions.

Figure 3–5. Target Device Signals



A 32-bit target sequence begins when the PCI master device asserts `framen` and drives the address and the command on the PCI bus. If the address matches one of the base address registers (BARs) in the PCI MegaCore function, it asserts `devseln` to claim the transaction. The master then asserts `irdyn` to indicate to the target device for a read operation that the master device can complete a data transfer, and for a write operation that valid data is on the `ad[31..0]` bus.

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

- The PCI MegaCore function has decoded a valid address for one of its BARs and it accepts the transactions (assert `devseln`)
- The PCI 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 3–1 shows the control signal combinations possible on the PCI bus during a PCI transaction. The PCI MegaCore function processes the PCI signal assertion from the local side. Therefore, the PCI MegaCore function only drives the control signals per the *PCI Local Bus Specification, Revision 3.0*. 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. Refer to “Target Transaction Terminations” on page 3–77 for more details.

Table 3–1. Control Signal Combination Transfer				
Type	devseln	trdyn	stopn	irdyn
Claim transaction	Assert	Don't care	Don't care	Don't care
Retry (1)	Assert	De-Assert	Assert	Don't care
Disconnect with data	Assert	Assert	Assert	Assert
Disconnect without data	Assert	De-assert	Assert	Don't care
Abort (2)	De-assert	De-assert	Assert	Don't care
Successful transfer	Assert	Assert	De-assert	Assert
Target wait state	Assert	De-assert	De-assert	Assert
Master wait state	Assert	Assert	De-assert	De-assert

Notes to Table 3–1:

- (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_mt64` and `pci_t64` functions accept either 32-bit transactions or 64-bit transactions on the PCI side. In both cases, the functions behave as 64-bit agents on the local side. A 64-bit transaction differs from a 32-bit transaction as follows:

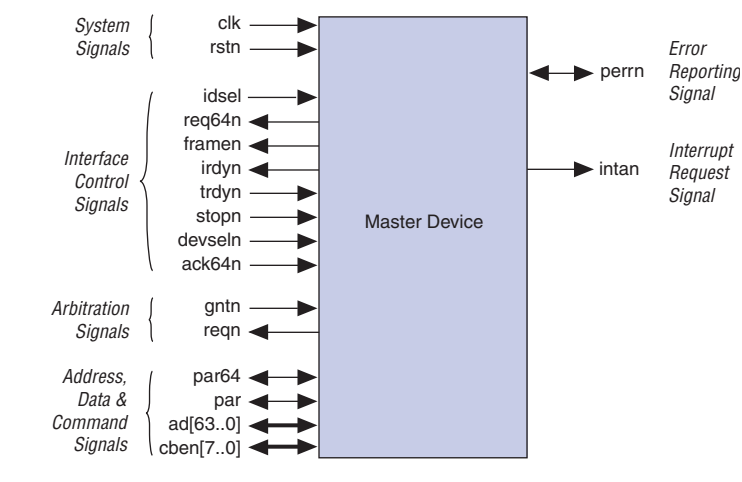
- In addition to asserting the `framen` signal, the PCI master asserts the `req64n` signal during the address phase informing the target device that it is requesting a 64-bit transaction.
- When the target device accepts the 64-bit transaction, it asserts `ack64n` in addition to `devseln` to inform the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

The `pci_mt64`, `pci_t64`, `pci_mt32`, and `pci_t32` functions support unlimited burst access cycles. Therefore, they can achieve a throughput of up to 132 Megabytes per second (MByte/s) for 32-bit, 33-MHz transactions, and up to 528 MByte/s for 64-bit, 66-MHz transactions. However, the *PCI Local Bus Specification, Revision 3.0* does not recommend bursting beyond 16 data cycles because of the latency of other devices that share the bus. You should be aware of the trade-off between bandwidth and increased latency.

Master Device Signals & Signal Assertion

Figure 3–6 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 a PCI MegaCore function operating as a master on the PCI bus. Figure 3–6 shows all master signals. The 64-bit extension signals, including `req64n`, `ack64n`, `par64`, `ad[63..32]`, and `cben[7..4]`, are not implemented in the `pci_mt32` function.

Figure 3–6. Master Device Signals



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



For 64-bit addressing, the master generates a dual-address cycle (DAC). On the first address phase, the `pci_mt64` function drives the lower 32-bit PCI address on `ad[31..0]`, the upper 32-bit PCI address on `ad[63..32]`, the DAC command on `cben[3..0]`, and the transaction command on `cben[7..4]`. On the second address phase, the `pci_mt64` function drives the upper 32-bit PCI address on `ad[31..0]` and the transaction command on `cben[3..0]`.

When the `pci_mt64` or `pci_mt32` function is ready to present or accept data on the bus, it asserts `irdyn`. At this point, the PCI master logic monitors the control signals driven by the target device. The target device decodes the address and command signals presented on the PCI bus during the address phase of the transaction and 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 3–1 shows the possible control signal combinations on the PCI bus during a transaction. The PCI function signals that it is ready to present or accept data on the bus by asserting `irdyn`. At this point, the `pci_mt64` master logic monitors the control signals driven by the target device and asserts its control signals appropriately. The local-side application can use the `lm_tsr[9..0]` signals to monitor the progress of the transaction. The master transaction can be terminated normally or abnormally. The local side signals a normal transaction termination by asserting the `lm_lastn` signal. The abnormal termination can be caused by either a target abort, master abort, or latency timer expiration. Refer to “Abnormal Master Transaction Termination” on page 3–125 for more details.

In addition to single-cycle and burst 32-bit transactions, the local side master can request 64-bit transactions by asserting the `lm_req64n` signal. In 64-bit transactions, the `pci_mt64` function behaves the same as a 32-bit transaction except for asserting the `req64n` signal with the same timing as the `framen` signal. Additionally, the `pci_mt64` function treats the local side as 64 bits when it requests 64-bit transactions and when the target device accepts 64-bit transactions by asserting the `ack64n` signal. Refer to “Master Mode Operation” on page 3–134 for more information on 64-bit master transactions.

PCI Bus Signals

The following PCI signals are used by the `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` 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 shared by multiple devices as a wire-OR. 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.



All of the PCI MegaCore function's logic is clocked by the PCI clock (`clk`). If you are interfacing to logic that has a different clock, you must design appropriate clock domain crossing logic.

Table 3–2 summarizes the PCI bus signals that provide the interface between the PCI MegaCore functions and the PCI bus.

Table 3–2. PCI Interface Signals (Part 1 of 4)			
Name	Type	Polarity	Description
<code>clk</code>	Input	–	Clock. The <code>clk</code> input provides the reference signal for all other PCI interface signals, except <code>rstn</code> and <code>intan</code> .
<code>rstn</code>	Input	Low	Reset. The <code>rstn</code> input initializes the PCI interface circuitry and can be asserted asynchronously to the PCI bus <code>clk</code> edge. When active, the PCI output signals are tri-stated and the open-drain signals, such as <code>serrn</code> , <code>float</code> .
<code>gntn</code>	Input	Low	Grant. The <code>gntn</code> input indicates to the PCI bus master device that it has control of the PCI bus. Every master device has a pair of arbitration signals (<code>gntn</code> and <code>reqn</code>) that connect directly to the arbiter.
<code>l_dis_64_extn</code>	Input	Low	Disable 64-bit extension signals. When you assign an APEX device in a 32-bit PCI bus and drive this signal low, it disables the PCI 64-bit extension signals. The extension signals include <code>ad_63..32_</code> , <code>cben_7..4_</code> , <code>req64n</code> , <code>ack64n</code> , and <code>par64</code> .
<code>reqn</code>	Output	Low	Request. The <code>reqn</code> output indicates to the arbiter that the PCI bus master wants to gain control of the PCI bus to perform a transaction.

Table 3–2. PCI Interface Signals (Part 2 of 4)

Name	Type	Polarity	Description
ad[63..0]	Tri-State	–	Address/data bus. The ad[63..0] bus is a time-multiplexed address/data bus; each bus transaction consists of an address phase followed by one or more data phases. The data phases occur when irdyn and trdyn are both asserted. In the case of a 32-bit data phase, only the ad[31..0] bus holds valid data. For pci_mt32 and pci_t32, only ad[31..0] is implemented.
cben[7..0]	Tri-State	–	Command/byte enable. The cben[7..0] bus is a time-multiplexed command/byte enable bus. During the address phase, this bus indicates the command. During the data phase, this bus indicates byte enables. For pci_mt32 and pci_t32, only cben[3..0] is implemented.
par	Tri-State	–	Parity. The par signal is even parity across the 32 least significant address/data bits and four least significant command/byte enable bits, i.e., the number of 1s on ad[31..0], cben[3..0], and par equal an even number. The par signal is valid one clock cycle after each address phase. For data phases, par is valid one clock cycle after either irdyn asserted on a write transaction or trdyn is asserted on a read transaction. Once par is valid, it remains valid until one clock cycle after the current data phase.
par64	Tri-State	–	Parity 64. The par64 signal is even parity across the 32 most significant address/data bits and the four most significant command/byte enable bits, i.e., the number of 1s on ad[63..32], cben[7..4], and par64 equal an even number. The par64 signal is valid one clock cycle after the address phase where req64n is asserted. For data phases, par64 is valid one clock cycle after either irdyn is asserted on a write transaction or trdyn is asserted on a read transaction. This signal is not implemented in the pci_mt32 and pci_t32 functions.
idsel	Input	High	Initialization device select. The idsel input is a chip select for configuration transactions.
framen (1)	STS	Low	Frame. The framen signal is an output from the current bus master that indicates the beginning and duration of a bus operation. When framen is initially asserted, the address and command signals are present on the ad[63..0] and cben[7..0] buses (ad[31..0] and cben[3..0] only for 32-bit functions). The framen signal remains asserted during the data operation and is deasserted to identify the end of a transaction.

Table 3–2. PCI Interface Signals (Part 3 of 4)

Name	Type	Polarity	Description
req64n (1)	STS	Low	Request 64-bit transfer. The req64n signal is an output from the current bus master and indicates that the master is requesting a 64-bit transaction. req64n has the same timing as framen. This signal is not implemented in pci_mt32 and pci_t32.
irdyn (1)	STS	Low	Initiator ready. The irdyn signal is an output from a bus master to its target and indicates that the bus master can complete the current data transaction. In a write transaction, irdyn indicates that the address bus has valid data. In a read transaction, irdyn indicates that the master is ready to accept data.
devseln (1)	STS	Low	Device select. Target asserts devseln to indicate that the target has decoded its own address and accepts the transaction.
ack64n (1)	STS	Low	Acknowledge 64-bit transfer. The target asserts ack64n to indicate that the target can transfer data using 64 bits. The ack64n has the same timing as devseln. This signal is not implemented in pci_mt32 and pci_t32.
trdyn (1)	STS	Low	Target ready. The trdyn signal is a target output, indicating that the target can complete the current data transaction. In a read operation, trdyn indicates that the target is providing valid data on the address bus. In a write operation, trdyn indicates that the target is ready to accept data.
stopn (1)	STS	Low	Stop. The stopn signal is a target device request that indicates to the bus master to terminate the current transaction. The stopn signal is used in conjunction with trdyn and devseln to indicate the type of termination initiated by the target.
perrn	STS	Low	Parity error. The perrn signal indicates a data parity error. The perrn signal is asserted one clock cycle following the par and par64 signals or two clock cycles following a data phase with a parity error. The PCI MegaCore functions assert the perrn signal if a parity error is detected on the par or par64 signals and the perrn_ena bit (bit 6) in the command register is set. The par64 signal is only evaluated during 64-bit transactions in pci_mt64 and pci_t64 functions. In pci_mt32 and pci_t32, only par is evaluated.
serrn	Open-Drain	Low	System error. The serrn signal indicates system error and address parity error. The PCI MegaCore functions assert serrn if a parity error is detected during an address phase and the serrn_ena enable bit (bit 8) in the command register is set.

Table 3–2. PCI Interface Signals (Part 4 of 4)

Name	Type	Polarity	Description
intan	Open-Drain	Low	Interrupt A. The <code>intan</code> signal is an active-low interrupt to the host and must be used for any single-function device requiring an interrupt capability. The PCI MegaCore functions assert <code>intan</code> only when the local side asserts the <code>lirqn</code> signal and the <code>int_dis</code> bit (bit 10 of the command register) is 0.

Note to Table 3–2:

- (1) In the PCI MegaCore function symbols, the bidirectional control signals are separated into two components: input and output. For example, `framen` has the input `framen_in` and the output `framen_out`. This separation of signals allows the PCI MegaCore function to obtain better slack on set-up times.

Parameterized Configuration Register Signals

Table 3–3 summarizes the PCI local interface signals for the parameterized configuration register signals.

Table 3–3. Parameterized Configuration Register Signals			
Name	Type	Polarity	Description
cache[7..0]	Output	–	Cache line-size register output. The cache[7..0] bus is the same as the configuration space cache line-size register. The local-side logic uses this signal to provide support for cache commands.
cmd_reg[6..0]	Output	–	Command register output. The cmd_reg[6..0] bus drives the important signals of the configuration space command register to the local side. Refer to Table 3–4.
stat_reg[6..0]	Output	–	Status register output. The stat_reg[6..0] bus drives the important signals of the configuration space status register to the local side. Refer to Table 3–5.

Table 3–4 shows definitions for the command register output bus bits.

Table 3–4. PCI Command Register Output Bus (cmd_reg[6..0]) Bit Definition		
Bit Number	Bit Name	Description
0	io_ena	I/O accesses enable. Bit 0 of the command register.
1	mem_ema	Memory access enable. Bit 1 of the command register.
2	mstr_ena	Master enable. Bit 2 of the command register. This signal is reserved for pci_t64 and pci_t32.
3	mwi_ena	Memory write and invalidate enable. Bit 4 of the command register.
4	perr_ena	Parity error response enable. Command register bit 6.
5	serr_ena	System error response enable. Command register bit 8.
6	int_dis (1)	Interrupt disable. Command register bit 10.

Note to Table 3–4:

(1) This signal is added for compliance with the *PCI Local Bus Specification, Revision 3.0*.

Table 3–5 shows definitions for the PCI status register bits.

Table 3–5. PCI Status Register Output Bus (stat_reg[6..0]) Bit Definition		
Bit Number	Bit Name	Description
0	perr_rep	Parity error reported. Status register bit 8.
1	tabort_sig	Target abort signaled. Status register bit 11.
2	tabort_rcvd	Target abort received. Status register bit 12.
3	mabort_rcvd	Master abort received. Status register bit 13.
4	serr_sig	Signaled system error. Status register bit 14.
5	perr_det	Parity error detected. Status register bit 15.
6	int_stat (1)	Interrupt status. Status register bit 3.

Note to Table 3–5:

(1) This signal is added for compliance with the *PCI Local Bus Specification, Revision 3.0*.

Local Address, Data, Command, & Byte Enable Signals

Table 3–6 summarizes the PCI local interface signals for the address, data, command, and byte enable signals.

Table 3–6. PCI Local Address, Data, Command & Byte Enable Signals (Part 1 of 3)

Name	Type	Polarity	Description
<code>l_adi[63..0]</code>	Input	–	Local address/data input. This bus is a local-side time multiplexed address/data bus. This bus changes operation depending on the function you are using and the type of transaction. During master transactions, the local side must provide the address on <code>l_adi[63..0]</code> when <code>lm_adr_ackn</code> is asserted. For 32-bit addressing, only the <code>l_adi[31..0]</code> signals are valid during the address phase. The <code>l_adi[63..0]</code> bus is driven active by the local-side logic during PCI bus-initiated target read transactions or local-side initiated master write transactions. For <code>pci_mt32</code> and <code>pci_t32</code>, only <code>l_adi[31..0]</code> is used. For the <code>pci_mt64</code> and <code>pci_t64</code> functions, the entire <code>l_adi[63..0]</code> bus is used to transfer data from the local side during 64-bit master write and 64-bit and 32-bit target read transactions.
<code>l_cbeni[7..0]</code>	Input	–	Local command/byte enable input. This bus is a local-side time multiplexed command/byte enable bus. During master transactions, the local side must provide the command on <code>l_cbeni[3..0]</code> when <code>lm_adr_ackn</code> is asserted. For 64-bit addressing, the local side must provide the DAC command (B"1101") on <code>l_cbeni[3..0]</code> and the transaction command on <code>l_cbeni[7..4]</code> when <code>lm_tsr[1]</code> is asserted. The local side must provide the command with the same encoding as specified in the <i>PCI Local Bus Specification, Revision 3.0</i>. The local-master device drives byte enables on the <code>l_cbeni[7..0]</code> bus during master transactions. The local master device must provide the byte-enable value on <code>l_cbeni[7..0]</code> during the next clock cycle after <code>lm_adr_ackn</code> is asserted. This is the same clock cycle that immediately follows a local side address phase. The PCI MegaCore functions drive the byte-enable value from the local side to the PCI side. The PCI MegaCore function maintains the same byte enables that were provided with the initial data word on the local side throughout the burst transaction. The PCI MegaCore function allows variable byte enable values from the local side to the PCI side if Allow Variable Byte Enables During Burst Transaction is turned on in the Parameterize - PCI Compiler wizard. Refer to “Advanced PCI MegaCore Function Features” on page 2–3 for more information. In <code>pci_mt32</code>, only <code>l_cbeni[3..0]</code> is implemented. Additionally, in <code>pci_mt64</code>, only <code>l_cbeni[3..0]</code> is used when a 32-bit master transaction is initiated.

Table 3–6. PCI Local Address, Data, Command & Byte Enable Signals (Part 2 of 3)

Name	Type	Polarity	Description
<code>l_adro[63..0]</code>	Output	–	Local address output. The <code>l_adro[63..0]</code> bus is driven by the PCI MegaCore functions during target transactions. The <code>pci_mt32</code> and <code>pci_t32</code> functions only implement <code>l_adro[31..0]</code>. During dual address transactions in the <code>pci_mt64</code> and <code>pci_t64</code> MegaCore functions, the <code>l_adro[63..32]</code> bus is driven with a valid address. DAC is indicated by the assertion of <code>lt_tsr[11]</code>. For more information on the local target status signals, refer to Table 3–8. The falling edge of <code>lt_framen</code> indicates a valid <code>l_adro[63..0]</code>. The PCI address is held at the local side as long as possible and should be assumed invalid at the end of the target transaction on the PCI bus. The end of the target transaction is indicated by <code>lt_tsr[8]</code> (<code>targ_access</code>) being deasserted.
<code>l_dato[63..0]</code>	Output	–	Local data output. The <code>l_dato[63..0]</code> bus is driven active during PCI bus-initiated target write transactions or local side-initiated master read transactions. The functionality of this bus changes depending on the function you are using and the transaction being considered. The <code>pci_mt32</code> and <code>pci_t32</code> functions implement only <code>l_dato[31..0]</code>. The operation in the <code>pci_mt64</code> and <code>pci_t64</code> MegaCore functions is dependent on the type of transaction being considered. During 64-bit target write transactions and master read transactions, the data is transferred on the entire <code>l_dato[63..0]</code> bus. During 32-bit master read transactions, the data is only transferred on <code>l_dato[31..0]</code>. During 32-bit target write transactions, the data is also only transferred on <code>l_dato[31..0]</code>; however, depending on the transaction address, the <code>pci_mt64</code> or <code>pci_t64</code> MegaCore function either asserts <code>l_ldat_ackn</code> or <code>l_hdat_ackn</code> to indicate whether the address for the current data word is a QWORD boundary (<code>ad[2..0] = B"000"</code>) or not.
<code>l_beno[7..0]</code>	Output	–	Local byte enable output. The <code>l_beno[7..0]</code> bus is driven by the PCI function during target transactions. This bus holds the byte enable value during data transfers. The functionality of this bus is different depending on the function being used and the transaction being considered. The <code>pci_mt32</code> and <code>pci_t32</code> functions implement only <code>l_beno[3..0]</code>. The operation in the <code>pci_mt64</code> and <code>pci_t64</code> MegaCore functions is dependent on the type of transaction being considered. During 64-bit target write transactions, the byte enables are transferred on the entire <code>l_beno[7..0]</code> bus. During 32-bit target write transactions, the byte enables are transferred on the <code>l_beno[3..0]</code> bus and, depending on the transaction address, the <code>pci_mt64</code> or <code>pci_t64</code> MegaCore function either asserts <code>l_ldat_ackn</code> or <code>l_hdat_ackn</code> to indicate whether the address for the current byte enables is at a QWORD boundary (<code>ad[2..0] = B"000"</code>) or not.

Table 3–6. PCI Local Address, Data, Command & Byte Enable Signals (Part 3 of 3)

Name	Type	Polarity	Description
<code>l_cmdo[3..0]</code>	Output	–	Local command output. The <code>l_cmdo[3..0]</code> bus is driven by the PCI MegaCore functions during target transactions. It has the bus command and the same timing as the <code>l_adro[31..0]</code> bus. The command is encoded as presented on the PCI bus.
<code>l_ldat_ackn</code>	Output	Low	Local low data acknowledge. The <code>l_ldat_ackn</code> output is used during target write and master read transactions. When asserted, <code>l_ldat_ackn</code> indicates that the least significant DWORD is being transferred on the <code>l_dato[31..0]</code> bus, i.e., when <code>l_ldat_ackn</code> is asserted, the address of the transaction is on a QWORD boundary (<code>ad[2..0] = B"000"</code>). The signals <code>lm_ackn</code> or <code>lt_ackn</code> must be used to qualify valid data. During target read transactions, <code>l_ldat_ackn</code> is used to indicate the first DWORD transferred to the PCI side. If the address of the transaction is a QWORD boundary, the <code>l_ldat_ackn</code> signal is asserted. This signal is not implemented in the <code>pci_mt32</code> and <code>pci_t32</code> functions.
<code>l_hdat_ackn</code>	Output	Low	Local high data acknowledge. The <code>l_hdat_ackn</code> output is used during target write and master read transactions. When asserted, <code>l_hdat_ackn</code> indicates that the most significant DWORD is being transferred on the <code>l_dato[31..0]</code> bus. In other words, when <code>l_hdat_ackn</code> is asserted, the address of the transaction is not a QWORD boundary (<code>ad[2..0] = B"100"</code>). The signals <code>lm_ackn</code> or <code>lt_ackn</code> must be used to qualify valid data. During target read transactions, <code>l_hdat_ackn</code> is used to indicate the first DWORD transferred to the PCI side. If the address of the transaction is not a QWORD boundary, <code>l_ldat_ackn</code> is deasserted and <code>l_hdat_ackn</code> is asserted. This signal is not implemented in the <code>pci_mt32</code> and <code>pci_t32</code> functions.

Target Local-Side Signals

Table 3–7 summarizes the target interface signals that provide the interface between the PCI MegaCore function and the local-side peripheral device(s) during target transactions.



When a local side transaction is not in progress, local side inputs should be driven to the deasserted state.

Table 3–7. Target Signals Connecting to the Local Side (Part 1 of 3)

Name	Type	Polarity	Description
lt_abortn	Input	Low	Local target abort request. The local side should assert this signal requesting the PCI MegaCore function to issue a target abort to the PCI master. The local side should request an abort when it has encountered a fatal error and cannot complete the current transaction.
lt_discn	Input	Low	Local target disconnect request. The lt_discn input requests the PCI MegaCore function to issue a retry or a disconnect. The PCI MegaCore function issues a retry or disconnect depending on when the signal is asserted during a transaction. The PCI bus specification requires that a PCI target issues a disconnect whenever the transaction exceeds its memory space. When using PCI MegaCore functions, the local side is responsible for asserting lt_discn if the transaction crosses its memory space.

Table 3–7. Target Signals Connecting to the Local Side (Part 2 of 3)

Name	Type	Polarity	Description
lt_rdyn	Input	Low	Local target ready. The local side asserts lt_rdyn to indicate a valid data input during target read, or ready to accept data input during a target write. During a target read, lt_rdyn deassertion suspends the current transfer (i.e., a wait state is inserted by the local side). During a target write, an inactive lt_rdyn signal directs the PCI MegaCore function to insert wait states on the PCI bus. The only time the function inserts wait states during a burst is when lt_rdyn inserts wait states on the local side. lt_rdyn is sampled one clock cycle before actual data is transferred on the local side. During target write transactions, lt_rdyn has also special functionality. To allow the local side ample time to issue a retry for the write cycle, the PCI MegaCore function does not assert trdyn in the first data phase unless the local side asserts lt_rdyn. In this case, the local side asserts lt_rdyn to indicate that it intends to complete at least one data phase and it is not going to issue a retry. Refer to the “Additional Design Guidelines for Target Transactions” on page 3–88 section for additional information about the lt_rdyn functionality.
lt_framen	Output	Low	Local target frame request. The lt_framen output is asserted while the PCI MegaCore function is requesting access to the local side. It is asserted one clock cycle before the function asserts devseln, and it is released after the last data phase of the transaction is transferred to/from the local side.
lt_ackn	Output	Low	Local target acknowledge. The PCI 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 delay the bursting operation. During a target write, lt_ackn de-assertion suspends the current transfer (i.e., a wait state is inserted by the PCI master). The lt_ackn signal is only inactive during a burst when the PCI bus master inserts wait states.
lt_dxfrn	Output	Low	Local target data transfer. The PCI MegaCore function asserts the lt_dxfrn signal when a data transfer on the local side is successful during a target transaction.

Table 3–7. Target Signals Connecting to the Local Side (Part 3 of 3)

Name	Type	Polarity	Description
lt_tsr[11..0]	Output	–	Local target transaction status register. The lt_tsr[11..0] bus carries several signals which can be monitored for the transaction status. Refer to Table 3–8 .
lirqn	Input	Low	Local interrupt request. The local-side peripheral device asserts lirqn to signal a PCI bus interrupt. Asserting this signal forces the PCI MegaCore function to assert the intan signal for as long as the lirqn signal is asserted and the int_dis bit (bit 10 of the command register) is 0.

Table 3–8 shows definitions for the local target transaction status register outputs.

Table 3–8. Local Target Transaction Status Register (lt_tsr[11..0]) Bit Definition		
Bit Number	Bit Name	Description
5..0	bar_hit[5..0]	Base address register hit. Asserting bar_hit[5..0] indicates that the PCI address matches that of a base address register and that the PCI MegaCore function has claimed the transaction. Each bit in the bar_hit[5..0] bus is used for the corresponding base address register (e.g., bar_hit[0] is used for BAR0). When BAR0 and BAR1 are used to implement a 64-bit base address register, bar_hit[0] and bar_hit[1] are asserted to indicate that the pci_mt64 and pci_t64 MegaCore functions have claimed the transaction. When BAR1 and BAR2 are used to implement a 64-bit base address register, bar_hit[1] and bar_hit[2] are asserted to indicate that the pci_mt64 and pci_t64 MegaCore functions have claimed the transaction.
6	exp_rom_hit	Expansion ROM register hit. The PCI MegaCore function asserts this signal when the transaction address matches the address in the expansion ROM BAR.
7	trans64bit	64-bit target transaction. The pci_mt64 and pci_t64 assert this signal when the current transaction is 64 bits. If a transaction is active and this signal is low, the current transaction is 32 bits. This bit is reserved for pci_mt32 and pci_t32.
8	targ_access	Target access. The PCI MegaCore function asserts this signal when a PCI target access is in progress.
9	burst_trans	Burst transaction. When asserted, this signal indicates that the current target transaction is a burst. This signal is asserted if the PCI MegaCore function detects both framen and irdyn signals asserted at the same time during the first data phase.
10	pci_xfr	PCI transfer. This signal is asserted to indicate that there was a successful data transfer on the PCI side during the previous clock cycle.
11	dac_cyc	Dual address cycle. When asserted, this signal indicates that the current transaction is using a dual address cycle.

Master Local-Side Signals

Table 3–9 summarizes the `pci_mt64` and `pci_mt32` master interface signals that provide the interface between the PCI MegaCore function and the local-side peripheral device(s) during master transactions.



When a local side transaction is not in progress, local side inputs should be deasserted.

Table 3–9. PCI Master Signals Interfacing to the Local Side (Part 1 of 2)

Name	Type	Polarity	Description
<code>lm_req32n</code>	Input	Low	Local master request 32-bit data transaction. The local side asserts this signal to request ownership of the PCI bus for a 32-bit master transaction. To request a master transaction, it is sufficient for the local-side device to assert <code>lm_req32n</code> for one clock cycle. When requesting a 32-bit transaction, only <code>l_adi[31..0]</code> for a master write transaction or <code>l_dat0[31..0]</code> for a master read transaction is valid. The local side cannot request the bus until the current master transaction has completed. After being granted mastership of the PCI bus, the <code>lm_req32n</code> signal should be asserted only after <code>lm_tsr[3]</code> is deasserted.
<code>lm_req64n</code>	Input	Low	Local master request 64-bit data transaction. The local side asserts this signal to request ownership of the PCI bus for a 64-bit master transaction. To request a master transaction, it is sufficient for the local side device to assert <code>lm_req64n</code> for one clock cycle. When requesting a 64-bit data transaction, <code>pci_mt64</code> requests a 64-bit PCI transaction. When the target does not assert its <code>ack64n</code> signal, the transaction will be 32 bits. In a 64-bit master write transaction where the target does not assert its <code>ack64n</code> signal, <code>pci_mt64</code> automatically accepts 64-bit data on the local side and multiplexes the data appropriately to 32 bits on the PCI side. When the local side requests 64-bit PCI transactions, it must ensure that the address is at a QWORD boundary. This signal is not implemented in <code>pci_mt32</code>. The local side cannot request the bus until the current master transaction has completed. After being granted mastership of the PCI bus, the <code>lm_req64n</code> signal should be asserted only after <code>lm_tsr[3]</code> is deasserted.
<code>lm_lastn</code>	Input	Low	Local master last. This signal is driven by the local side to request that the <code>pci_mt64</code> or <code>pci_mt32</code> master interface ends the current transaction. When the local side asserts this signal, the PCI MegaCore function master interface deasserts <code>framen</code> as soon as possible and asserts <code>irdyn</code> to indicate that the last data phase has begun. The local side must assert this signal for one clock cycle to initiate the end of the current master transaction.

Table 3–9. PCI Master Signals Interfacing to the Local Side (Part 2 of 2)

Name	Type	Polarity	Description
lm_rdyn	Input	Low	Local master ready. The local side asserts the lm_rdyn signal to indicate a valid data input during a master write, or ready to accept data during a master read. During a master write, the lm_rdyn signal de-assertion suspends the current transfer (i.e., wait state is inserted by the local side). During a master read, an inactive lm_rdyn signal directs pci_mt64 or pci_mt32 to insert wait states on the PCI bus. The only time pci_mt64 or pci_mt32 inserts wait states during a burst is when the lm_rdyn signal inserts wait states on the local side. The lm_rdyn signal is sampled one clock cycle before actual data is transferred on the local side.
lm_adr_ackn	Output	Low	Local master address acknowledge. pci_mt64 or pci_mt32 assert the lm_adr_ackn signal to the local side to acknowledge the requested master transaction. During the same clock cycle when lm_adr_ackn is asserted low, the local side must provide the transaction address on the l_adi[31..0] bus and the transaction command on the l_cmdi[3..0] bus.
lm_ackn	Output	Low	Local master acknowledge. pci_mt64 and pci_mt32 assert the lm_ackn signal to indicate valid data output during a master read, or ready to accept data during a master write. During a master write, an inactive lm_ackn signal indicates that pci_mt64 and pci_mt32 is not ready to accept data, and local logic should hold off the bursting operation. During a master read operation, the lm_ackn signal de-assertion suspends the current transfer (i.e., a wait state is inserted by the PCI target). During a burst when the PCI bus target inserts wait states, the lm_ackn signal goes inactive.
lm_dxfrn	Output	Low	Local master data transfer. During a master transaction, pci_mt64 and pci_mt32 assert this signal when a data transfer on the local side is successful.
lm_tsr[9..0]	Output	–	Local master transaction status register bus. These signals inform the local interface of the transaction's progress. Refer to Table 3–10 for a detailed description of the bits in this bus.

Table 3–10 shows definitions for the local master transaction status register outputs.

Table 3–10. <i>pci_mt64 & pci_mt32 Local Master Transaction Status Register (lm_tsr[9..0]) Bit Definition (1)</i>		
Bit Number	Bit Name	Description
0	request	Request. This signal indicates that the <code>pci_mt64</code> or <code>pci_mt32</code> function is requesting mastership of the PCI bus (i.e., it is asserting its <code>reqn</code> signal). The <code>request</code> bit is not asserted if the following is true: The PCI bus arbiter has parked on the <code>pci_mt64</code> or <code>pci_mt32</code> function and the <code>gntn</code> signal is already asserted when the function requests mastership of the bus.
1 (1)	grant	Grant. This signal is active after the <code>pci_mt64</code> or <code>pci_mt32</code> function has detected that <code>gntn</code> is asserted.
2 (1)	adr_phase	Address phase. This signal is active during a PCI address phase where <code>pci_mt64</code> or <code>pci_mt32</code> is the bus master.
3	dat_phase	Data phase. This signal is active while the <code>pci_mt64</code> or <code>pci_mt32</code> function is in data transfer mode. The signal is active after the address phase and remains active until the turn-around state begins.
4	lat_exp	Latency timer expired. This signal indicates that <code>pci_mt64</code> or <code>pci_mt32</code> terminated the master transaction because the latency timer counter expired.
5	retry	Retry detected. This signal indicates that the <code>pci_mt64</code> or <code>pci_mt32</code> function terminated the master transaction because the target issued a retry. Per the PCI specification, a transaction that ends in a retry must be retried at a later time.
6	disc_wod	Disconnect without data detected. This signal indicates that the <code>pci_mt64</code> or <code>pci_mt32</code> signal terminated the master transaction because the target issued a disconnect without data.
7	disc_wd	Disconnect with data detected. This signal indicates that <code>pci_mt64</code> or <code>pci_mt32</code> terminated the master transaction because the target issued a disconnect with data.
8	dat_xfr	Data transfer. This signal indicates that a successful data transfer occurred on the PCI side in the preceding clock cycle. This signal can be used by the local side to keep track of how much data was actually transferred on the PCI side.
9	trans64	64-bit transaction. This signal indicates that the target claiming the transaction asserted its <code>ack64n</code> signal. Because <code>pci_mt32</code> does not request 64-bit transactions, this signal is reserved.

Note to Table 3–10:

- (1) Some arbiters may initially assert `gntn` (in response to either the `pci_mt64` or `pci_mt32` function requesting mastership of the PCI bus), but then deassert `gntn` (before the `pci_mt64` or `pci_mt32` have asserted `framen`) to give mastership of the bus to a higher priority device. In systems where this situation may occur, the local side logic should hold the address and command on the `l_adi[63..0]` and `l_cbeni[7..0]` buses until the `adr_phase` bit is asserted (`lm_tsr[2]`) to ensure that the `pci_mt64` or `pci_mt32` function has assumed mastership of the bus and that the current address and command bits have been transferred.

PCI Bus Commands

Table 3–11 shows the PCI bus commands that can be initiated or responded to by the PCI MegaCore functions.

Table 3–11. PCI Bus Command Support Summary			
cben[3..0] Value	Bus Command Cycle	Master	Target
0000	Interrupt acknowledge	No	Yes (1)
0001	Special cycle	No	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 (2)	Yes	Yes
1101	Dual address cycle (DAC)	Yes (3)	Yes (3)
1110	Memory read line (2)	Yes	Yes
1111	Memory write and invalidate (2)	Yes	Yes

Notes to Table 3–11:

- (1) Interrupt acknowledge support can be enabled on the **Advanced PCI MegaCore Function Features** page of the **Parameterize - PCI Compiler** wizard. When support is enabled, the target accepts the interrupt acknowledge command and aliases it as a memory read command.
- (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 `l_cmdo[3..0]` bus with the encoding shown in Table 3–11.
- (3) This command is not supported by the `pci_mt32` and `pci_t32` MegaCore functions.

During the address phase of a transaction, the `cben[3..0]` bus is used to indicate the transaction type (Table 3–11).

The PCI MegaCore functions respond to standard memory read/write, cache-line memory read/write, I/O read/write, and configuration read/write commands. The bus commands are discussed in greater detail in “Target Mode Operation” on page 3–44 and “Master Mode Operation” on page 3–88.

In master mode, the `pci_mt64` and `pci_mt32` functions 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 memory read line (MRL) and memory write-and-invalidate (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 3.0* and the *Compliance Checklist, Revision 3.0*. 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 Altera PCI MegaCore functions.

Table 3–12 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 MegaCore functions.

Table 3–12. PCI Bus Configuration Registers				
Address	Byte			
	3	2	1	0
0x00	Device ID		Vendor ID	
0x04	Status Register		Command Register	
0x08	Class Code			Revision ID
0x0C	BIST	Header Type	Latency Timer	Cache Line Size
0x10	Base Address Register 0			
0x14	Base Address Register 1			
0x18	Base Address Register 2			
0x1C	Base Address Register 3			
0x20	Base Address Register 4			
0x24	Base Address Register 5			
0x28	Card Bus CIS Pointer			
0x2C	Subsystem ID		Subsystem Vendor ID	
0x30	Expansion ROM Base Address Register			
0x34	Reserved			Capabilities Pointer
0x38	Reserved			
0x3C	Maximum Latency	Minimum Grant	Interrupt Pin	Interrupt Line

Table 3–13 summarizes the 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 run time, i.e., from the perspective of other PCI bus agents. You can set some of the read-only registers when creating a custom PCI design by using the IP Toolbench **Parameterize - PCI Compiler** wizard. For example, you can change the

device ID register value on the **Read-Only PCI Configuration Registers** page. The specified default state is defined as the state of the register when the PCI bus is reset.

Table 3–13. Supported Configuration Registers Address Map

Address Offset	Range Reserved	Bytes Used/Reserved	Read/Write	Mnemonic	Register Name
0x00	0x00-0x01	2/2	Read	ven_id	Vendor ID
0x02	0x02-0x03	2/2	Read	dev_id	Device ID
0x04	0x04-0x05	2/2	Read/write	comd	Command
0x06	0x06-0x07	2/2	Read/write	status	Status
0x08	0x08-0x08	1/1	Read	rev_id	Revision ID
0x09	0x09-0x0B	3/3	Read	class	Class code
0x0C	0x0C-0x0C	1/1	Read/write	cache	Cache line size (1)
0x0D	0x0D-0x0D	1/1	Read/write	lat_tmr	Latency timer (1)
0x0E	0x0E-0x0E	1/1	Read	header	Header type
0x10	0x10-0x13	4/4	Read/write	bar0	Base address register zero
0x14	0x14-0x17	4/4	Read/write	bar1	Base address register one
0x18	0x18-0x1B	4/4	Read/write	bar2	Base address register two
0x1C	0x1C-0x1F	4/4	Read/write	bar3	Base address register three
0x20	0x20-0x23	4/4	Read/write	bar4	Base address register four
0x24	0x24-0x27	4/4	Read/write	bar5	Base address register five
0x28	0x28-0x2B	4/4	Read	cardbus_ptr	CardBus CIS pointer
0x2C	0x2C-0x2D	2/2	Read	sub_ven_id	Subsystem vendor ID
0x2E	0x2E-0x2F	2/2	Read	sub_id	Subsystem ID
0x30	0x30-0x33	4/4	Read/write	exp_rom_bar	Expansion ROM BAR
0x34	0x34-0x35	1/1	Read	cap_ptr	Capabilities pointer
0x3C	0x3C-0x3C	1/1	Read/write	int_ln	Interrupt line
0x3D	0x3D-0x3D	1/1	Read	int_pin	Interrupt pin
0x3E	0x3E-0x3E	1/1	Read	min_gnt	Minimum grant (1)
0x3F	0x3F-0x3F	1/1	Read	max_lat	Maximum latency (1)

Note to Table 3–13:

(1) These registers are supported by the pci_mt64 and pci_mt32 MegaCore functions only.

Vendor ID Register

Vendor ID is a 16-bit read-only register that identifies the manufacturer of the device. The value of this register is assigned by the PCI SIG; the default value of this register is the Altera vendor ID value, which is 0x1172. However, by setting the `ven_id` value through the wizard, you can change the value of the vendor ID register to your PCI SIG-assigned vendor ID value. Refer to [Table 3–14](#).

Table 3–14. Vendor ID Register Format			
Data Bit	Mnemonic	Read/Write	Definition
15..0	<code>ven_id</code>	Read	PCI vendor ID

Device ID Register

Device ID is a 16-bit read-only register that identifies the device type. The value of this register is assigned by the manufacturer. The default value of the device ID register is 0x0004. You can change the value of the device ID register through the wizard. Refer to [Table 3–15](#).

Table 3–15. Device ID Register Format			
Data Bit	Mnemonic	Read/Write	Definition
15..0	<code>dev_id</code>	Read	Device ID

Command Register

Command is a 16-bit read/write register that provides basic control over the ability of the PCI function to respond to the PCI bus and/or access it. Refer to [Table 3–16](#).

Table 3–16. Command Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	io_ena	Read/write	I/O access enable. When high, io_ena lets the function respond to the PCI bus I/O accesses as a target.
1	mem_ena	Read/write	Memory access enable. When high, mem_ena lets the function respond to the PCI bus memory accesses as a target.
2	mstr_ena	Read/write	Master enable. When high, mstr_ena allows the function to request mastership of the PCI bus. Bit 2 is hardwired to 1 when PCI master host bridge options are enabled through the wizard.
3	Unused	—	—
4	mwi_ena	Read/write	Memory write and invalidate enable. This bit controls whether the master may generate a MWI command. Although the 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 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 function to report address parity errors via the serrn output. However, to signal a system error, the perr_ena bit must also be high.
9	Unused	—	—
10	int_dis	Read/write	Interrupt disable. A value of 1 disables the PCI MegaCore function from asserting intan on the PCI bus. However, the interrupt is only disabled after the preexisting interrupt has been serviced.
15..11	Unused	—	—

Status Register

Status is a 16-bit register that provides the status of bus-related events. Read transactions from the status register behave normally. However, status register 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 0x4000 to the status register clears bit 14 and leaves the rest of the bits unchanged. The default value of the status register is 0x0400. Refer to [Table 3–17](#).

Table 3–17. Status Register Format (Part 1 of 2)

Data Bit	Mnemonic	Read/Write	Definition
2..0	Unused	–	Reserved.
3	int_stat	Read	Interrupt status. This bit is read only and is set when the <code>int_dis</code> bit (bit 10 of the command register) is 0 and <code>intan</code> is asserted on the PCI bus. This signal is driven to the local side on the <code>stat_reg[6]</code> output.
4	cap_list_ena	Read	Capabilities list enable. This bit is read only and is set by the user when enabling the Capabilities List Pointer through the wizard. When set, this bit enables the capabilities list pointer register at offset 0x34. Refer to “ Capabilities Pointer ” on page 3–42 for more details.
5	pci_66mhz_capable	Read	PCI 66-MHz capable. When set, <code>pci_66mhz_capable</code> indicates that the PCI device is capable of running at 66 MHz. The PCI MegaCore functions can function at either 66 MHz or 33 MHz depending on the device used. You can set this bit to 1 by turning on PCI 66MHz Capable on the initial page of the IP Toolbench Parameterize - PCI Compiler wizard.
7..6	Unused	–	Reserved.
8	dat_par_rep	Read/write	Reported data parity. When high, <code>dat_par_rep</code> indicates that during a read transaction the function asserted the <code>perrn</code> output as a master device, or that during a write transaction the <code>perrn</code> output was asserted as a target device. This bit is high only when the <code>perr_ena</code> bit (bit 6 of the command register) is also high. This signal is driven to the local side on the <code>stat_reg[0]</code> output.
10..9	devsel_tim	Read	Device select timing. The <code>devsel_tim</code> bits indicate target access timing of the function via the <code>devseln</code> output. The PCI MegaCore functions are designed to be slow target devices (i.e., <code>devsel_tim</code> = B"10").

Table 3–17. Status Register Format (Part 2 of 2)

Data Bit	Mnemonic	Read/Write	Definition
11	tabort_sig	Read/write	Signaled target abort. This bit is set when a local peripheral device terminates a transaction. The function automatically sets this bit if it issued a target abort after the local side asserted <code>lt_abortn</code> . This bit is driven to the local side on the <code>stat_reg[1]</code> output.
12	tar_abrt_rec	Read/write	Target abort. When high, <code>tar_abrt_rec</code> indicates that the function in master mode has detected a target abort from the current target device. This bit is driven to the local side on the <code>stat_reg[2]</code> output.
13	mstr_abrt	Read/write	Master abort. When high, <code>mstr_abrt</code> indicates that the function in master mode has terminated the current transaction with a master abort. This bit is driven to the local side on the <code>stat_reg[3]</code> output.
14	serr_set	Read/write	Signaled system error. When high, <code>serr_set</code> indicates that the function drove the <code>serrn</code> output active, i.e., an address phase parity error has occurred. The function signals a system error only if an address phase parity error was detected and <code>serr_ena</code> was set. This signal is driven to the local side on the <code>stat_reg[4]</code> output.
15	det_par_err	Read/write	Detected parity error. When high, <code>det_par_err</code> indicates that the function detected either an address or data parity error. Even if parity error reporting is disabled (via <code>perr_ena</code>), the function sets the <code>det_par_err</code> bit. This signal is driven to the local side on the <code>stat_reg[5]</code> output.

Revision ID Register

Revision ID is an 8-bit read-only register that identifies the revision number of the device. The value of this register is assigned by the manufacturer (e.g., Altera for the PCI MegaCore functions.) For the Altera PCI MegaCore functions, the default value of the revision ID register is the revision number of the function. Refer to [Table 3–18](#). You can change the value of the revision ID register through the wizard.

Table 3–18. 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 3.0* for detailed bit information. The default value of the class code register is 0xFF0000. You can change the value of the `class_code` register using the **Parameterize - PCI Compiler** wizard. Refer to [Table 3–19](#).

Table 3–19. Class Code Register Format			
Data Bit	Mnemonic	Read/Write	Definition
23..0	<code>class_code</code>	Read	Class code

Cache Line Size Register

The cache line size register specifies the system cache line size in DWORDs. This read/write register is written by system software at power-up. The value in this register is driven to the local side on the `cache [7..0]` bus. The local side must use this value when using the memory read line, memory read multiple, and memory write and invalidate commands in master mode. Refer to [Table 3–20](#).



This register is implemented in the `pci_mt64` and `pci_mt32` functions only.

Table 3–20. Cache Line Size Register Format			
Data Bit	Mnemonic	Read/Write	Definition
7..0	<code>cache</code>	Read/write	Cache line size

Latency Timer Register

The latency timer register is an 8-bit register with bits 2, 1, and 0 tied to ground. The register defines the maximum amount of time, in PCI bus clock cycles, that the PCI function can retain ownership of the PCI bus. After initiating a transaction, the function decrements its latency timer by one on the rising edge of each clock cycle. The default value of the latency timer register is 0x00. Refer to [Table 3–21](#).



This register is implemented in the `pci_mt64` and `pci_mt32` functions only.

Table 3–21. 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 function as a single-function device. The default value of the header type register is 0x00. Refer to [Table 3–22](#).

Table 3–22. Header Type Register Format

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

Base Address Registers

The PCI function supports up to six BARs. Each BAR (BAR n) has identical attributes. Use the **Parameterize - PCI Compiler** wizard to instantiate BARs in the function on an individual basis. BARs must be used in sequence, starting with BAR0; one or more of the BARs in the function must be instantiated. The logic for the unused BARs is automatically reduced by the Quartus II software when the PCI function is compiled.

Each BAR has its own parameter BAR n (where n is the BAR number). The value for this parameter is a 32-bit hexadecimal number that can be updated through the wizard to select a combination of the following BAR options:

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



When compiling the PCI function, the Quartus 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 3.0*. 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 through the wizard.

In a memory BAR, bits 1 and 2 indicate the location of the address space in the memory map. You can control the location of specific BAR addresses (i.e., whether they are mapped in 32- or 64-bit address space) through options in the wizard. The `pci_mt64` and `pci_t64` functions allow you to implement a 64-bit BAR using BAR1 and BAR0, or by using BAR2 and BAR1. The BAR n parameters will be updated accordingly.

Bit 3 of a memory BAR controls whether the BAR is prefetchable. If you choose the prefetchable memory option for an individual BAR in the wizard, bit 3 of the corresponding BAR n parameter will be updated.

Table 3–23 shows the format of memory BARs.

Table 3–23. 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 _{<i>n</i>} parameter.
2..1	mem_type	Read	Memory type. The mem_type bits indicate the type of memory that can be implemented in the function's memory address space. Only the following two possible values are valid for the PCI MegaCore functions: locate memory space in the 32-bit address space and locate memory space in the 64-bit address space.
3	pre_fetch	Read	Memory prefetchable. The pre_fetch bit indicates whether the blocks of memory are prefetchable by the host bridge.
31..4	bar	Read/write	Base address registers.

In addition to the type of space reserved by the BAR, the wizard allows you to define the size of address space reserved for each individual BAR and sets the BAR_{*n*} parameter value accordingly. The value for parameter BAR_{*n*} defines the number of read/write bits instantiated in the corresponding BAR (Refer to Section 6.2.5 of the *PCI Local Bus Specification, Revision 3.0*). The number of read/write bits instantiated in a BAR is indicated 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 (e.g., 0xFF000000 is legal, but 0xFF700000 is not). The wizard does not offer options that set the BAR_{*n*} parameters to illegal values.

For high-end systems that require more than 4 GBytes of memory space, the pci_mt64 and pci_t64 functions support 64-bit addressing. These functions offer the option to use either BARs 1 and 0 or BARs 2 and 1 to implement a 64-bit BAR.

When implementing a 64-bit BAR, the least significant BAR contains the lower 32-bit BAR and the most significant BAR contains the upper 32-bit BAR. When implementing a 64-bit BAR, the wizard allows the option of which BARs to use and sets the BAR_{*n*} parameters accordingly. On the least significant BAR, bits [31..4] are read/write registers that are used to indicate the size of the memory, along with the most significant BAR. For the most significant BAR, the wizard allows you to choose the maximum number of read/write registers to implement per the application.

For example, if a 64-bit BAR on BARs 1 and 0 is implemented and the designer indicates 8 as the maximum number of address bits to decode on the upper BAR, the upper 24 bits [31 . . 8] of BAR1 will be read-only bits tied to ground. The eight least significant bits [7 . . 0] of BAR1 are read/write registers, and— along with bits [31 . . 4] of BAR0—they indicate the size of the memory. When a 64-bit memory BAR is implemented, the remaining BARs can still be used for 32-bit memory or I/O base address registers in conjunction with a 64-bit BAR setting. If BARs 2 and 1 are used to implement a 64-bit BAR, BAR0 must be used as a 32-bit memory or I/O base address register.



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

Like a memory BAR, an I/O BAR can be instantiated on any of the six BARs available for the PCI function. The wizard offers the option to implement a 32-bit BAR as memory or I/O and sets the bits [1 . . 0] of the corresponding BAR_n parameter accordingly. The *PCI Local Bus Specification, Revision 3.0* prevents any single I/O BAR from reserving more than 256 bytes of I/O space. Refer to [Table 3–24](#).

Table 3–24. 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 MegaCore functions allow you to set default base addresses that can be used to claim transactions without requiring the configuration of the corresponding BARs. The wizard allows you to implement this feature on an individual BAR_n basis and sets the corresponding parameters accordingly. When using the hardwire BAR feature, the corresponding BAR_n attributes must indicate the appropriate BAR settings, such as size and type of address space.



When implementing a hardwire BAR, the corresponding BARs become read-only. A configuration write to the hardwired BAR will proceed normally. However, a configuration read of hardwired BARs will return the value set in the hardwire BAR_n 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 MegaCore functions do not enable this register. The CIS Pointer register can be enabled and the register's value can be set through the wizard. [Table 3–25](#) shows this register's format. For more information on the CardBus CIS pointer register, refer to the *PC Card Standard Specification, Version 2.10*.

Table 3–25. CIS Pointer Register Format

Data Bit	Mnemonic	Read/Write	Definition
2..0	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. The PCI MegaCore functions do not support the condition where the CIS pointer register points to the configuration space.
27..3	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. Refer to [Table 3–26](#). The default value of the subsystem vendor ID register is 0x0000. However, you can change the value through the wizard.

Table 3–26. Subsystem Vendor ID Register Format

Data Bit	Mnemonic	Read/Write	Definition
15..0	sub_ven_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. Refer to [Table 3–27](#). The default value of the subsystem ID register is 0x0000. However, you can change the value through the wizard.

Table 3–27. Subsystem ID Register Format

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

Expansion ROM Base Address Register

The expansion ROM base address register contains a 32-bit hexadecimal number that defines the base address and size information of the expansion ROM. Instantiate the expansion ROM BAR using the **Parameterize - PCI Compiler** wizard. The expansion ROM BAR functions exactly like a 32-bit BAR, except that the encoding of the bottom bits is different. Bit 0 in the register is a read/write and is used to indicate whether or not the device accepts accesses to its expansion ROM. You can disable the expansion ROM address space by setting bit 0 to 0. You can enable the address decoding of the expansion ROM by setting bit 0 to 1. The upper 21 bits correspond to the upper 21 bits of the expansion ROM base address. The amount of address space a device requests must not be greater than 16 Megabytes (MBytes). The expansion ROM BAR is formatted per the *PCI Local Bus Specification, Revision 3.0*. Refer to [Table 3–28](#).

Table 3–28. Expansion ROM Base Address Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	adr_ena	Read/write	Address decode enable. The <code>adr_ena</code> 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.

The PCI MegaCore functions allow you to set a default expansion ROM base address using the hardwire option in the **Parameterize - PCI Compiler** wizard. Using a hardwire BAR allows the function to claim transactions without requiring the configuration of the expansion ROM BAR. When using the hardwire expansion ROM BAR feature, the expansion ROM BAR attributes must indicate the appropriate BAR settings.



When implementing a hardwire expansion ROM BAR, the corresponding BARs become read only. However, bit 0 is read/write, allowing you to disable the expansion ROM BAR after power-up.

Capabilities Pointer

The capabilities pointer register is an 8-bit read-only register that can be enabled through the wizard. The capabilities pointer value entered through the wizard points to the first item in the list of capabilities. For a list of the capability IDs, refer to Appendix H of the *PCI Local Bus Specification, Revision 3.0*. The address value of this pointer must be 0x40 or greater, and each capability must be within DWORD boundaries. Refer to [Table 3–29](#).

Table 3–29. Capabilities Pointer Format

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

Configuration transactions to addresses greater than or equal to 0x40 are transferred to the local side of the PCI MegaCore functions and operate as 32-bit transactions. The local side must implement the necessary logic for the capabilities registers.

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 0x00. [Table 3–30](#) shows the

format of the Interrupt Line Register.

Table 3–30. Interrupt Line Register Format

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



The interrupt pin can be enabled or disabled in the wizard. The interrupt pin register will be set to 0x00 if the interrupt option is disabled in the **Parameterize - PCI Compiler** wizard.

Interrupt Pin Register

The interrupt pin register is an 8-bit read-only register that defines the PCI function PCI bus interrupt request line to be `intan`. The default value of the interrupt pin register is 0x01. Refer to [Table 3–31](#).

Table 3–31. Interrupt Pin Register Format

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

Minimum Grant Register

The minimum grant register is an 8-bit read-only register that defines the length of time the 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. You can set this register through the wizard. Refer to [Table 3–32](#).

Table 3–32. Minimum Grant Register Format

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

Maximum Latency Register

The maximum latency register is an 8-bit read-only register that defines the frequency in which the function would like to gain access to the PCI bus. Refer to [Table 3–33](#). You can set this register through the wizard.

Table 3–33. 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 MegaCore functions. Although this section includes waveform diagrams showing typical PCI cycles in target mode for the pci_mt64 MegaCore function, these waveforms are also applicable for the pci_mt32, pci_t64, and pci_t32 MegaCore functions. The pci_mt64 and pci_t64 MegaCore functions support both 32-bit and 64-bit transactions. [Table 3–34](#) lists the PCI and local side signals that apply for each PCI function.

Table 3–34. PCI MegaCore Function Signals (Part 1 of 3)

Signal Name	pci_mt64	pci_t64	pci_mt32	pci_t32
PCI Signals				
clk	✓	✓	✓	✓
rstn	✓	✓	✓	✓
gntn	✓		✓	
reqn	✓		✓	
ad[63..0]	✓	✓	ad[31..0]	ad[31..0]
cben[7..0]	✓	✓	cben[3..0]	cben[3..0]
par	✓	✓	✓	✓
par64	✓	✓		
idsel	✓	✓	✓	✓
framen	✓	✓	✓	✓
req64n	✓	✓		
irdyn	✓	✓	✓	✓
devseln	✓	✓	✓	✓
ack64n	✓	✓		

Table 3–34. PCI MegaCore Function Signals (Part 2 of 3)

Signal Name	pci_mt64	pci_t64	pci_mt32	pci_t32
trdyn	✓	✓	✓	✓
stopn	✓	✓	✓	✓
perrn	✓	✓	✓	✓
serrn	✓	✓	✓	✓
intan	✓	✓	✓	✓
Local-Side Datapath Signals				
l_adi[63..0]	✓	✓	l_adi[31..0]	l_adi[31..0]
l_cbeni[7..0]	✓		l_cbeni[3..0]	
l_adro[63..0]	✓	✓	l_adro[31..0]	l_adro[31..0]
l_dato[63..0]	✓	✓	l_dato[31..0]	l_dato[31..0]
l_beno[7..0]	✓	✓	l_beno[3..0]	l_beno[3..0]
l_cmddo[3..0]	✓	✓	✓	✓
l_ldat_ackn	✓	✓		
l_hdat_ackn	✓	✓		
Target Local-Side Control Signals				
lt_abortn	✓	✓	✓	✓
lt_discn	✓	✓	✓	✓
lt_rdyn	✓	✓	✓	✓
lt_framen	✓	✓	✓	✓
lt_ackn	✓	✓	✓	✓
lt_dxfrn	✓	✓	✓	✓
lt_tsr[11..0]	✓	✓	✓	✓
lirqn	✓	✓	✓	✓
cache[7..0]	✓		✓	
cmd_reg[6..0]	✓	✓	✓	✓
stat_reg[6..0]	✓	✓	✓	✓
Master Local-Side Control Signals				
lm_req32n	✓		✓	
lm_req64n	✓			
lm_lastn	✓		✓	

Table 3–34. PCI MegaCore Function Signals (Part 3 of 3)

Signal Name	pci_mt64	pci_t64	pci_mt32	pci_t32
lm_rdyn	✓		✓	
lm_adr_ackn	✓		✓	
lm_ackn	✓		✓	
lm_dxfrn	✓		✓	
lm_tsr[9..0]	✓		✓	

The `pci_mt64` and `pci_t64` MegaCore functions support the following 64-bit target memory transactions:

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

Each PCI function supports the following 32-bit transactions:

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



The `pci_mt64` and `pci_t64` functions assume that the local side is 64 bits during memory transactions and 32 bits during I/O transactions. For memory read transactions, these functions automatically read 64-bit data on the local side and transfer the data to the PCI master, one DWORD at a time, if the PCI bus is 32 bits wide.

A read or write transaction begins after a master device acquires mastership of the PCI bus and asserts `framen` to indicate the beginning of a bus transaction. If the transaction is a 64-bit transaction, the master device asserts the `req64n` signal at the same time as it asserts the `framen` signal. The clock cycle where the `framen` signal is asserted is called the address phase. During the address phase, the master device drives the transaction address and command on `ad[31..0]` and `cben[3..0]`, respectively. When `framen` is asserted, the PCI MegaCore function latches the address and command signals on the first clock edge and starts the address decode phase. If the transaction address matches the target, the target asserts the `devseln` signal to claim the transaction. In the case of 64-bit transactions, the `pci_mt64` and `pci_t64` assert the `ack64n` signal at the same time as the `devseln` signal indicating that the `pci_mt64` and `pci_t64` accepts the 64-bit transaction. All PCI MegaCore functions implement slow decode (i.e., the `devseln` and

ack64n signals in the pci_mt64 and pci_t64 functions are asserted three clock cycles after a valid address is presented on the PCI bus). In all operations except configuration read/write, one of the lt_tsr[5..0] signals is driven high, indicating the BAR range address of the current transaction.

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

Memory transactions can be single-cycle or burst. In target mode, the PCI MegaCore function supports an unlimited length of zero-wait state memory burst read or write transactions. 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 3-77 describes how to initiate the different types of termination.



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

I/O transactions are always single-cycle 32-bit transactions. Therefore, the PCI MegaCore function handles them like single-cycle memory commands. Any of the six BARs in the PCI MegaCore functions can be configured to reserve I/O space. Refer to “[Base Address Registers](#)” on page 3-37 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 PCI 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

This section describes the behavior of the PCI MegaCore functions in the following types of target read transactions:

- Memory read
- I/O read
- Configuration read

Memory Read Transactions

The PCI MegaCore functions support the following types of matched bus width and mismatched bus width memory read transactions in target mode:

- Single-cycle memory read
- Burst memory read
- Mismatched bus width memory read



Mismatched bus-width transactions are 32-bit PCI transactions performed by the `pci_mt64` and `pci_t64` MegaCore functions.

For all memory read transactions, the following sequence of events is the same:

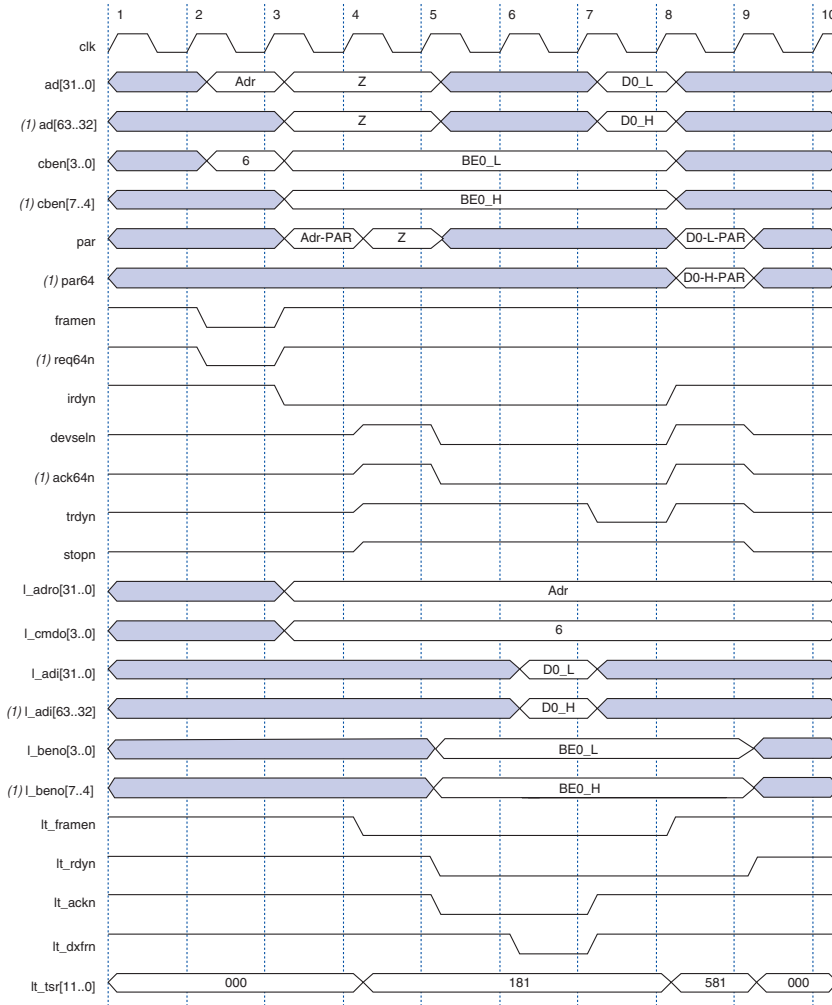
1. The address phase occurs when the PCI master asserts `framen` (and `req64n` in the case of a 64-bit transaction) and drives the address on `ad[31..0]` and the command on `cben[3..0]`. Asserting `req64n` indicates to the target device that the master device is requesting a 64-bit data transaction.
2. Turn-around cycles on the `ad` bus occur during the clock cycle immediately following the address phase. During turn-around cycles the PCI side drives correct byte enables on the `cben` bus for the first data phase but tri-states the `ad` bus. This process is necessary because the PCI agent driving the `ad` bus changes during read cycles.
3. If the address of the transaction matches the memory range specified in a base address register, the PCI MegaCore function turns on the drivers for the `ad` bus, `devseln`, `trdyn`, `stopn`, and `par` (as well as `par64` and `ack64n` for 64-bit transactions) in the following clock cycle.

4. The PCI MegaCore function drives and asserts `devseln` (and `ack64n` for 64-bit transactions) to indicate to the master device that it is accepting the transaction.
5. One or more data phases follow, depending on the type of read transaction.

Single-cycle Memory Read Target Transactions

Figure 3–7 shows the waveform for a 64-bit single-cycle memory read target transaction. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` MegaCore functions.

Figure 3–7. Single-Cycle Memory Read Target Transaction



Note Figure 3–7:

(1) This signal is not applicable to the `pci_mt32` or `pci_t32` MegaCore functions.

Table 3–35 shows the sequence of events for a 64-bit single-cycle memory read target transaction. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` MegaCore functions.

Table 3–35. Single-Cycle Memory Read Target Transaction (Part 1 of 2)

Clock Cycle	Event
1	The PCI bus is idle.
2	The address phase occurs.
3	The PCI 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 cycle 3, the master deasserts the <code>framen</code> and <code>req64n</code> signals and asserts <code>irdyn</code> to indicate that only one data phase remains in the transaction. For a single-cycle memory read, this phase is the only data phase in the transaction. The PCI MegaCore function begins to decode the address during clock cycle 3, and if the address falls in the range of one of its BARs, the transaction is claimed. The PCI master tri-states the <code>ad</code> bus for the turn-around cycle.
4	If the PCI MegaCore function detects an address hit in clock cycle 3, several events occur during clock cycle 4: • The PCI MegaCore function informs the local-side device that it is claiming the read transaction by asserting <code>lt_framen</code> and the bit on <code>lt_tsr[5..0]</code> that corresponds to the BAR range hit. In Figure 3–7, <code>lt_tsr[0]</code> is asserted indicating that a base address register zero hit. • The MegaCore function drives the transaction command on <code>l_cmdo[3..0]</code> and address on <code>l_adro[31..0]</code>. • The PCI MegaCore function turns on the drivers of <code>devseln</code>, <code>ack64n</code>, <code>trdyn</code>, and <code>stopn</code>, getting ready to assert <code>devseln</code> and <code>ack64n</code> in clock cycle 5. • <code>lt_tsr[7]</code> is asserted to indicate that the pending transaction is 64-bits. • <code>lt_tsr[8]</code> is asserted to indicate that the PCI side of the PCI MegaCore function is busy.
5	The PCI MegaCore function asserts <code>devseln</code> and <code>ack64n</code> to claim the transaction. The function also drives <code>lt_ackn</code> to the local-side device to indicate that it is ready to accept data on the <code>l_adi</code> bus. The PCI MegaCore function also enables the output drivers of the <code>ad</code> bus to ensure that it is not tri-stated for a long time while waiting for valid data. Although the local side asserts <code>lt_rdyn</code> during clock cycle 5, the data transfer does not occur until clock cycle 6.
6	<code>lt_rdyn</code> is asserted in clock cycle 5, indicating that valid data is available on the <code>l_adi</code> bus in clock cycle 6. The PCI MegaCore function registers the data into its internal pipeline on the rising edge of clock cycle 7. The local side transfer is indicated by the <code>lt_dxfrn</code> signal. The <code>lt_dxfrn</code> signal is low during the clock cycle where a data transfer on the local side occurs. The local side data transfer occurs if <code>lt_ackn</code> is asserted on the current clock edge while <code>lt_rdyn</code> is asserted on the previous clock edge. The <code>lt_dxfrn</code> signal is asserted to indicate a successful data transfer.
7	The rising edge of clock cycle 7 registers the valid data from the <code>l_adi</code> bus and drives the data on the <code>ad</code> bus. At the same time, the PCI MegaCore function asserts the <code>trdyn</code> signal to indicate that there is valid data on the <code>ad</code> bus.

Table 3–35. Single-Cycle Memory Read Target Transaction (Part 2 of 2)

Clock Cycle	Event
8	The PCI MegaCore function deasserts <code>trdyn</code> , <code>devseln</code> , and <code>ack64n</code> to end the transaction. To satisfy the requirements for sustained tri-state buffers, the PCI MegaCore function drives <code>devseln</code> , <code>ack64n</code> , <code>trdyn</code> , and <code>stopn</code> high during this clock cycle. Additionally, the PCI MegaCore function tri-states the <code>ad</code> bus because the cycle is complete. The rising edge of clock cycle 8 signals the end of the last data phase because <code>framen</code> is deasserted and <code>irdyn</code> and <code>trdyn</code> are asserted. In clock cycle 8, the PCI MegaCore function also informs the local side that no more data is required by deasserting <code>lt_framen</code> , and <code>lt_tsr[10]</code> is asserted to indicate a successful data transfer on the PCI side during the previous clock cycle.
9	The PCI MegaCore function informs the local-side device that the transaction is complete by deasserting the <code>lt_tsr[11..0]</code> signals. Additionally, the PCI MegaCore function tri-states <code>devseln</code> , <code>ack64n</code> , <code>trdyn</code> , and <code>stopn</code> to begin the turn-around cycle on the PCI bus.



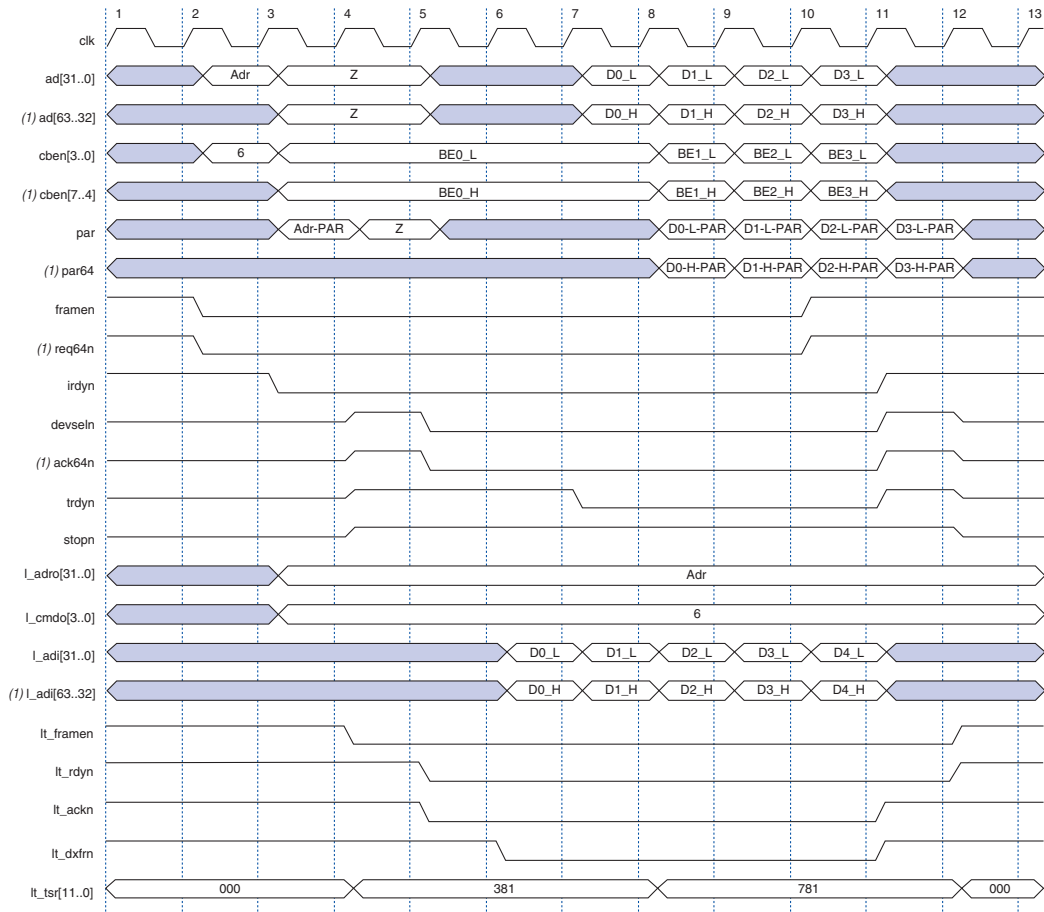
The local-side design must ensure that PCI latency rules are not violated while the PCI MegaCore function waits for data. If the local-side design is unable to meet the latency requirements, it must assert `lt_discn` to request that the PCI MegaCore function terminate the transaction. The PCI target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

Burst Memory Read Target Transactions

The sequence of events for a burst memory read target transaction is the same as that of a single-cycle memory read target transaction. However, during a burst read transaction, more data is transferred and both the local-side design and the PCI master can insert wait states at any point during the transaction. Figure 3–8 illustrates a burst memory read target transaction. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` MegaCore functions.

The transaction shown in Figure 3–8 is a 64-bit zero-wait state burst transaction with four data phases. The local side transfers five quad words (QWORDS) in clock cycles 6 through 10. The PCI MegaCore function transfers data to the PCI side in clock cycles 7 through 10. Because of the PCI MegaCore function's zero-wait state requirement, the PCI side reads ahead from the local side. Also, because the `l_ben0` bus is not available until after a local data phase has completed, the delay between data transfers on the local side and PCI side requires the local target device to supply valid data on all bytes. If the local side is not prefetchable (i.e., reading ahead will result in lost or corrupt data), it must not accept burst read transactions, and it should disconnect after the first QWORD transfer on the local side. Additionally, Figure 3–8 shows the `lt_tsr[9]` signal asserted in clock cycle 4 because the master device has `framen` and `irdyn` signals asserted, thus indicating a burst transaction.

A burst transaction is indicated by the PCI MegaCore function if it detects both `irdyn` and `framen` are asserted on the PCI side after the address phase. The PCI MegaCore function asserts `lt_tsr[9]` to indicate a burst transaction. The function asserts `lt_tsr[9]` if both `irdyn` and `framen` are asserted during a valid target transaction. If `lt_tsr[9]` is not asserted during a transaction, it indicates that `irdyn` and `framen` have not been detected or asserted during the transaction. Typically this situation indicates that the current transaction is single-cycle. However, this situation is not guaranteed because it is possible for the master to delay the assertion of `irdyn` in the first data phase by up to 8 clock cycles. In other words, if `lt_tsr[9]` is asserted during a valid target transaction, it indicates that the impending transaction is a burst, but if `lt_tsr[9]` is not asserted it may or may not indicate that the transaction is single-cycle.

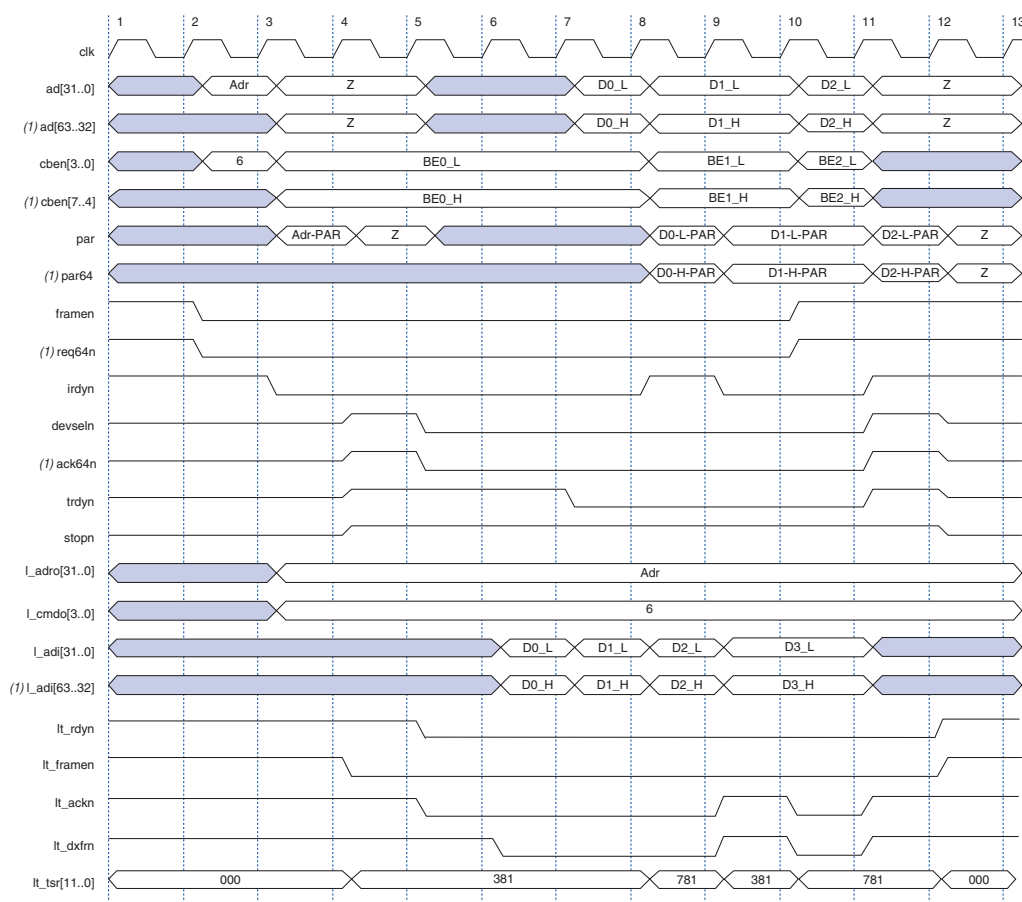
Figure 3–8. Zero-Wait State Burst Memory Read Target Transaction**Note to Figure 3–8:**

(1) This signal is not applicable to the `pci_mt32` or `pci_t32` MegaCore functions.

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

The 64-bit extension signals shown in Figure 3–9 are not applicable to the `pci_mt32` and `pci_t32` MegaCore functions.

Figure 3–9. Burst Memory Read Target Transaction with PCI Master Wait State

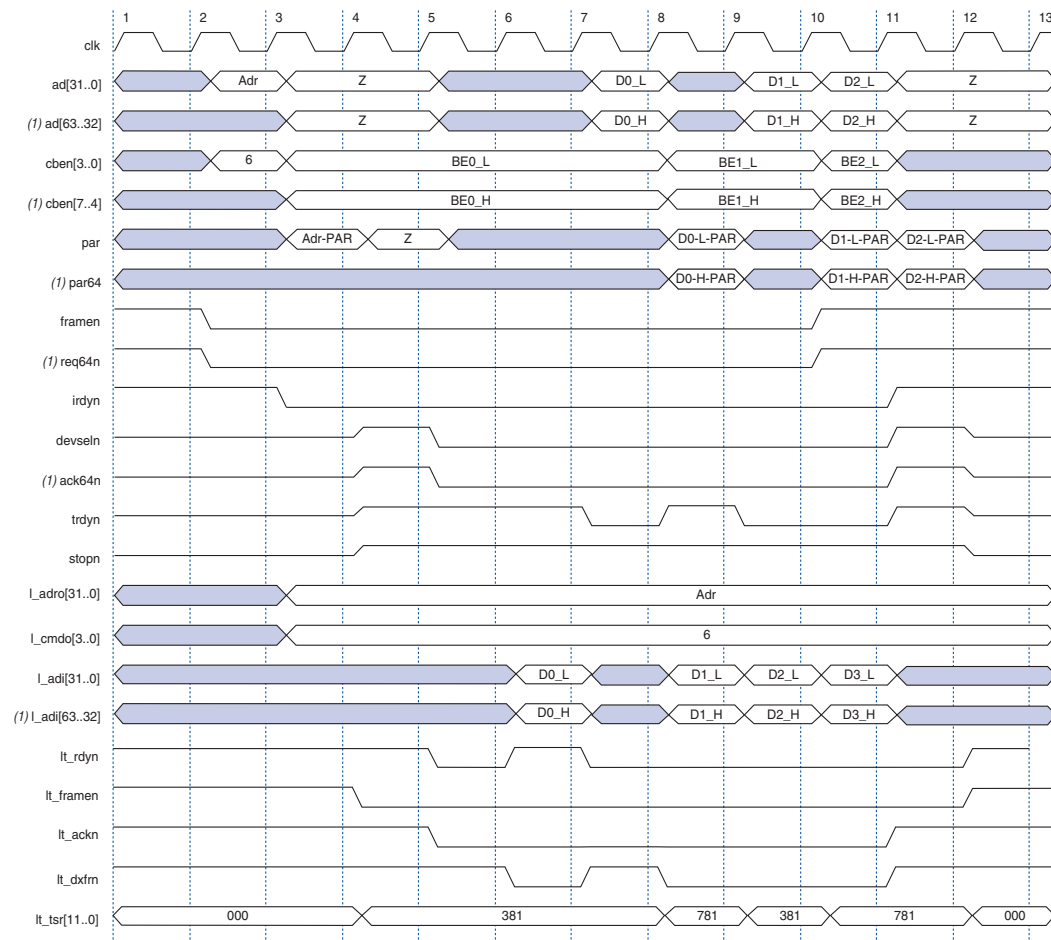


Note to Figure 3–9:

- (1) This signal is not applicable to the `pci_mt32` or `pci_t32` MegaCore functions.

Figure 3–10 shows the same transaction as shown in Figure 3–8 with the local side inserting a wait state. The local side deasserts `lt_rdyn` in clock cycle 6. Deasserting `lt_rdyn` in clock cycle 6 suspends the local side data transfer in clock cycle 7 by deasserting the `lt_dxfrn` signal. Because no data is transferred in clock cycle 7 from the local side, the PCI MegaCore function deasserts `trdyn` in clock cycle 8 thus inserting a PCI wait state.

Figure 3–10. Burst Memory Read Target Transaction with Local-Side Wait State



Note to Figure 3–10:

(1) This signal is not applicable to the `pci_mt32` or `pci_t32` MegaCore functions.

Mismatched Bus Width Memory Read Target Transactions

The following description applies only to the `pci_mt64` and `pci_t64` MegaCore functions handling mismatched bus width memory read target transactions.

When using the `pci_mt64` or `pci_t64` MegaCore functions to accept 32-bit memory read transactions, the local side data bus width is 64 bits while the PCI data bus width is 32 bits. The `pci_mt64` and `pci_t64` functions handle this bus width mismatch and automatically perform DWORD alignment.

The `pci_mt64` and `pci_t64` functions always assume a 64-bit local side data bus width during memory read transactions. The functions read 64-bit data (QWORD, or two DWORDs) and automatically transfer one DWORD at a time to the PCI side. For the first PCI data phase, the `pci_mt64` and `pci_t64` also perform automatic DWORD alignment, depending on the PCI starting address of the transaction.

If the address of the transaction is a QWORD boundary (`ad[2..0] == B"000"`), the first DWORD transferred to the PCI side is the low DWORD, and `pci_mt64` or `pci_t64` assert both `l_ldat_ackn` and `l_hdat_ackn` to indicate that the PCI MegaCore function will transfer both DWORDs that were transferred on the local side. However, if the address of the transaction is not at a QWORD boundary (`ad[2..0] == B"100"`), the first DWORD transferred to the PCI side is the high DWORD. The low DWORD is not transferred to the PCI side. The `pci_mt64` and `pci_t64` functions deassert `l_ldat_ackn` and assert `l_hdat_ackn` during the first data transfer on the local side to indicate that only the high DWORD is transferred to the PCI side.

Figure 3-11 shows a 32-bit single-cycle mismatched bus width memory read target transaction, which applies to the `pci_mt64` and `pci_t64` functions. Refer to Figure 3-7 for the description of a 32-bit single-cycle memory read transaction using the `pci_mt32` and `pci_t32` functions.

The sequence of events in Figure 3-11 is exactly the same as in Figure 3-7, except for the following cases:

- During the address phase (clock cycle 3), the master does not assert `req64n` because the transaction is 32 bits
- The `pci_mt64` or `pci_t64` function does not assert `ack64n` when it asserts `devseln`
- The local side is informed that the pending transaction is 32 bits because `lt_tsr[7]` is not asserted while `lt_framen` is asserted

Figure 3–11 shows that the local side transfers a full QWORD in clock cycle 6. In clock cycle 7, the `pci_mt64` and `pci_t64` functions drive the least significant DWORD on `ad[31..0]`. The `pci_mt64` and `pci_t64` functions drive the correct parity value on the `par` signal in clock cycle 8.



The `pci_mt64` and the `pci_t64` functions always transfer 64-bit data on the local side. In a 32-bit single-cycle memory read transaction, only one DWORD is transferred to the PCI master.

Figure 3–11. 32-Bit PCI and 64-Bit Local-Side Single-Cycle Memory Read Target Transaction

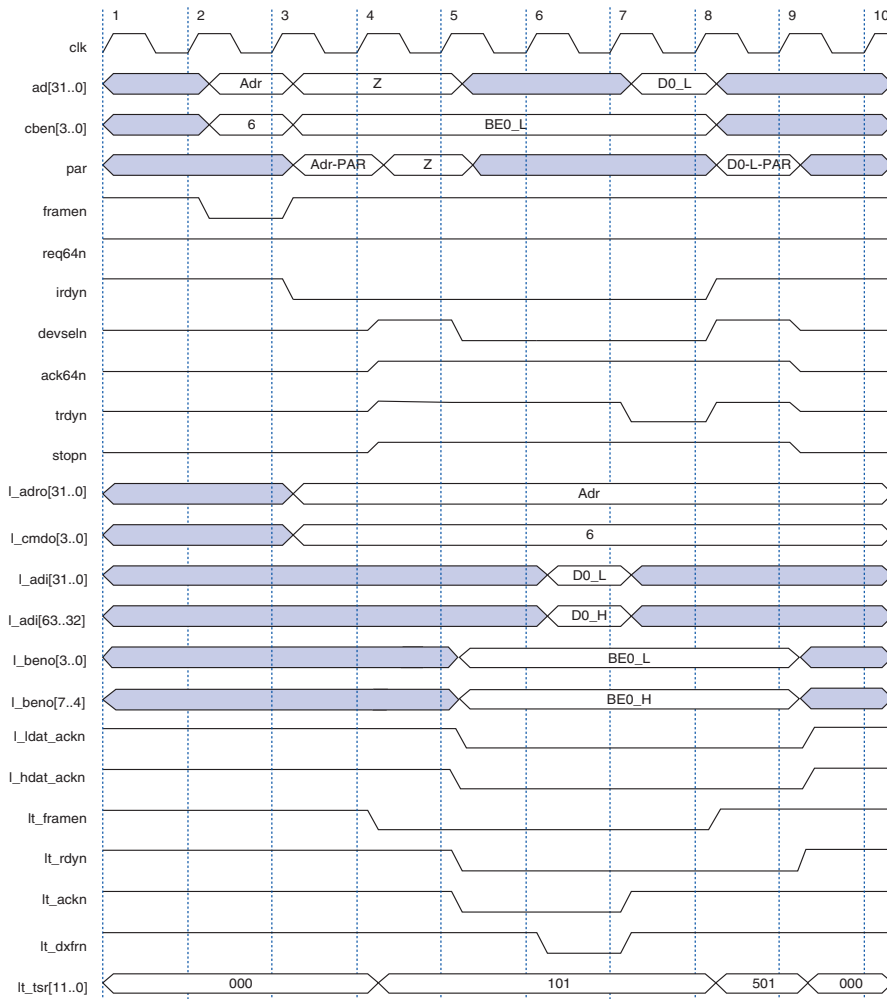
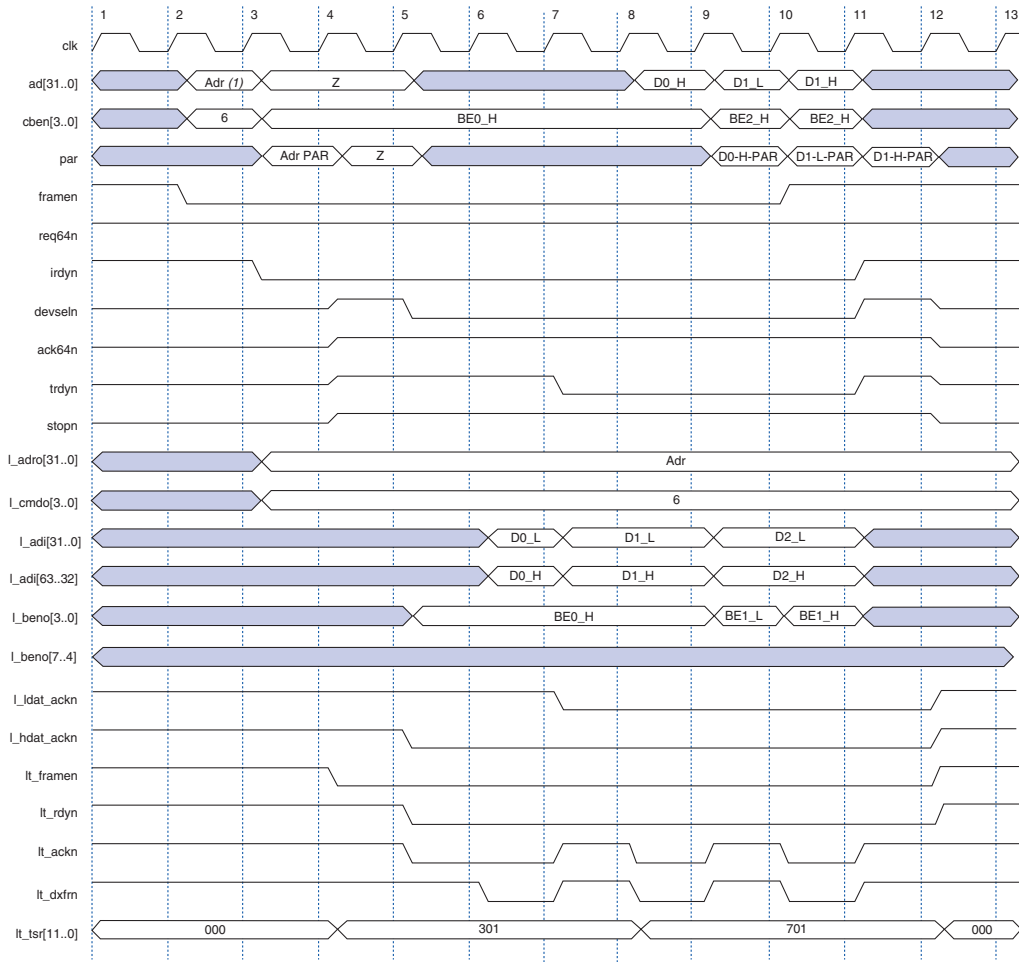


Figure 3–12 shows a 32-bit PCI side and 64-bit local side burst memory read target transaction. (This figure only applies to the `pci_mt64` and `pci_t64` functions.)

The events in Figure 3–12 are the same as those shown in Figure 3–8, except that a 64-bit transfer shown in Figure 3–12 takes one clock cycle on the local side but requires two clock cycles on the PCI side. The function automatically inserts local side wait states in clock cycles 7 and 9 to temporarily suspend the local transfer allowing sufficient time for the data to be transferred on the PCI side. In Figure 3–12, `lt_tsr[7]` is not asserted and `lt_tsr[9]` is asserted indicating that the transaction is a 32-bit burst. If the local side cannot handle 32-bit burst transactions, it must disconnect after the first local transfer.

Also, because the address of the transaction is not at a QWORD boundary (`ad[2..0] == B"100"`), the first DWORD transferred to the PCI side is the high DWORD. The first low DWORD is not transferred to the PCI side. The `pci_mt64` and `pci_t64` functions deassert `l_ldat_ackn` and assert `l_hdat_ackn` during the first data transfer on the local side to indicate that only the high DWORD is transferred to the PCI side, as shown in Figure 3–12 at clock cycle 7.

Figure 3–12. 32-Bit PCI and 64-Bit Local-Side Burst Memory Read Target Transaction**Note to Figure 3–12:**

(1) The value on *ad*[31..0] is not a QWORD address boundary (*ad*[2..0] == B"100").

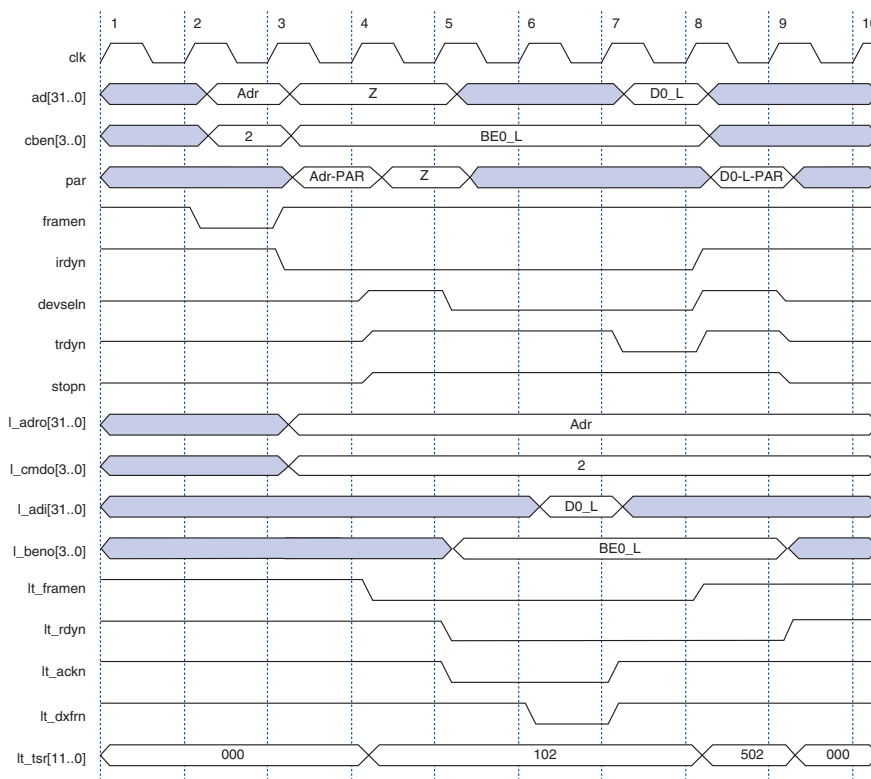
I/O Read Transactions

I/O read transactions by definition are 32 bits wide. Figure 3–13 shows a sample I/O read transaction. The sequence of events is the same as 32-bit single-cycle memory read transactions. The main distinction between the two transactions is the command on the `lt_cmdbo[3..0]` bus. In Figure 3–13, `lt_tsr[11..0]` indicates that the base address register that detected the address hit is BAR1. Additionally, during an I/O transaction `l_1dat_ackn` and `l_hdat_ackn` are not relevant.



The PCI MegaCore functions do not ensure that the combination of the `ad[1..0]` and `cbe[3..0]` signals is valid during the address phase of an I/O transaction. Local side logic should implement this functionality if performing I/O transactions. Refer to the *PCI Local Bus Specification, Revision 3.0* for more information on handling invalid combinations of these signals.

Figure 3–13. I/O Read Transaction



Configuration Read Transactions

Configuration read transactions are 32 bits. Configuration cycles are automatically handled by the PCI MegaCore functions and do not require local side actions. [Figure 3–14](#) shows a typical configuration read transaction. This figure applies to all PCI MegaCore functions. The configuration read transaction is similar to 32-bit single-cycle transactions, except for the following terms:

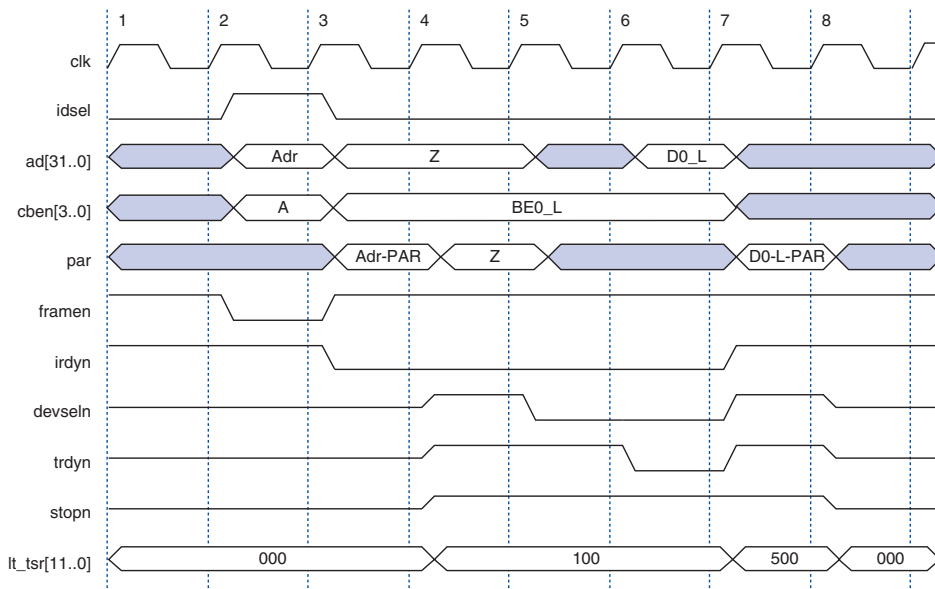
- During the address phase, `idsel` must be asserted
- Because the configuration read does not require data from the local side, the PCI MegaCore functions assert `trdyn` independent from the `lt_rdyn` signal

The second case above results in `trdyn` being asserted in clock cycle 6 instead of clock cycle 7 as shown in [Figure 3–14](#). The configuration read cycle ends in clock cycle 8.



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

Figure 3–14. Configuration Read Transaction



Target Write Transactions

This section describes the behavior of the PCI MegaCore functions in the following types of target write transactions:

- Memory write
- I/O write
- Configuration write

Memory Write Transactions

The PCI MegaCore functions support the following types of matched bus width and mismatched bus width memory write transactions in target mode:

- Single-cycle memory write
- Burst memory write
- Mismatched bus width memory write



Mismatched bus-width transactions are 32-bit PCI transactions performed by the `pci_mt64` and `pci_t64` MegaCore functions.

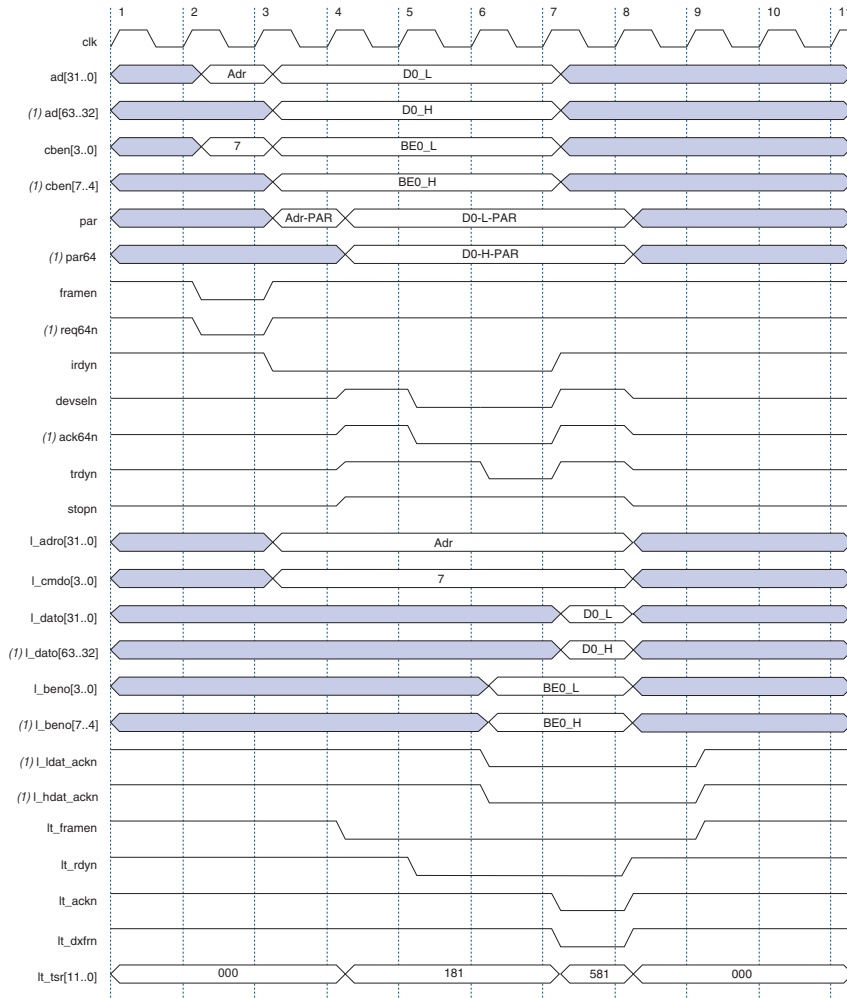
For all memory write transactions, the following sequence of events is the same:

1. The address phase occurs when the PCI master asserts `framem` (and `req64n` in the case of a 64-bit transaction) and drives the address on `ad[31..0]` and the command on `cben[3..0]`. Asserting `req64n` indicates to the target device that the master device is requesting a 64-bit data transaction.
2. If the address of the transaction matches the memory range specified in a base address register, the PCI MegaCore function turns on the drivers for the `ad bus`, `devseln`, `trdyn`, `stopn`, and `par` (as well as `par64` and `ack64n` for 64-bit transactions) in the following clock cycle.
3. The PCI MegaCore function drives and asserts `devseln` (and `ack64n` for 64-bit transactions) to indicate to the master device that it is accepting the transaction.
4. One or more data phases follow, depending on the type of write transaction.

Single-cycle Memory Write Target Transactions

Figure 3–15 shows the waveform for a 64-bit single-cycle memory write target transaction. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` MegaCore functions.

Figure 3–15. Single-Cycle Memory Write Target Transaction



Note to Figure 3–15:

(1) This signal is not applicable to the `pci_mt32` or `pci_t32` MegaCore functions.

Table 3–36 shows the sequence of events for a 64-bit single-cycle memory write target transaction. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` MegaCore functions.

Table 3–36. Single-Cycle Memory Write Target Transactions (Part 1 of 2)

Clock Cycle	Event
1	The PCI bus is idle.
2	The address phase occurs.
3	The PCI 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 cycle 3, the master deasserts the <code>framen</code> and <code>req64n</code> signals and asserts <code>irdyn</code> to indicate that only one data phase remains in the transaction. For a single-cycle memory write target transaction, this phase is the only data phase in the transaction. The MegaCore function uses clock cycle 3 to decode the address, and if the address falls in the range of one of its BARs, the transaction is claimed.
4	If the PCI MegaCore function detects an address hit in clock cycle 3, several events occur during clock cycle 4: • The PCI MegaCore function informs the local-side device that it is going to claim the write transaction by asserting <code>lt_framen</code> and the bit on <code>lt_tsr[5..0]</code> that corresponds to the BAR range hit. In Figure 3–15, <code>lt_tsr[0]</code> is asserted indicating that a base address register zero hit. • The PCI MegaCore function drives the transaction command on <code>l_cmdbo[3..0]</code> and address on <code>l_adro[31..0]</code>. • The PCI MegaCore function turns on the drivers of <code>devseln</code>, <code>ack64n</code>, <code>trdyn</code>, and <code>stopn</code> getting ready to assert <code>devseln</code> and <code>ack64n</code> in clock cycle 5. • <code>lt_tsr[7]</code> is asserted to indicate that the pending transaction is 64 bits. • <code>lt_tsr[8]</code> is asserted to indicate that the PCI side of the PCI MegaCore function is busy.
5	The PCI MegaCore function asserts <code>devseln</code> to claim the transaction. Figure 3–15 also shows the local side asserting <code>lt_rdyn</code>, indicating that it is ready to receive data from the PCI MegaCore function in clock cycle 6. To allow the local side ample time to issue a retry for the write cycle, the PCI MegaCore function does not assert <code>trdyn</code> in the first data phase unless the local side asserts <code>lt_rdyn</code>. If <code>lt_rdyn</code> is not asserted in clock cycle 5 (Figure 3–15), the PCI MegaCore function delays the assertion of <code>trdyn</code>.
6	The PCI MegaCore function asserts <code>trdyn</code> to inform the PCI master that it is ready to accept data. Because <code>irdyn</code> is already asserted, this clock cycle is the first and last data phase in this cycle.

Table 3–36. Single-Cycle Memory Write Target Transactions (Part 2 of 2)

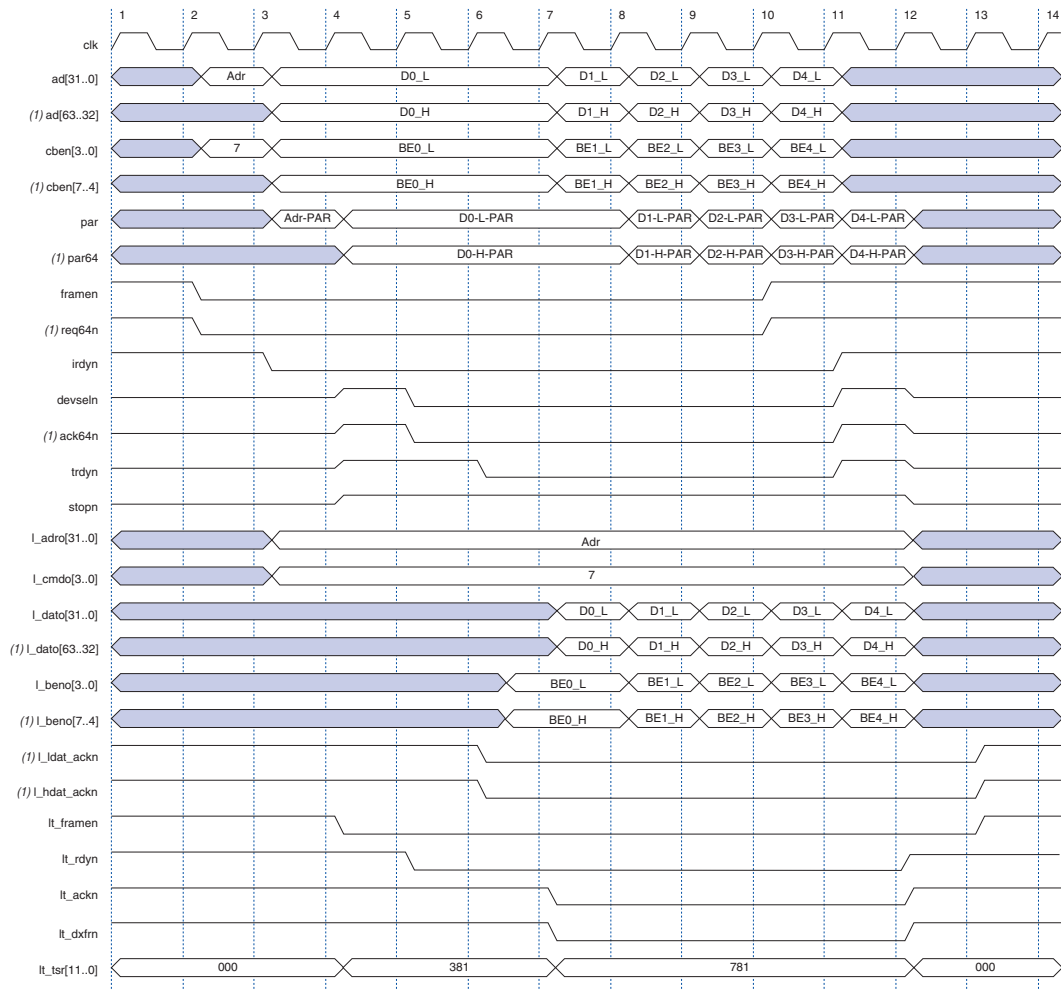
Clock Cycle	Event
7	The rising edge of clock cycle 7 registers the valid data from the <code>ad</code> bus and drives the data on the <code>l_dato</code> bus, registers valid byte enables from the <code>cben</code> bus, and drives the byte enables on the <code>l_beno</code> bus. At the same time, the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

Burst Memory Write Target Transactions

The sequence of events in a burst write transaction is the same as for a single-cycle memory write target 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 3–16 shows a 64-bit zero-wait state burst memory write target transaction with five data phases. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` functions. The PCI master writes five QWORDS to the PCI MegaCore function during clock cycles 6 through 10. The PCI MegaCore function transfers the same data to the local side during clock cycles 7 through 11. Additionally, Figure 3–16 shows the `lt_tsr[9]` signal asserted in clock cycle 4 because the master device has the `framen` and `irdyn` signals asserted, thus indicating a burst transaction.

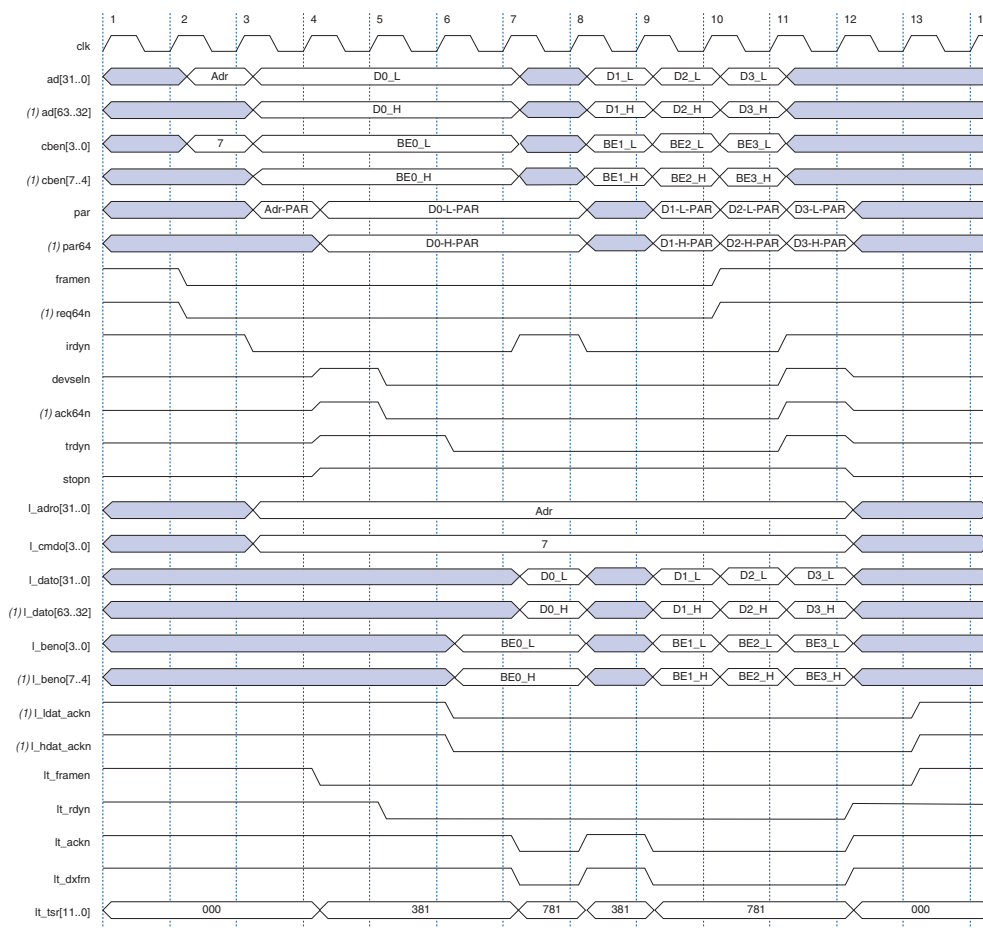
A burst transaction is indicated by the PCI MegaCore function if it detects both `irdyn` and `framen` are asserted on the PCI side after the address phase. The PCI MegaCore function asserts `lt_tsr[9]` to indicate a burst transaction. If `lt_tsr[9]` is not asserted during a transaction, it indicates that `irdyn` and `framen` have not been detected or asserted during the transaction. Typically this event indicates that the current transaction is single-cycle. However, this situation is not guaranteed because it is possible for the master to delay the assertion of `irdyn` in the first data phase by up to 8 clock cycles. In other words, if `lt_tsr[9]` is asserted during a valid target transaction, it indicates that the impending transaction is a burst, but if `lt_tsr[9]` is not asserted it may or may not indicate that the transaction is single-cycle.

Figure 3–16. Zero-Wait State Burst Memory Write Target Transaction**Note to Figure 3–16:**

(1) This signal is not applicable to the pci_mt32 or pci_t32 MegaCore functions.

Figure 3–17 shows the same transaction as in Figure 3–16 with the PCI bus master inserting a wait state. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` functions. The PCI bus master inserts a wait state by deasserting the `irdyn` signal in clock cycle 7. The effect of this wait state on the local side is shown in clock cycle 8 as the MegaCore function deasserts `lt_ackn` and as a result `lt_dxfrn` is also deasserted. This prevents data from being transferred to the local side in clock cycle 8 because the internal pipeline of the function does not have valid data.

Figure 3–17. Burst Memory Write Target Transaction with PCI Master Wait State

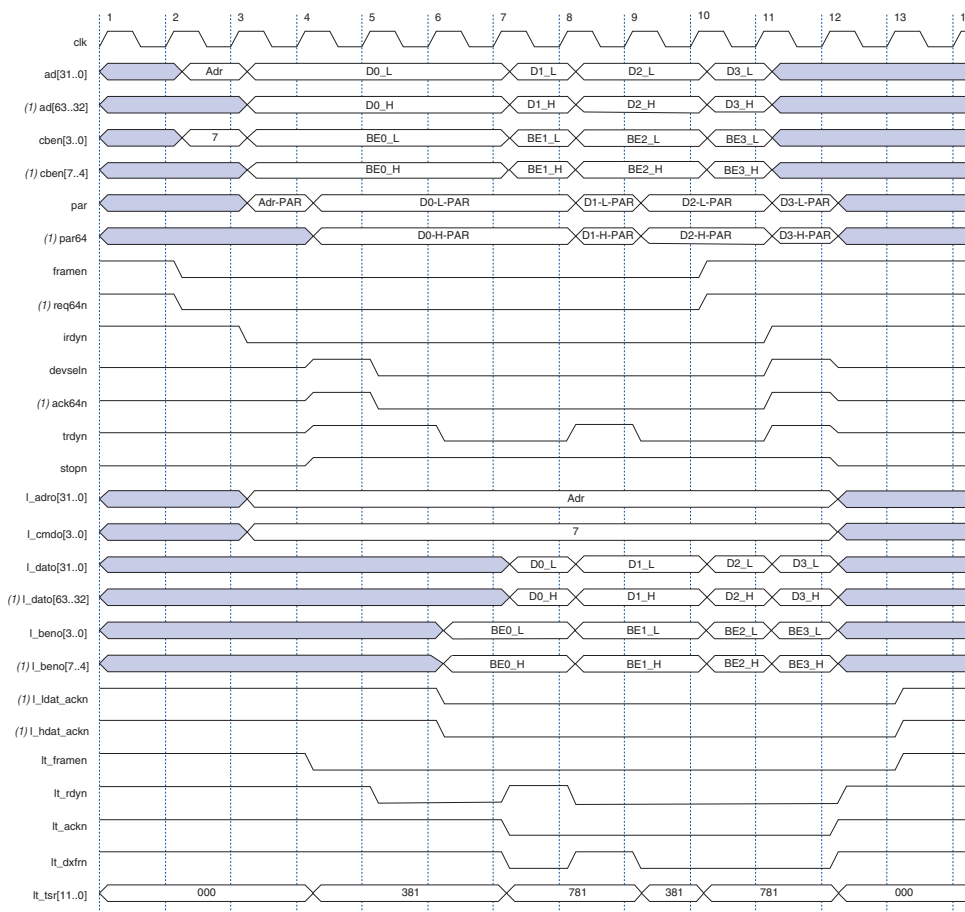


Note to Figure 3–17:

(1) This signal is not applicable to the `pci_mt32` or `pci_t32` MegaCore functions.

Figure 3–18 shows the same transaction as in Figure 3–16 with the local side inserting a wait state. The 64-bit extension signals are not applicable to the `pci_mt32` and `pci_t32` functions. The local side deasserts `lt_rdyn` in clock cycle 7. The function shows that deasserting `lt_rdyn` in clock cycle 7 suspends the local side data transfer in clock cycle 8 by deasserting `lt_dxfrn`. Because the local side is unable to accept additional data in clock cycle 8, the function deasserts `trdyn` in clock cycle 8 as well, preventing PCI data from being transferred from the master device.

Figure 3–18. Burst Memory Write Target Transaction with Local-Side Wait State



Note to Figure 3–18:

(1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

Mismatched Bus-Width Memory Write Target Transactions

The following description applies only to the `pci_mt64` and `pci_t64` functions handling mismatched bus width memory write target transactions.

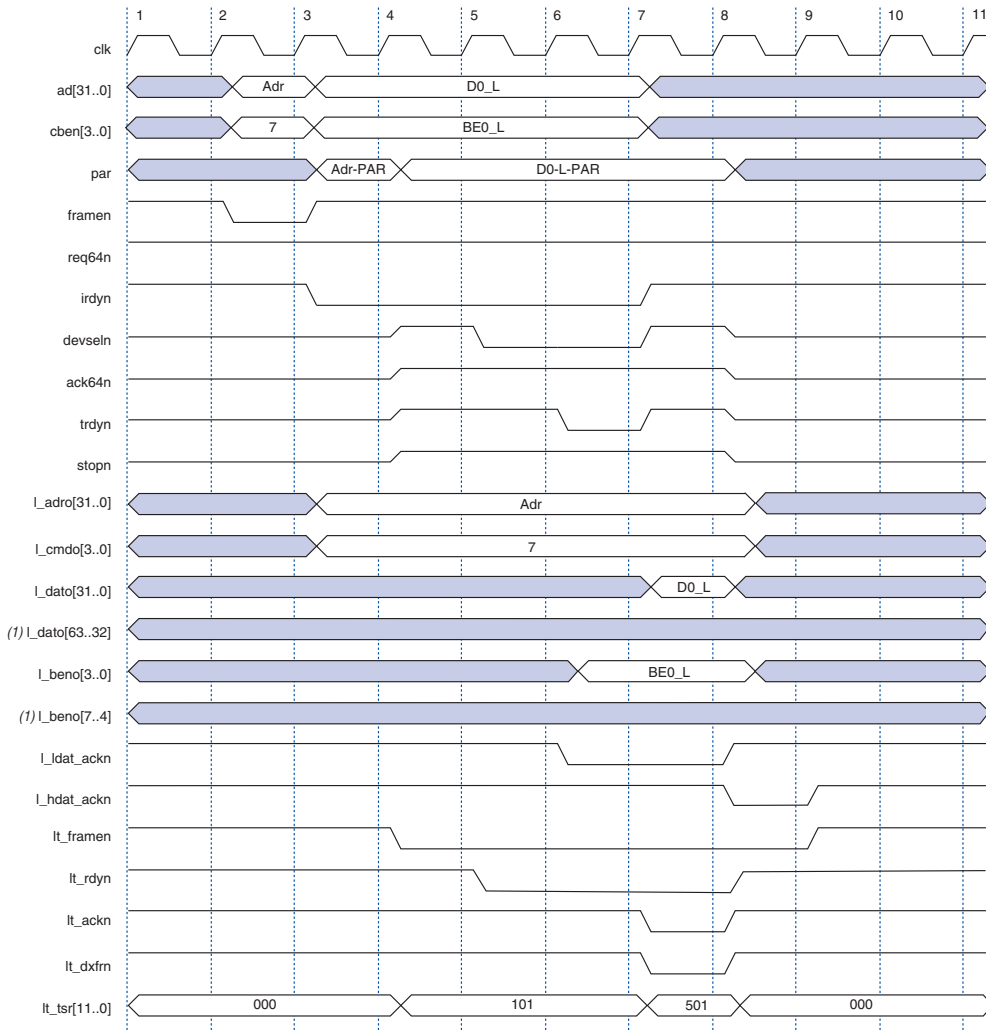
When using the `pci_mt64` or `pci_t64` MegaCore functions to accept 32-bit memory write transactions, the local side data bus width is 64 bits while the PCI data bus width is 32 bits. The `pci_mt64` and `pci_t64` functions transfer 32-bit data from the PCI side and drive that data to the `l_dat0[31..0]` bus. The `pci_mt64` and `pci_t64` functions decode whether the low or high DWORD is addressed by the master device, based on the starting address of the transaction. If the address of the transaction is a QWORD boundary (`ad[2..0] == B"000"`), the first DWORD transferred is considered the low DWORD and `pci_mt64` or `pci_t64` asserts `l_ldat_ackn` accordingly; if the address of the transaction is not at a QWORD boundary (`ad[2..0] == B"100"`), the first DWORD transferred is considered to be the high DWORD and the `pci_mt64` or `pci_t64` function asserts `l_hdat_ackn` accordingly.

Figure 3–19 shows a 32-bit single-cycle mismatched bus width memory write transaction applying to the `pci_mt64` and `pci_t64` functions. Refer to Figure 3–15 for a description of a 32-bit single cycle memory write transaction using the `pci_mt32` or `pci_t32` function. The sequence of events in Figure 3–19 is exactly the same as in Figure 3–15, except for the following:

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

In Figure 3–19, the local-side transfer occurs in clock cycle 7 because `lt_dxfrn` is asserted during that clock cycle. At the same time, `l_ldat_ackn` is asserted to indicate that the low DWORD is valid. This event occurs because the address used in the example is at QWORD boundary.

Figure 3–19. 32-Bit PCI & 64-Bit Local-Side Single-Cycle Memory Write Target Transaction



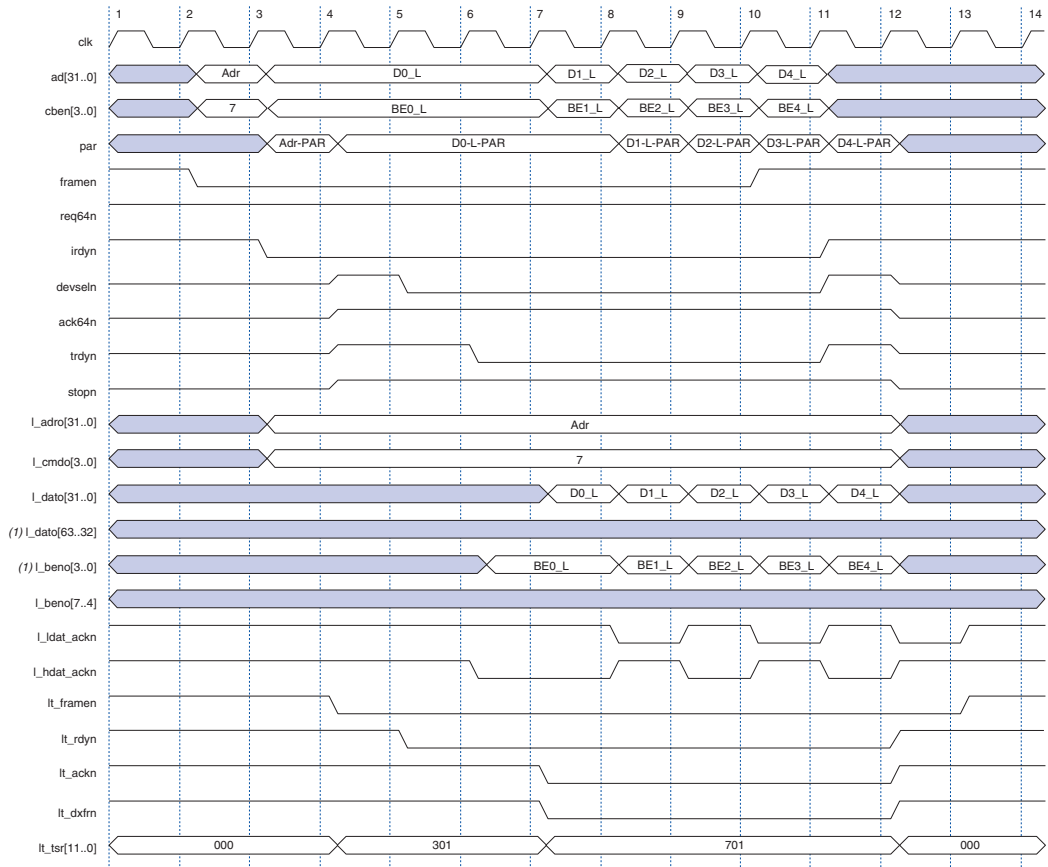
Note to Figure 3–19:

(1) Ignore this signal for this transaction.

Figure 3–20 shows a 32-bit burst memory write transaction; the events are the same for Figure 3–17. The main difference between the two figures is that in Figure 3–20 `l_ldat_ackn` and `l_hdat_ackn` toggle to indicate which DWORD is valid on the local side. In Figure 3–20, the high DWORD is transferred first because the address used is not a QWORD boundary. The `l_hdat_ackn` signal is asserted during clock cycle 6 and continues to be asserted until the first DWORD is transferred on the local side during clock cycle 7. The local side is informed that the pending transaction is a 32-bit burst because `lt_tsr[7]` is not asserted and `lt_tsr[9]` is asserted. If the local side cannot handle 32-bit burst transactions, it can disconnect after the first local transfer.

Figure 3–20 only applies to the `pci_mt64` and `pci_t64` functions. For the `pci_mt32` and `pci_t32` functions, Figure 3–17 reflects the waveforms for a 32-bit burst memory write transaction, excluding the 64-bit extension signals as noted.

Figure 3–20. 32-Bit PCI & 64-Bit Local-Side Burst Memory Write Target Transaction



Note to Figure 3–20:

- (1) Ignore this signal for this transaction.

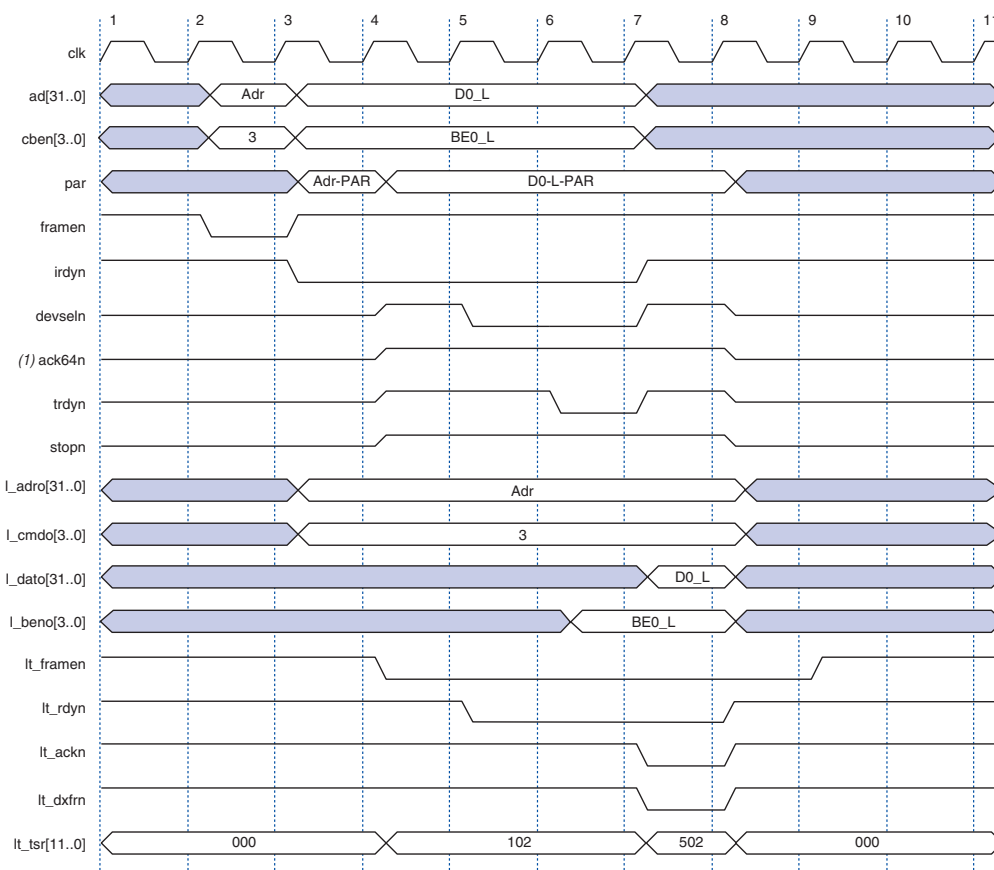
I/O Write Transactions

I/O write transactions by definition are 32 bits. Figure 3–21 shows a sample I/O write transaction. The sequence of events is the same as 32-bit single-cycle memory write transactions. The main distinction between the two transactions is the command on the `lt_cmdo[3..0]` bus.



The PCI MegaCore functions do not ensure that the combination of the `ad[1..0]` and `cbe[3..0]` signals is valid during the address phase of an I/O transaction. Local side logic should implement this functionality if performing I/O transactions. Refer to the *PCI Local Bus Specification, Revision 3.0* for more information on handling invalid combinations of these signals.

Figure 3–21. I/O Write Transaction



Configuration Write Transactions

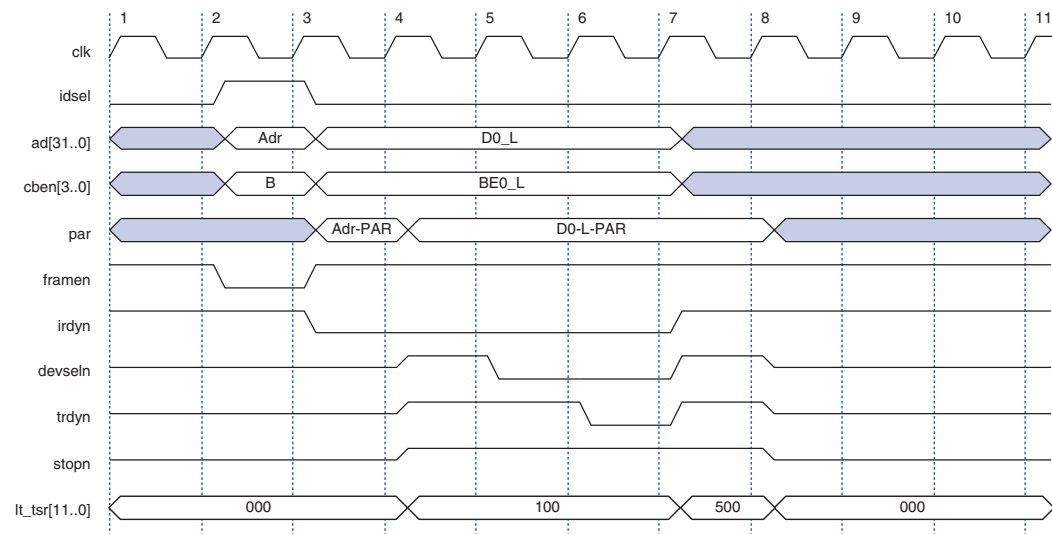
Configuration write transactions are 32 bits. Configuration cycles are automatically handled by the PCI MegaCore functions and do not require local side actions. Figure 3–22 shows a typical configuration write transaction. The configuration write transaction is similar to a 32-bit single-cycle transaction, except for the following:

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



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

Figure 3–22. 32-Bit 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 3.0*. 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.

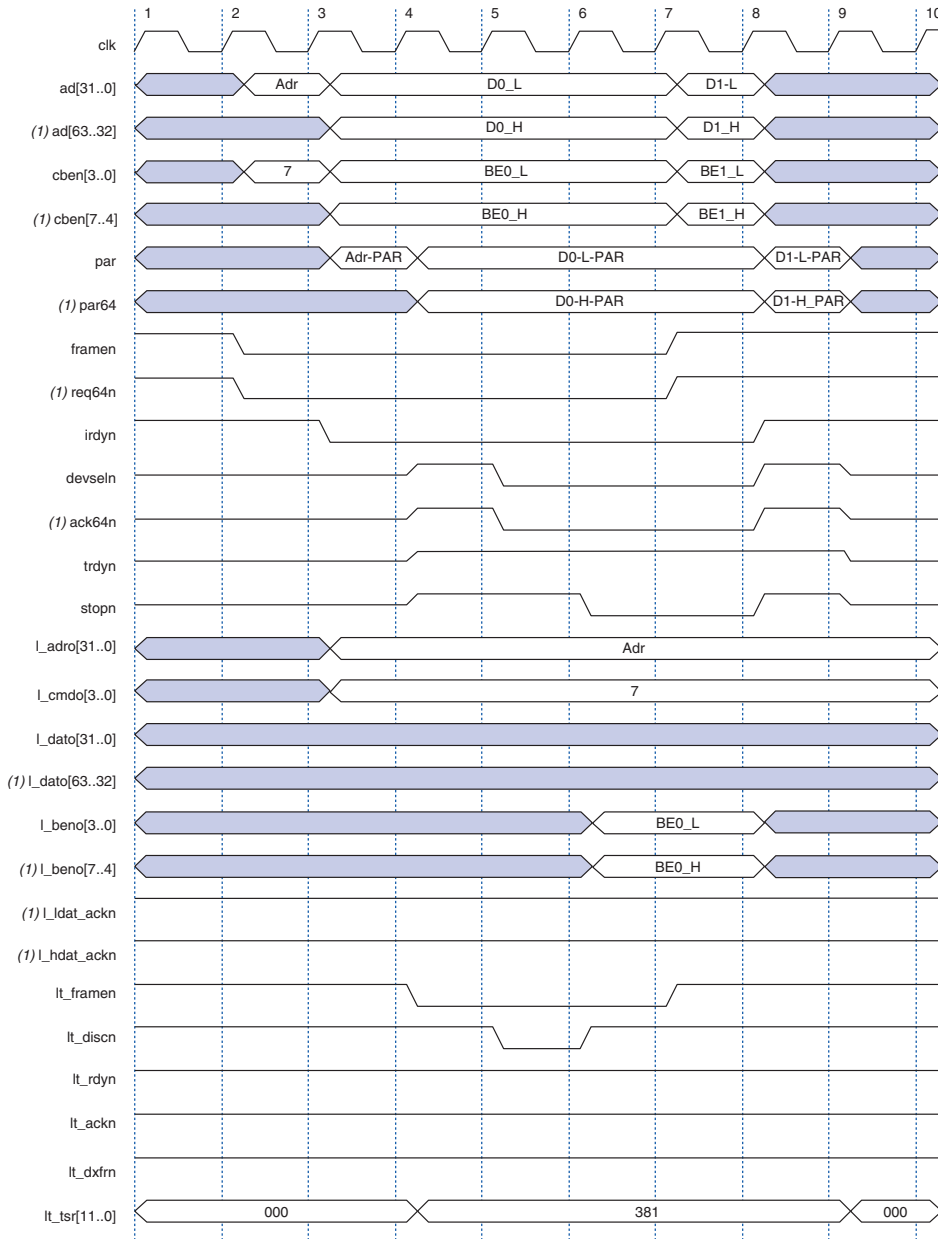
The local-side device can also request a target abort, which indicates that a catastrophic error has occurred in the device. This termination is requested by asserting `lt_abortn` during a target transaction other than a configuration transaction.



For more details on these termination types, refer to the *PCI Local Bus Specification, Revision 3.0*.

Retry

The local-side device can request a retry if, for example, the device cannot meet the initial latency requirement or because the local resource cannot transfer data. 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 phase 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 the PCI MegaCore function asserting the `trdyn` signal on the PCI bus. Therefore, asserting `lt_discn` forces a disconnect instead of a retry. In a read transaction, the local-side device can request a retry as long as data has not been transferred to the PCI MegaCore function. [Figure 3-23](#) applies to all PCI MegaCore functions, excluding the 64-bit signals as noted for `pci_mt32` and `pci_t32`.

Figure 3–23. Target Retry**Note to Figure 3–23:**(1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

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, one more data phase is completed while the PCI bus master finishes the transaction. A disconnect without data occurs when the target device deasserts `trdyn` while `stopn` is asserted, thus ensuring that no more data phases are completed in the transaction. Depending on the sequence of the `lt_rdyn` and `lt_discn` signals' assertion on the local side and the `irdyn` signal's assertion on the PCI side, the PCI MegaCore function issues either a disconnect with data or disconnect without data. Since disconnect with data or disconnect without data transactions depend upon the state of the `irdyn` signal, you must design your logic to disconnect after the specified number of DWORDs are transferred on the PCI bus. You can use `lt_ackn` and `lt_dxfrn` to check the number of DWORDs transferred on the local side and use `lt_tsr[10]` to check the number of DWORDs transferred on the PCI bus.

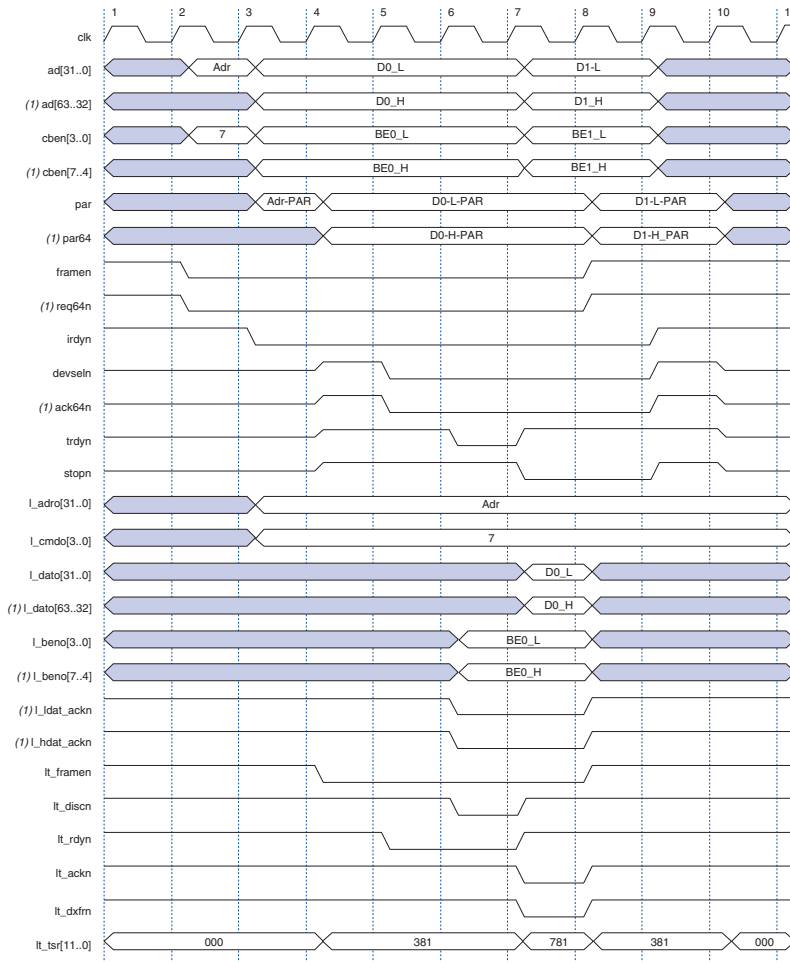


The *PCI Local Bus Specification, Revision 3.0* requires that a target device issues a disconnect if a burst transaction goes beyond its address range. In this case, the local-side device must request a disconnect. The local-side device must keep track of the current data transfer address; if the transfer exceeds its address range, the local side should request a disconnect by asserting `lt_discn`.

Figure 3–24 shows an example of a disconnect during a burst write transaction that ensures only a single data phase is completed.

Figure 3–24 applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`. In Figure 3–24 `lt_rdyn` is asserted in clock cycle 5 and `lt_discn` is asserted in clock cycle 6. This transaction informs the PCI MegaCore function that the local side is ready to accept data but also wants to disconnect. As a result, the PCI MegaCore function disconnects after one data phase.

Figure 3–24. Single Data Phase Disconnect in a Burst Write Transaction

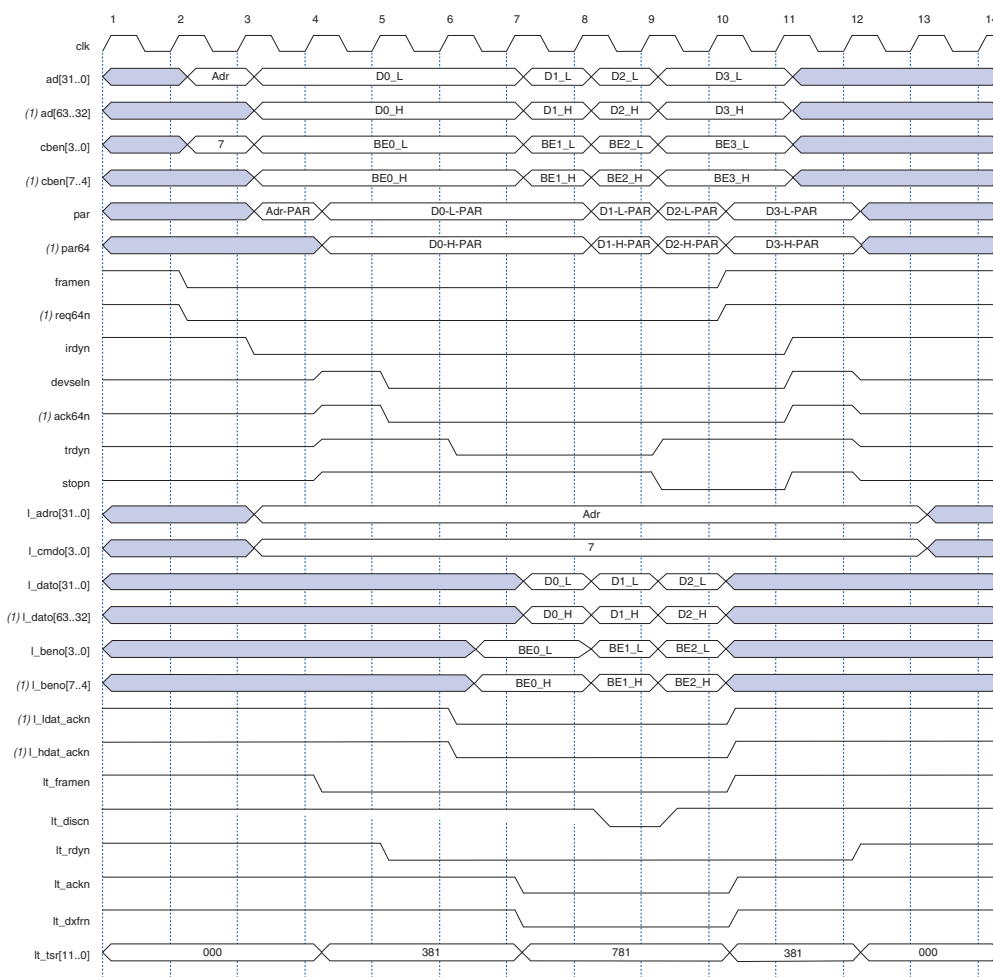


Note to Figure 3–24:

(1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

Figure 3–25 shows an example of a disconnect during a burst target write transaction where multiple data phases are completed. Figure 3–25 applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`. One additional data phase will be completed on the local side following the assertion of `lt_discn`.

Figure 3–25. Disconnect in a Burst Write Transaction

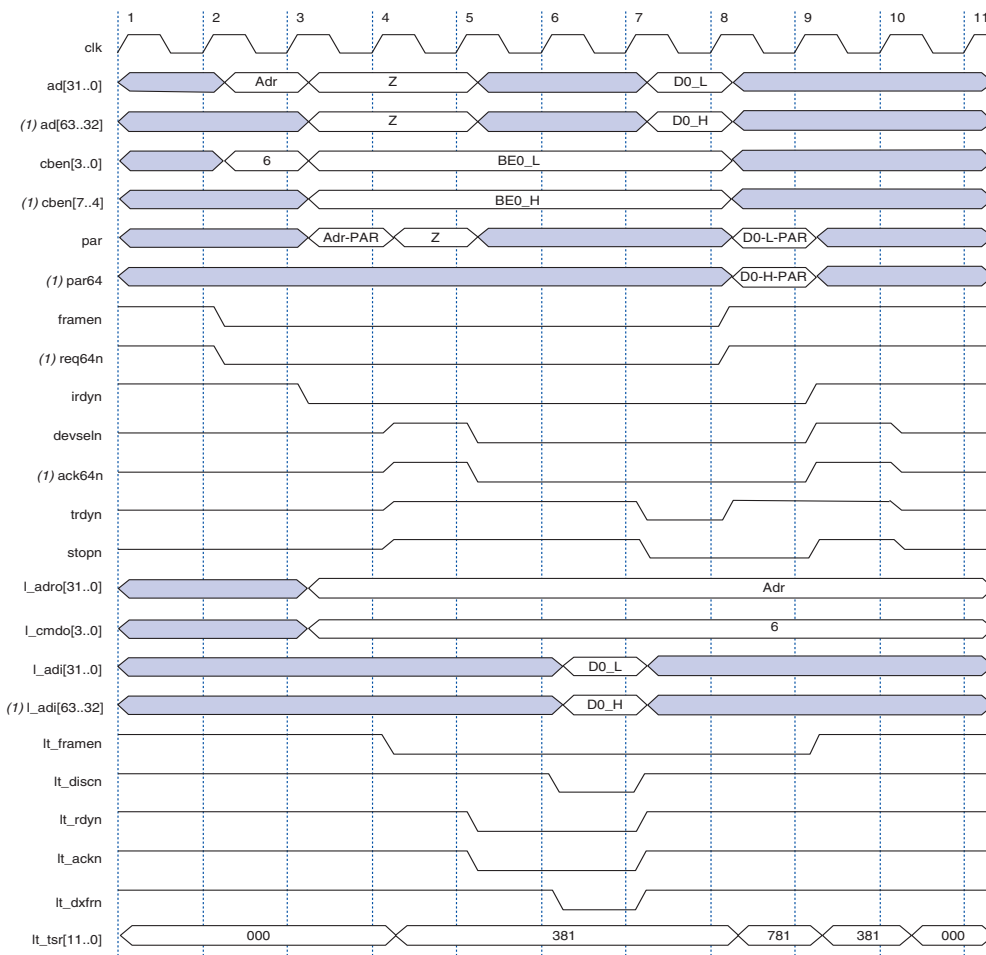


Note to Figure 3–25:

- (1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

Figure 3–26 shows an example of a disconnect during a burst read transaction that ensures only a single data phase is completed. In Figure 3–26, `lt_rdyn` is asserted in clock cycle 5 and `lt_discn` is asserted in clock cycle 6. This transaction informs the PCI MegaCore function that the local side is ready with data but also wants to disconnect. As a result the PCI MegaCore function disconnects after one data phase. This figure applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`.

Figure 3–26. Single Cycle Disconnect in a Burst Read Transaction

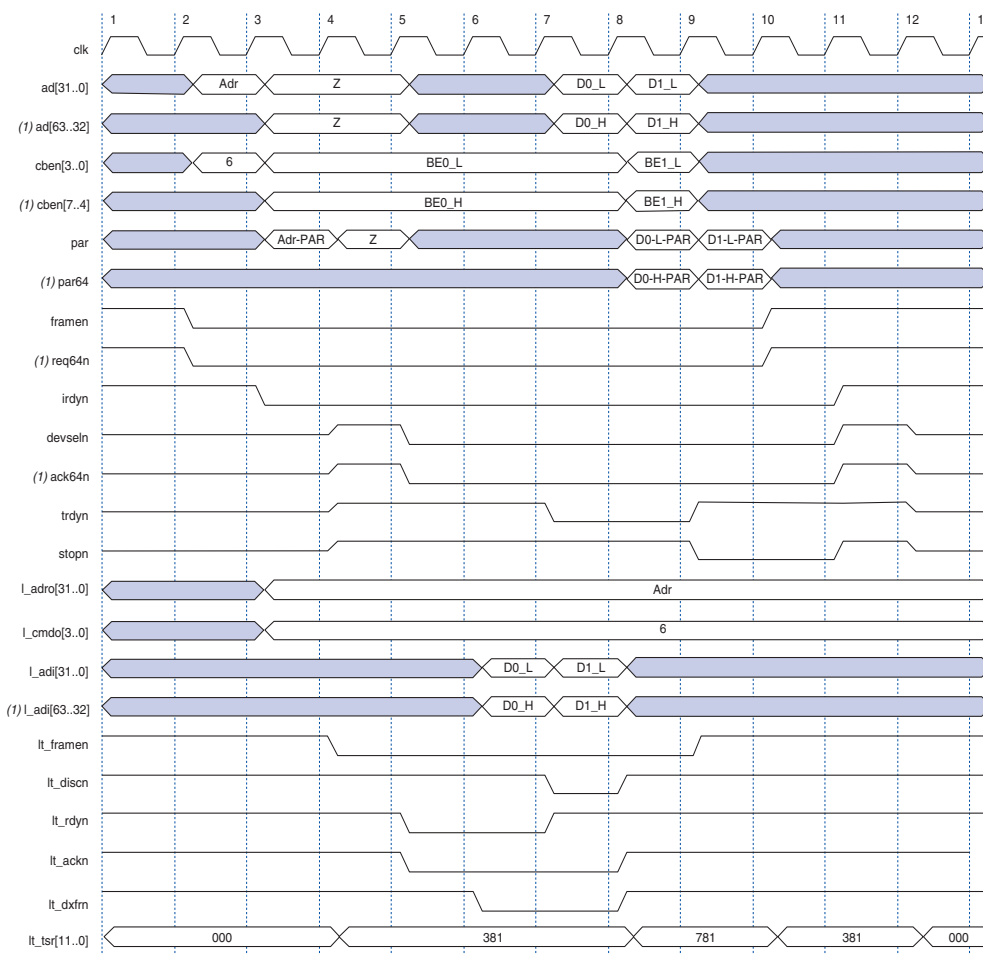


Note to Figure 3–26:

- (1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

Figure 3–27 shows an example of a disconnect during a burst target read transaction, and it applies to all PCI functions—excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`. During burst target read transactions, `lt_discn` should be asserted with the last data phase on the local side. The `lt_rdyn` signal is asserted during clock cycle 5 indicating that valid data will be available on the local side in clock cycle 6. Then, `lt_discn` is asserted in clock cycle 7 indicating the last data phase to be completed on the local side.

Figure 3–27. Disconnect in a Burst Read Transaction



Note to Figure 3–27:

- (1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

Figure 3–28 shows an example of a disconnect during a 32-bit read on non-QWORD aligned addresses.

Figure 3–28. 32-bit PCI & 64-bit Local Side Disconnect on Non-QWORD Aligned Address

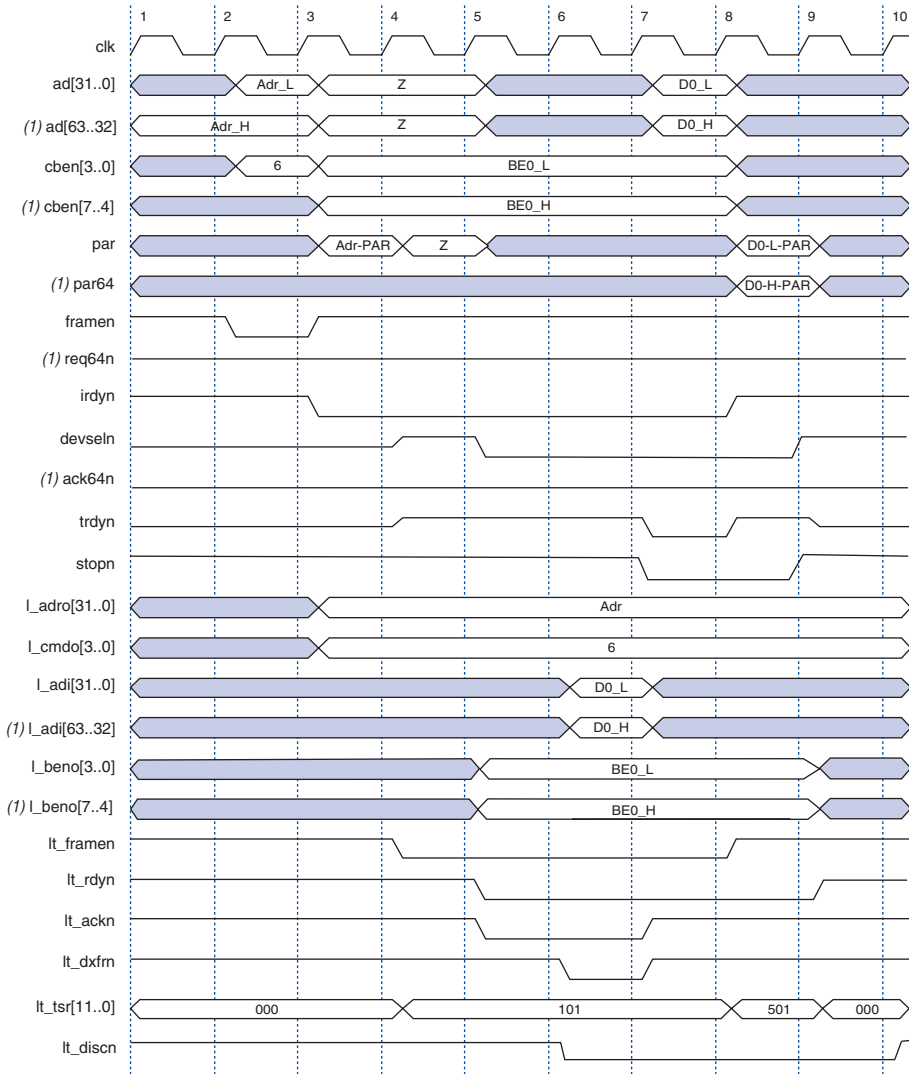
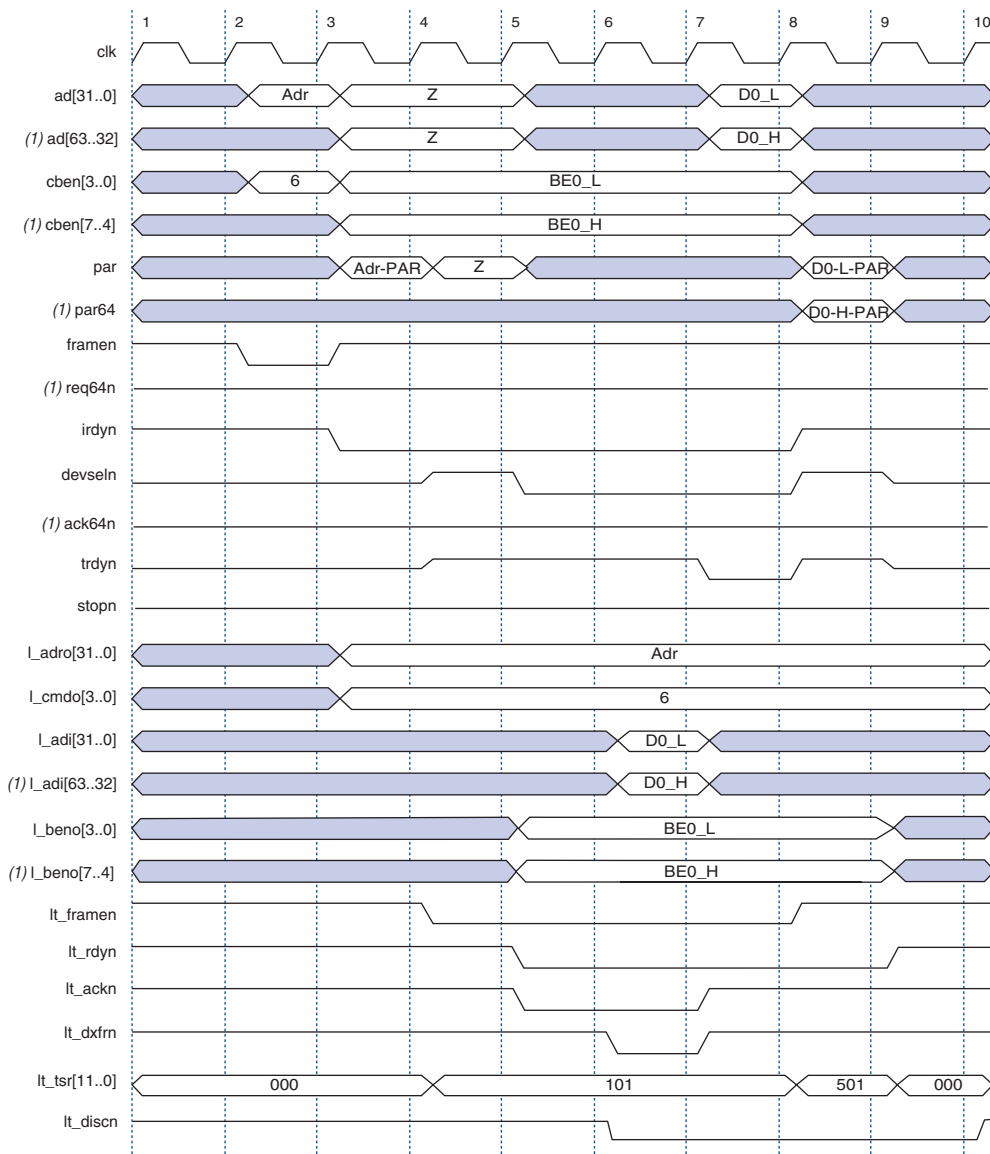


Figure 3–27 shows an example of a disconnect during a 32-bit read on QWORD aligned addresses. The PCI MegaCore function completes and does not disconnect even though `lt_discn` is asserted. This is an expected behavior of the PCI MegaCore function.

Figure 3–29. 32-bit PCI & 64-bit Local Side Disconnect on QWORD Aligned Address



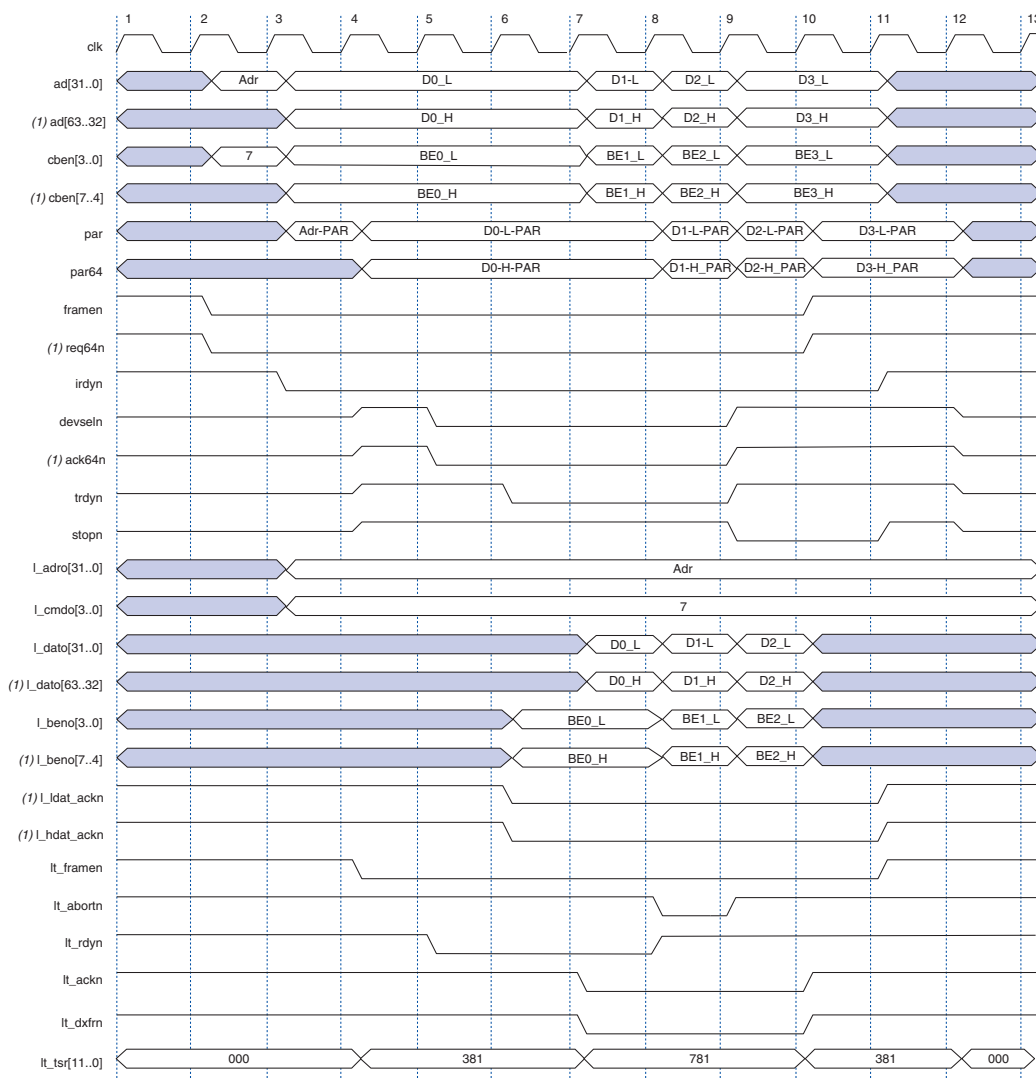
Target Abort

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

A target device issues an abort by deasserting `devseln` and `trdyn` and asserting `stopn`. A target device must set the `tabort_sig` bit in the PCI status register whenever it issues a target abort. Refer to “[Status Register](#)” on page 3–33 for more details. [Figure 3–30](#) shows the PCI MegaCore function issuing an abort during a burst write cycle. It applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`.



The *PCI Local Bus Specification, Revision 3.0* requires that a target device issues an abort if the target device shares bytes in the same `DWORD` with another device, and the byte enable combination received byte requests outside its address range. This condition most commonly occurs during I/O transactions. The local-side device must ensure that this requirement is met, and if it receives this type of transaction, it must assert `lt_abortn` to request a target abort termination.

Figure 3–30. Target Abort**Note to Figure 3–30:**

(1) This signal is not applicable to either the `pci_mt32` or `pci_t32` MegaCore functions.

Additional Design Guidelines for Target Transactions

Altera recommends that the local-side application deassert the `lt_rdyn` signal after the target transaction completes to avoid false triggering of internal state machines if the next target transaction begins immediately. You can detect that the current target transaction has completed if `lt_ackn` and `lt_tsr[8]` are both deasserted.

Asserting wait states on the last data phase of a PCI write transaction can cause a data loss if another PCI transaction begins during the wait states. This is because the PCI MegaCore function has only one register pipeline phase that is used to register the PCI data. To prevent data loss, the local side design should load the data into a holding register if a wait state is needed on the last data phase.

The local-side design must ensure that PCI latency rules are not violated while the PCI MegaCore function waits to transfer data. If the local-side design is unable to meet the latency requirements, it must assert `lt_discn` to request that the PCI 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 clock cycles, and the subsequent data phases must not take more than 8 clock cycles to complete.

The *PCI Local Bus Specification, Revision 3.0* requires that a target device issues a disconnect if a burst transaction goes beyond its address range. In this case, the local-side device must request a disconnect. The local-side device must keep track of the current data transfer address; if the transfer exceeds its address range, the local side should request a disconnect by asserting `lt_discn`.

The *PCI Local Bus Specification, Revision 3.0* requires that a target device issues an abort if the target device shares bytes in the same DWORD with another device, and the byte enable combination received byte requests outside its address range. This condition most commonly occurs during I/O transactions. The local-side device must ensure that this requirement is met, and if it receives this type of transaction, it must assert `lt_abortn` to request a target abort termination.

Master Mode Operation

This section describes all supported master transactions for both the `pci_mt64` and `pci_mt32` functions. Although this section includes waveform diagrams showing typical PCI cycles in master mode for the `pci_mt64` function, the waveforms also apply to the `pci_mt32` function. [Table 3–37](#) lists the PCI and local side signals that apply for each

PCI MegaCore function.

Table 3–37. PCI MegaCore Function Signals (Part 1 of 2)		
Signal Name	pci_mt64	pci_mt32
PCI Signals		
clk	✓	✓
rstn	✓	✓
gntn	✓	✓
reqn	✓	✓
ad[63..0]	✓	ad[31..0]
cben[7..0]	✓	cben[3..0]
par	✓	✓
par64	✓	
idsel	✓	✓
framen	✓	✓
req64n	✓	
irdyn	✓	✓
devseln	✓	✓
ack64n	✓	
trdyn	✓	✓
stopn	✓	✓
perrn	✓	✓
serrn	✓	✓
intan	✓	✓
Local-side Data Path Signals		
l_adi[63..0]	✓	l_adi[31..0]
l_cbeni[7..0]	✓	l_cbeni[3..0]
l_adro[63..0]	✓	l_adro[31..0]
l_dato[63..0]	✓	l_dato[31..0]
l_beno[7..0]	✓	l_beno[3..0]
l_cmdo[3..0]	✓	✓
l_ldat_ackn	✓	
l_hdat_ackn	✓	

Table 3–37. PCI MegaCore Function Signals (Part 2 of 2)

Signal Name	pci_mt64	pci_mt32
Target Local-side Control Signals		
lt_abortn	✓	✓
lt_discn	✓	✓
lt_rdyn	✓	✓
lt_framen	✓	✓
lt_ackn	✓	✓
lt_dxfrn	✓	✓
lt_tsr[11..0]	✓	✓
lirqn	✓	✓
cache[7..0]	✓	✓
cmd_reg[5..0]	✓	✓
stat_reg[5..0]	✓	✓
Master Local-side Control Signals		
lm_req32n	✓	✓
lm_req64n	✓	
lm_lastn	✓	✓
lm_rdyn	✓	✓
lm_adr_ackn	✓	✓
lm_ackn	✓	✓
lm_dxfrn	✓	✓
lm_tsr[9..0]	✓	✓

The PCI MegaCore functions support both 64-bit and 32-bit transactions. The pci_mt64 function supports the following 64-bit PCI memory transactions:

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



64-bit single-cycle memory write transactions are only supported if the **Assume ack64n Response** option is turned on in the **Parameterize - PCI Compiler** wizard. For more information on the **Assume ack64n Response** option, refer to [“Assume ack64n Response” on page 2–6](#).

The `pci_mt64` and `pci_mt32` functions support the following 32-bit PCI transactions:

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

A master operation begins when the local-side master interface asserts the `lm_req64n` signal to request a 64-bit transaction or the `lm_req32n` signal to request a 32-bit transaction. The PCI function asserts the `reqn` signal to the PCI bus arbiter to request bus ownership. When the PCI bus arbiter grants the PCI function bus ownership by asserting the `gntn` signal, the local side is alerted and must provide the address and command.

Once the PCI MegaCore function has acquired mastership of the PCI bus, the function asserts `framen` to indicate the beginning of a bus transaction, which is referred to as the address phase. During the address phase, the function drives the address and command signals on the `ad[31..0]` and `cben[3..0]` buses. If the local side requests a 64-bit transaction when using the `pci_mt64` function, the function asserts the `req64n` and `framen` signals at the same time. After the PCI MegaCore function master device has completed the address phase, the master waits for the target devices on the bus to decode the address and claim the transaction by asserting `devseln`. With a 64-bit transaction, the target device asserts `ack64n` and `devseln` at the same time if it can accept the 64-bit transaction. If the target device does not assert `ack64n`, the master device completes a 32-bit transaction.

Both the `pci_mt64` and `pci_mt32` functions support single-cycle and memory burst transactions. In a read transaction, data is transferred from the PCI target device to the local-side device. In a write transaction, data is transferred from the local side to the PCI target device. A memory transaction can be terminated by the local side or by the PCI target device. When the PCI target terminates the transaction, the local side is informed of the conditions of the termination by specific bits in the `lm_tsr[9..0]` bus. The function treats memory write and invalidate, memory read multiple, and memory read line commands in a similar manner to the corresponding memory write/read commands. Therefore, the local side must implement any special handling required by these commands. The function outputs the cache line size register value to the local side for this purpose.

The `pci_mt64` and `pci_mt32` functions can generate transactions as specified in Table 3–11. When the local side requests I/O or configuration cycles, the function automatically issues a 32-bit single-cycle read/write transaction.



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

PCI Bus Parking

By asserting the `gntn` signal of a master device that has not requested bus access, the PCI bus arbiter may park on any master device when the bus is idle. In accordance with the *PCI Local Bus Specification, Revision 3.0*, if the arbiter parks on `pci_mt64` or `pci_mt32`, the function drives the `ad[31..0]`, `cbe[3..0]` and `par` signals.

If the arbiter has parked the bus on `pci_mt64` or `pci_mt32` and the local side requests a transaction, the `request` bit (i.e., `lm_tsr[0]`) will not be asserted on the local side. The local state machine will immediately assert the `grant` bit (i.e., `lm_tsr[1]`).

Design Consideration

The arbiter may remove the `gntn` signal after the local side has asserted `lm_req64n` or `lm_req32n` to request the bus, but before the master function has been able to assert the `framen` signal to claim the bus. In this case, the `lm_tsr` signals will transition from the grant state (i.e., `lm_tsr[1]` asserted) back to the request state (i.e., `lm_tsr[0]` asserted) until the arbiter grants the bus to the requesting function again. In systems where this situation may occur, the local-side logic should hold the address and command on the `l_adi[31..0]` and `l_cbeni[3..0]` buses until the address phase bit (i.e., `lm_tsr[2]`) is asserted to ensure that the `pci_mt64` or `pci_mt32` function has assumed mastership of the bus and that the current address and command have been transferred.

Master Read Transactions

This section describes the behavior of the PCI MegaCore functions in the following types of master read transactions:

- Memory read
- I/O and configuration read

Memory Read Transactions

The PCI MegaCore functions support the following types of matched bus width and mismatched bus width memory read transactions in master mode:

- Burst memory read
- Single-cycle memory read
- Mismatched bus width memory read



Mismatched bus-width transactions are 32-bit PCI transactions performed by the `pci_mt64` MegaCore function.

For each type of transaction, the following sequence of events is the same:

1. The local side asserts `lm_req32n` to request a 32-bit transaction (or `lm_req64n` to request a 64-bit transaction.) Consequently, the PCI side asserts `reqn` to request bus ownership from the PCI arbiter.
2. When the PCI arbiter grants bus ownership by asserting the `gntn` signal, the PCI side asserts `lm_adr_ackn` on the local side to acknowledge the transaction address and command. During the same clock cycle when `lm_adr_ackn` is asserted, the local side must provide the address on `l_adi[31..0]` and the command on `l_cbeni[3..0]`. At the same time, the PCI side turns on the drivers for `framen` (and `req64n` for 64-bit transactions.)
3. The PCI side begins the PCI address phase. During the PCI address phase, the local side must provide the byte enables for the transaction on the `l_cbeni` bus. At the same time, the PCI side turns on the driver for `irdyn`.



The PCI MegaCore function uses the initial byte enable values throughout the transaction, and ignores any changes to the signals on the `l_cbeni` bus after this phase. If the **Allow Variable Byte Enables During Burst Transactions** option is turned on in the **Parameterize - PCI Compiler** wizard, you must keep the byte enables constant throughout the transaction. Typically the byte enable values are set to 0x00 for master read transactions.

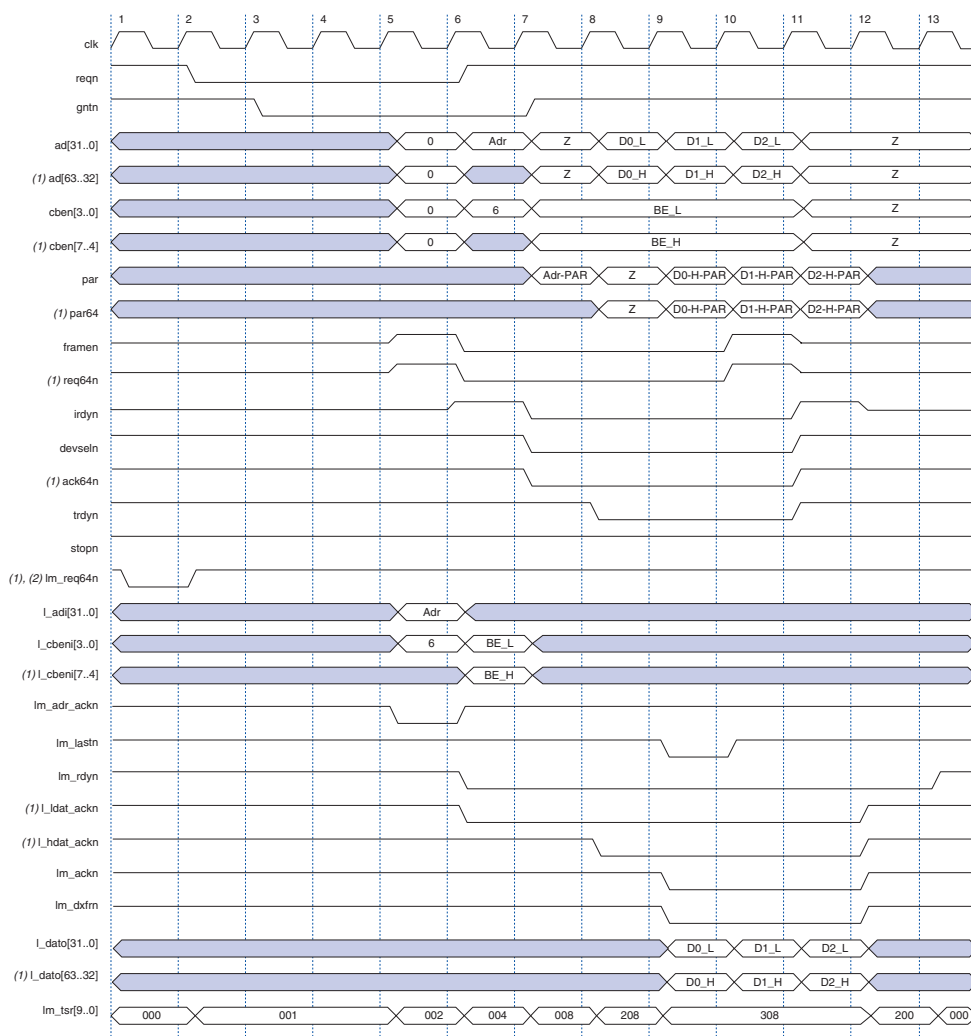
4. A turn-around cycle on the `ad` bus occurs during the clock cycle immediately following the address phase. During the turn-around cycle, the PCI side tri-states the `ad` bus, but drives the correct byte enables on the `cben` bus for the first data phase. This process is necessary because the `pci_mt64` function must release the bus so another PCI agent can drive it.
5. A PCI target asserts `devseln` to claim the transaction. One or more data phases follow, depending on the type of read transaction.

The `pci_mt64` and `pci_mt32` functions treat 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.

Burst Memory Read Master Transactions

Figure 3–31 shows the waveform for a 64-bit zero-wait state burst memory read master transaction. In this transaction, three data words are transferred from the PCI side to the local side. The 64-bit extension signals are not applicable to the `pci_mt32` function.

Figure 3–31. Zero-Wait State Burst Memory Read Master Transaction



Notes to Figure 3–31:

- (1) This signal is not applicable to the `pci_mt32` MegaCore function.
- (2) For `pci_mt32`, `lm_req32n` should be substituted for `lm_req64n` for 32-bit master transactions.

Table 3–38 shows the sequence of events for a 64-bit zero-wait state burst memory read master transaction. The 64-bit extension signals are not applicable to the `pci_mt32` function.

Table 3–38. Zero-Wait State Burst Memory Read Master Transaction (Part 1 of 3)

Clock Cycle	Event
1	The local side asserts <code>lm_req64n</code> to request a 64-bit transaction.
2	The function outputs <code>reqn</code> to the PCI bus arbiter to request bus ownership. At the same time, the function asserts <code>lm_tsr[0]</code> to indicate to the local side that the master is requesting the PCI bus.
3	The PCI bus arbiter asserts <code>gntn</code> to grant the PCI bus to the function. Although Figure 3–31 shows that the grant occurs immediately and the PCI bus is idle at the time <code>gntn</code> is asserted, this action may not occur immediately in a real transaction. Before the function proceeds, it waits for <code>gntn</code> to be asserted and the PCI bus to be idle. A PCI bus idle state occurs when both <code>framen</code> and <code>irdyn</code> are deasserted.
5	The function turns on its output drivers, getting ready to begin the address phase. The function also asserts <code>lm_adr_ackn</code> to indicate to the local side that it must provide the address and command for the transaction. During the same clock cycle, the local side must provide the PCI address on <code>l_adi[31..0]</code> and the PCI command on <code>l_cbeni[3..0]</code>. The function continues to assert its <code>reqn</code> signal until the end of the address phase. The function also asserts <code>lm_tsr[1]</code> to indicate to the local side that the PCI bus has been granted.
6	The function begins the 64-bit memory read transaction with the address phase by asserting <code>framen</code> and <code>req64n</code>. At the same time, the local side must provide the byte enables for the transaction on the <code>l_cbeni</code> bus. The PCI MegaCore function uses this byte enable value throughout the transaction, and ignores any changes to the signals on the <code>l_cbeni</code> bus after this clock cycle. If the Allow Variable Byte Enables During Burst Transactions option is turned on in the Parameterize - PCI Compiler wizard, you must keep the byte enables constant throughout the rest of the transaction. Typically, the byte enable values are set to 0x00 for master read transactions. The local side also asserts <code>lm_rdyn</code> to indicate that it is ready to accept data. The function asserts <code>lm_tsr[2]</code> to indicate to the local side that the PCI bus is in its address phase. If the arbiter deasserts <code>gntn</code> in less than 3 clock cycles, the PCI MegaCore function does not assert <code>lm_tsr[2]</code> in this clock cycle. For recommendations of how to accommodate scenarios where the arbiter deasserts <code>gntn</code> in less than three clock cycles, refer to “Design Consideration” on page 3–92 for more information.

Table 3–38. Zero-Wait State Burst Memory Read Master Transaction (Part 2 of 3)

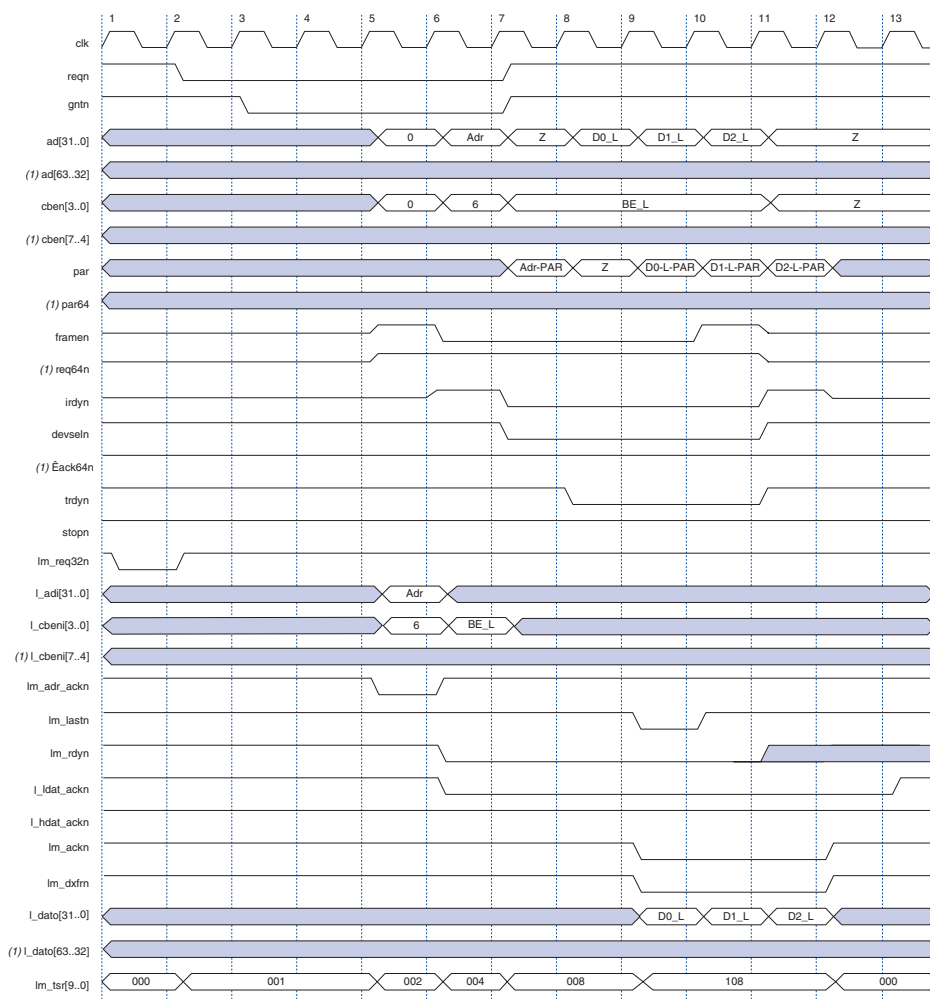
Clock Cycle	Event
7	The function asserts <code>irdyn</code> to inform the target that the function is ready to receive data. On the first data phase the function asserts <code>irdyn</code> regardless of whether the <code>lm_rdyn</code> signal is asserted on the local side to indicate that the local side is ready to accept data. For subsequent data phases, the function does not assert <code>irdyn</code> unless the local side is ready to accept data. The target claims the transaction by asserting <code>devseln</code>. In this case, the target performs a fast address decode. The target also asserts <code>ack64n</code> to inform the function that it can transfer 64-bit data. During this clock cycle, the function also asserts <code>lm_tsr[3]</code> to inform the local side that it is in data phase mode.
8	The target asserts <code>trdyn</code> to inform the function that it is ready to transfer data. Because the function has already asserted <code>irdyn</code>, a data phase is completed on the rising edge of clock cycle 9. At the same time, <code>lm_tsr[9]</code> is asserted to indicate to the local side that the target can transfer 64-bit data.
9	The function asserts <code>lm_ackn</code> to inform the local side that the function has registered data from the PCI side on the previous cycle and is ready to send the data to the local side master interface. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code>, <code>l_ldat_ackn</code>, and <code>l_hdat_ackn</code> signals indicate to the local side that valid data is available on the <code>l_data</code> bus. Because <code>irdyn</code> and <code>trdyn</code> are asserted, another data phase is completed on the PCI side on the rising edge of clock cycle 10. On the local side, the <code>lm_lastn</code> signal is asserted. Because <code>lm_lastn</code>, <code>irdyn</code>, and <code>trdyn</code> are asserted during this clock cycle, this action guarantees to the local side that, at most, two more data phases will occur on the PCI side: one during this clock cycle and another on the following clock cycle (clock cycle 10). The last data phase on the PCI side takes place during clock cycle 10. The function also asserts <code>lm_tsr[8]</code> in the same clock cycle to inform the local side that a successful data transfer has occurred on the PCI bus during the previous clock cycle.

Table 3–38. Zero-Wait State Burst Memory Read Master Transaction (Part 3 of 3)

Clock Cycle	Event
10	Because <code>lm_lastn</code> was asserted and a data phase was completed in the previous cycle, <code>framen</code> and <code>req64n</code> are deasserted, while <code>irdyn</code> and <code>trdyn</code> are asserted. This action indicates that the last data phase is completed on the PCI side on the rising edge of clock cycle 11. On the local side, the function continues to assert <code>lm_ackn</code>, informing the local side that the function has registered data from the PCI side on the previous cycle and is ready to send the data to the local side master interface. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code>, <code>l_ldat_ackn</code>, and <code>l_hdat_ackn</code> signals indicate to the local side that another valid data is available on the <code>l_dat0</code> bus. The local side has now received two valid 64-bit data. The function continues to assert <code>lm_tsr[8]</code> informing the local side that a successful data transfer has occurred on the PCI bus during the previous clock cycle.
11	On the PCI side, <code>irdyn</code>, <code>devseln</code>, <code>ack64n</code>, and <code>trdyn</code> are deasserted, indicating that the current transaction on the PCI side is completed. There will be no more PCI data phases. On the local side, the function continues to assert <code>lm_ackn</code>, informing the local side that the function has registered data from the PCI side on the previous cycle and is ready to send the data to the local-side master interface. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code>, <code>l_ldat_ackn</code>, and <code>l_hdat_ackn</code> signals indicate to the local side that data on the <code>l_dat0</code> bus is valid. The local side has now received three 64-bit words of data. Because the local side has received all the data that was registered from the PCI side, the local side can now deassert <code>lm_rdyn</code>. Otherwise, if there is still some data that has not been transferred from the PCI side to the local side, <code>lm_rdyn</code> must continue to be asserted. The function continues to assert <code>lm_tsr[8]</code> informing the local side that a successful data transfer has occurred on the PCI bus during the previous clock cycle.
12	The function deasserts <code>lm_tsr[3]</code>, informing the local side that the data transfer mode is completed. Therefore, <code>lm_ackn</code> and <code>lm_dxfrn</code> are also deasserted.

Figure 3–32 shows the same transaction as in Figure 3–31, but the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. This figure applies to both `pci_mt64` and `pci_mt32`, excluding the 64-bit extension signals as noted for `pci_mt32`. The `pci_mt64` function does not assert `req64n` on the PCI side. Therefore, the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid.

Figure 3–32. 32-Bit PCI & 32-Bit Local-Side Master Burst Memory Read Transaction

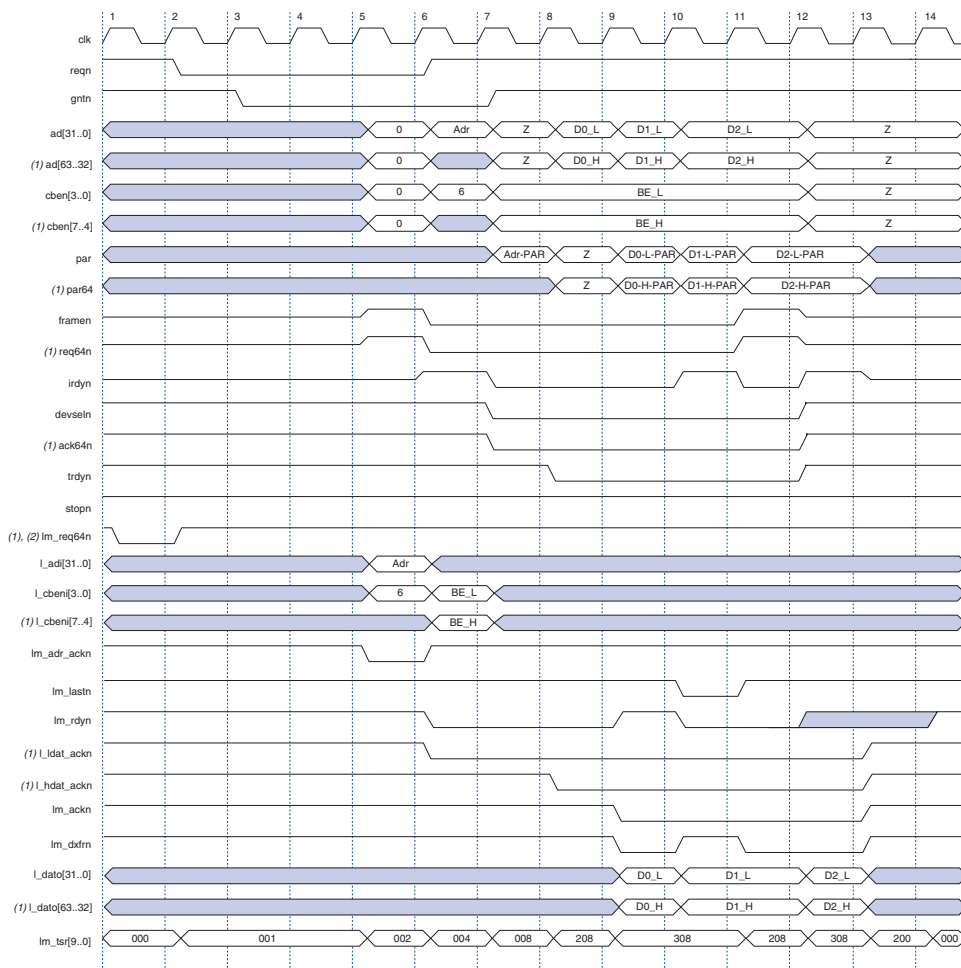


Note to Figure 3–32:

(1) This signal is not applicable to the `pci_mt32` MegaCore function.

Figure 3–33 shows the same transaction as in Figure 3–31 with the local side inserting a wait state. This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`.

Figure 3–33. Burst Memory Read Master Transaction with Local-Side Wait State

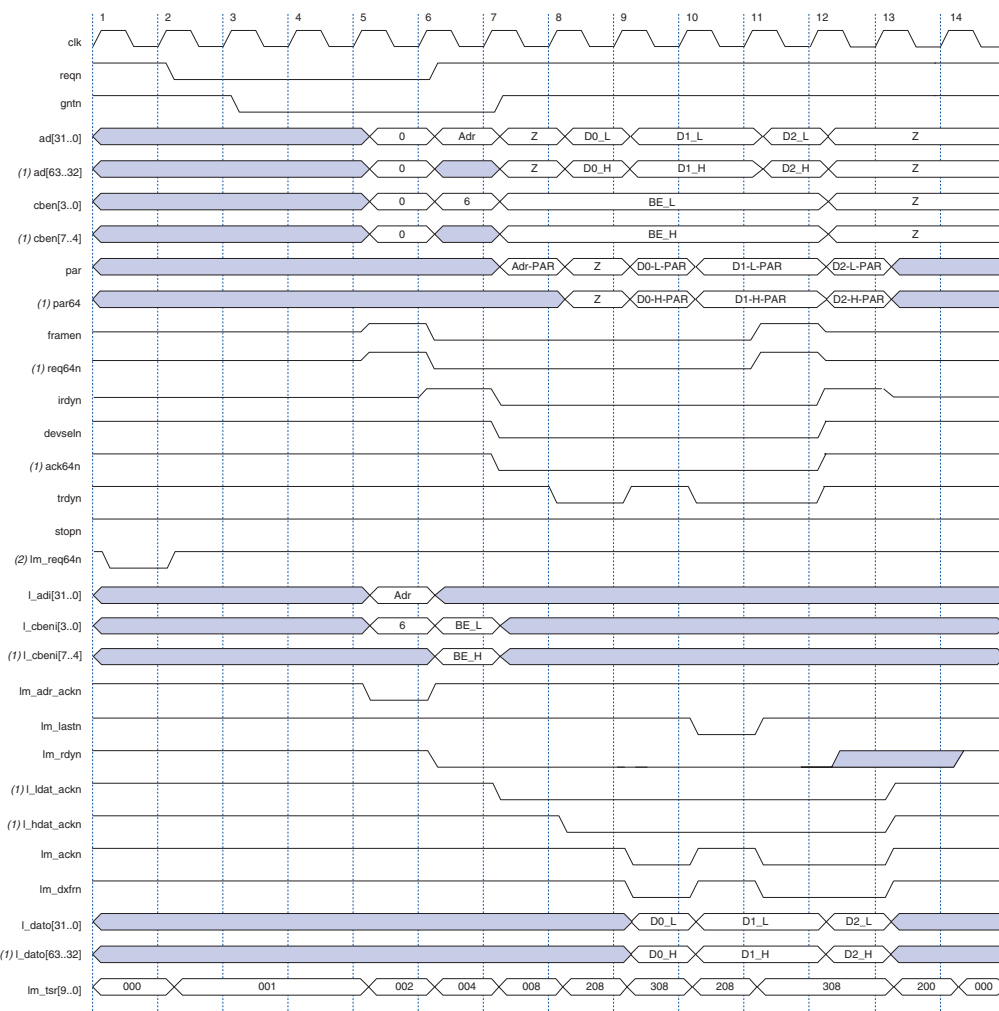


Notes to Figure 3–33:

- (1) This signal is not applicable to the `pci_mt32` MegaCore function.
- (2) For `pci_mt32`, `lm_req32n` should be substituted for `lm_req64n` for 32-bit master transactions.

The local side deasserts `lm_rdyn` in clock cycle 9. Consequently, on the following clock cycle (clock cycle 10), the `pci_mt64` function suspends data transfer on the local side by deasserting the `lm_dxfrn` signal and on the PCI side by deasserting the `irdyn` signal.

Figure 3–34 shows the same transaction as in Figure 3–31 with the PCI bus target inserting a wait state. This figure applies to both `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`. The PCI side inserts a wait state by deasserting `trdyn` in clock cycle 9. Consequently, on the following clock cycle (clock cycle 10), the function deasserts the `lm_ackn` and `lm_dxfrn` signal on the local side. Data transfer is suspended on the PCI side in clock cycle 9 and on the local side in clock cycle 10.

Figure 3–34. Burst Memory Read Master Transaction with PCI-Side Wait State**Notes to Figure 3–34:**

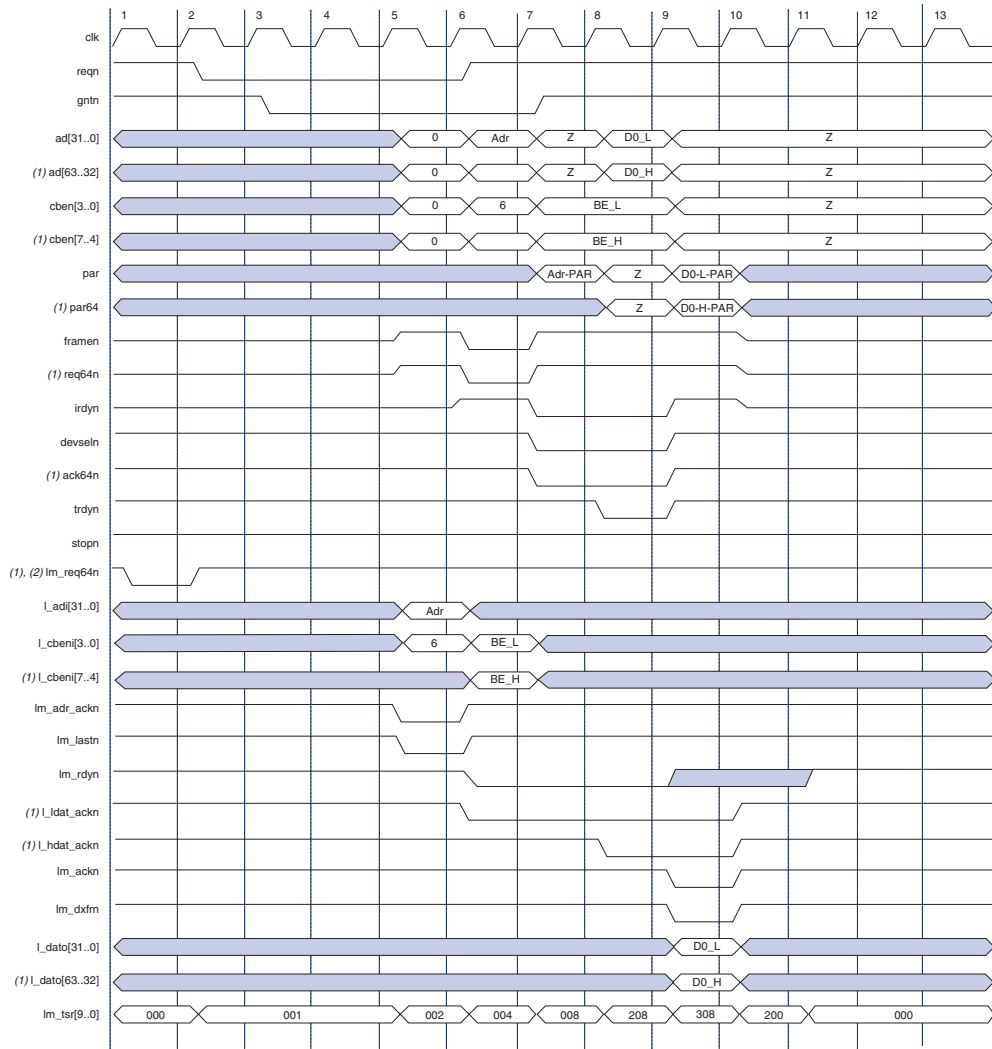
- (1) This signal is not applicable to the pci_mt32 MegaCore function.
- (2) For pci_mt32, lm_req32n should be substituted for lm_req64n for 32-bit master transactions.

Single-Cycle Memory Read Master Transaction

Figure 3–35 shows a 64-bit single-cycle memory read master transaction. Figure 3–35 shows the same transaction as in Figure 3–31 with just one data phase. This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`. In clock cycle 6, `framen` and `req64n` are asserted to begin the address phase. At the same time, the local side should assert the `lm_lastn` signal on the local side to indicate that it wants to transfer only one 64-bit data word. In a real application, in order to indicate a single-cycle 64-bit data transfer, the `lm_lastn` signal can be asserted on any clock cycle between the assertion of `lm_req64n` and the address phase.



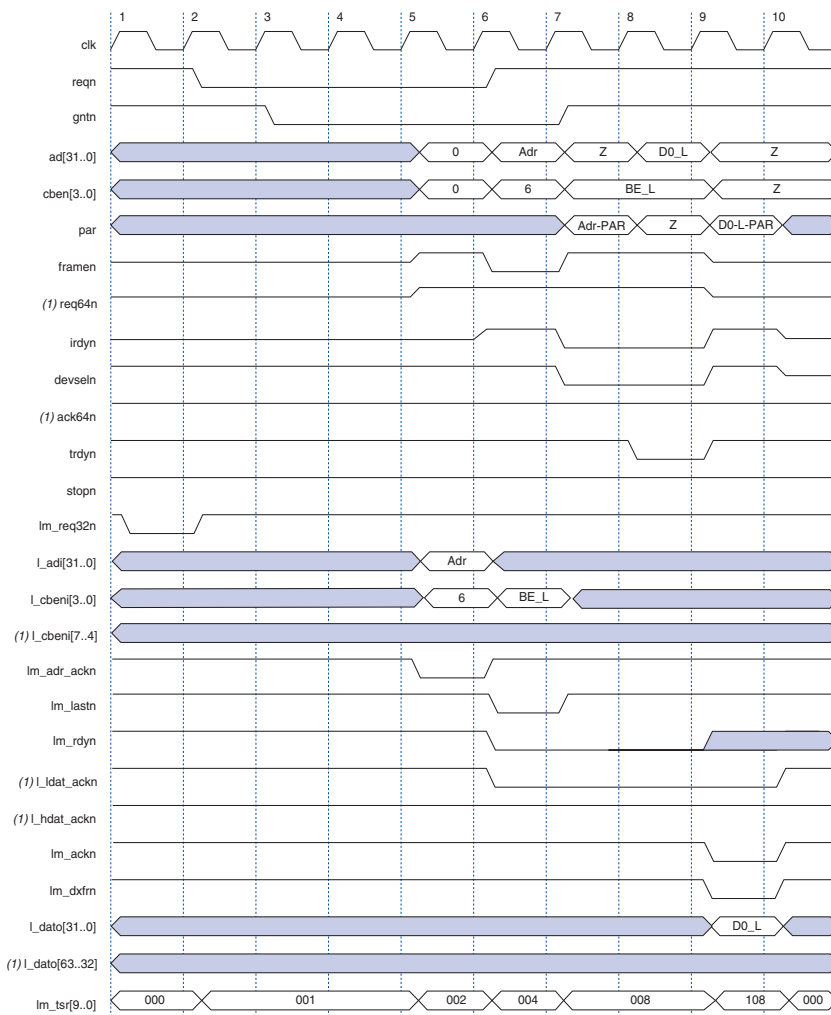
If your application is a system that has only 64-bit PCI devices and the local side wants to transfer one 64-bit data word, Altera recommends that you perform a 64-bit single-cycle memory read master transaction. However, if your application is a system that has 32-bit and 64-bit PCI devices and the local side wants to transfer one 64-bit data word, Altera recommends that you perform a 32-bit burst memory read transaction.

Figure 3–35. 64-Bit Single-Cycle Memory Read Master Transaction**Notes to Figure 3–35:**

- (1) This signal is not applicable to the pci_mt32 MegaCore function.
- (2) For pci_mt32, lm_req32n should be substituted for lm_req64n for 32-bit master transactions.

Figure 3–36 shows a 32-bit single cycle memory read master transaction. The transaction shown in Figure 3–36 is the same as shown in Figure 3–35, except that in Figure 3–36 the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`.

Figure 3–36. 32-Bit PCI & 32-Bit Local-Side Single-Cycle Memory Read Master Transaction



Note to Figure 3–36:

(1) This signal is not applicable to the `pci_mt32` MegaCore function.

Mismatched Bus Width Burst Memory Read Master Transactions

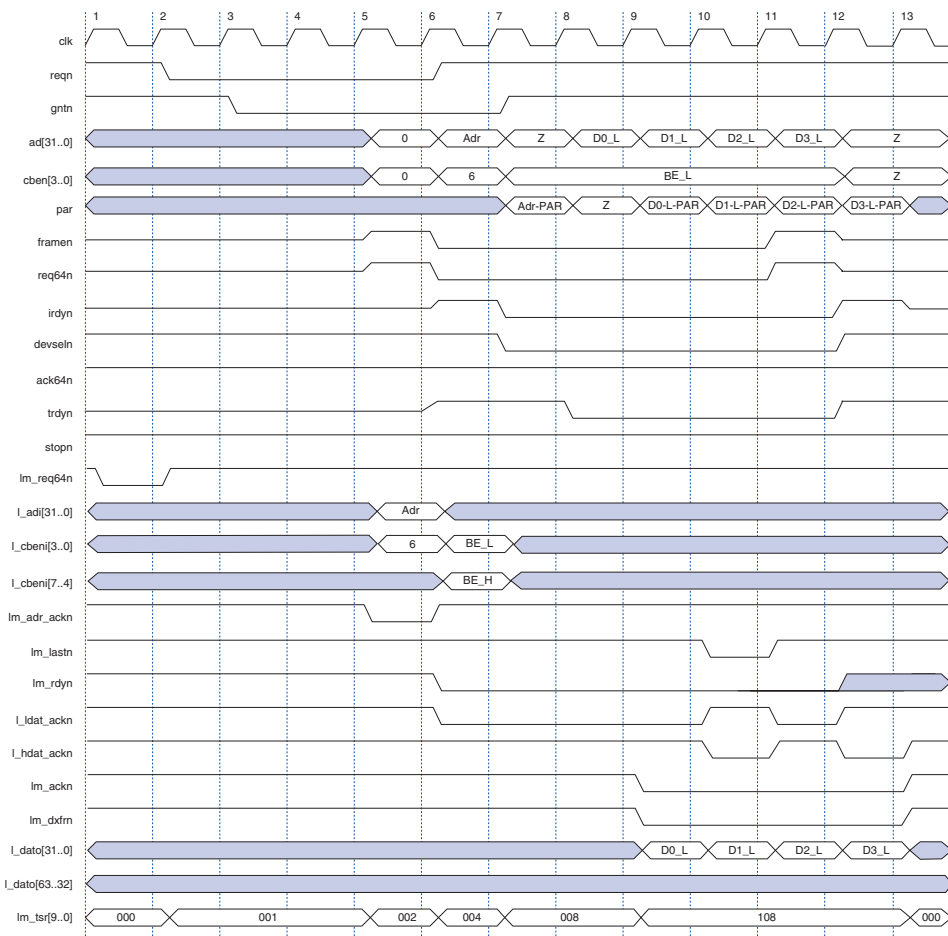
The following description applies only to the `pci_mt64` MegaCore functions handling mismatched bus width memory read master transactions.

Figure 3–37 shows a 32-bit PCI and 64-bit local side burst memory read master transaction. The events shown in Figure 3–37 are the same as those shown in Figure 3–31. In this transaction, the local-side master interface requests a 64-bit transaction by asserting `lm_req64n`. The `pci_mt64` function asserts `req64n` on the PCI side. However, the PCI target cannot transfer 64-bit data, and therefore does not assert `ack64n` in clock cycle 7 and the upper address `ad[63..32]` and the upper command/byte enables `cbe[7..4]` are invalid.

Valid data is only presented on the `l_dat[31..0]` bus; however, because the PCI side is 32 bits wide and the local side is 64 bits wide, the `l_ldat_ackn` and `l_hdat_ackn` signals toggle to indicate whether the the low or high DWORD is being transferred on the local side. Along with these signals, valid data is qualified with `lm_ackn` asserted.



Because the local-side master interface is 64 bits and the PCI target is only 32 bits, these transactions always begin on 64-bit boundaries, which results in `l_ldat_ackn` always asserted first.



I/O and configuration read transactions by definition are 32 bits wide. The sequence of events is the same as in a 32-bit single-cycle memory read master transaction, as shown in [Figure 3–36](#). This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`.

Master Write Transactions

This section describes the behavior of the PCI MegaCore functions in the following types of master write transactions:

- Memory write
- I/O and configuration write

Memory Write Transactions

The PCI MegaCore functions support the following types of matched bus width and mismatched bus width memory write transactions in master mode:

- Burst memory write
- 32-bit single-cycle memory write
- 64-bit single-cycle memory write
- Mismatched bus width memory write



Mismatched bus-width transactions are 32-bit PCI transactions performed by the `pci_mt64` MegaCore function.

For each type of transaction, the following sequence of events is the same:

1. The local side asserts `lm_req32n` (and `lm_req64n` in the case of a 64-bit transaction) to request a transaction. Consequently, the PCI side asserts `reqn` to request mastership of the bus from the PCI arbiter.
2. When the PCI bus arbiter grants mastership by asserting the `gntn` signal, the local side asserts `lm_adr_ackn` to acknowledge the transaction's address and command. During the same clock cycle when `lm_adr_ackn` is asserted, the local side provides the address on the `l_adi` bus and the command on `l_cbeni[3..0]`. At the same time, the `pci_mt64` or `pci_mt32` function turns on the drivers for `framen` (and `req64n`, in the case of a 64-bit transaction.)

3. The PCI function begins the PCI address phase. During the PCI address phase, the local side must provide the byte enables for the transaction on the `l_cbeni` bus. For burst transactions, byte enables are used throughout the transaction. At the same time, the PCI side turns on the driver for `irdyn`.



You can change the byte enables for the successive data words in burst transactions by turning on **Allow Variable Byte Enable During Burst Transactions** option in the **Advanced PCI MegaCore Function Features** page of the **Parameterize - PCI Compiler** wizard. Refer to [“Allow Variable Byte Enables During Burst Transactions” on page 2–5](#) for more information about this option.

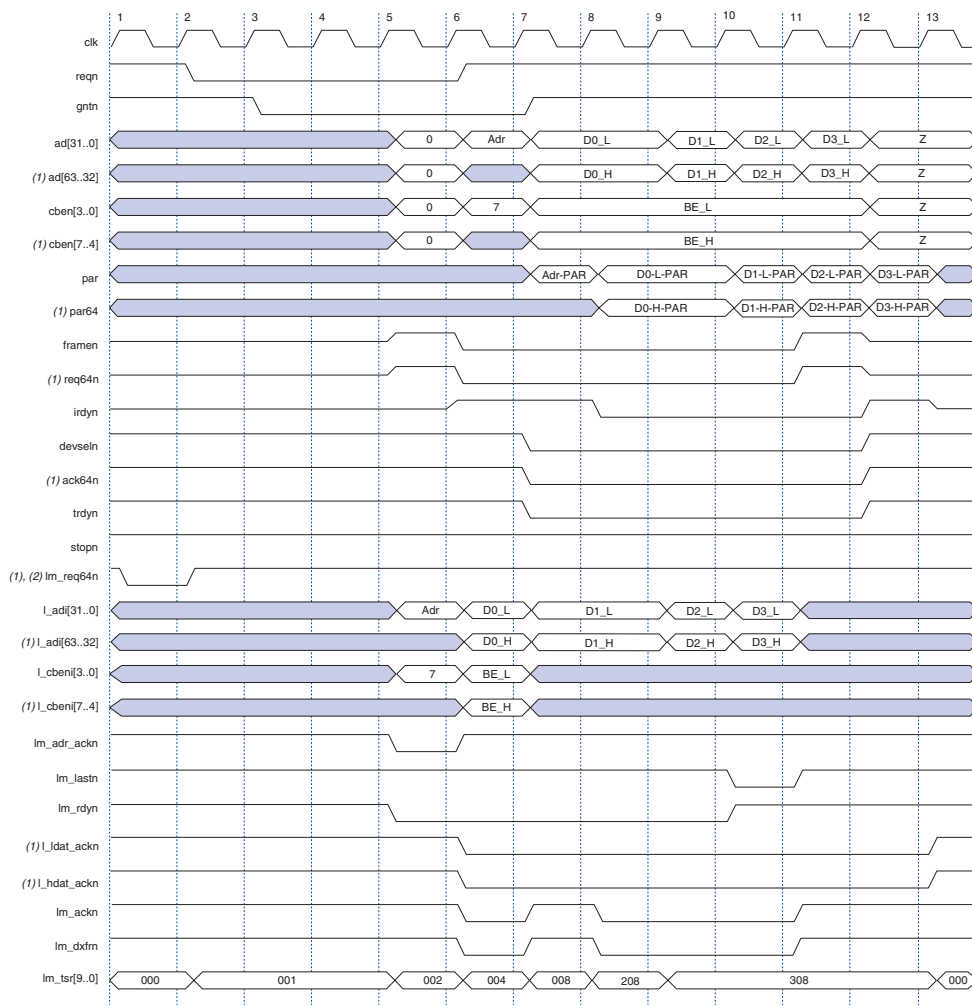
4. If the address of the transaction matches the memory range specified in the base address register (BAR) of a PCI target, the PCI target asserts `devseln` to claim the transaction. One or more data phases follow next, depending on the type of write transaction.

The `pci_mt64` and `pci_mt32` functions treat memory write and memory write and invalidate in the same way. Any additional requirements for the memory write and invalidate command must be implemented by the local-side design.

Burst Memory Write Master Transactions

Figure 3–38 shows the waveform for a 64-bit zero-wait state burst memory write master transaction. This figure applies to both `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`. In this transaction, four 64-bit QWORDS are transferred from the local side to the PCI side.

Figure 3–38. Zero-Wait State Burst Memory Write Master Transaction



Notes to Figure 3–38:

- (1) This signal is not applicable to the `pci_mt32` MegaCore function.
- (2) For `pci_mt32`, `lm_req32n` should be substituted for `lm_req64n` for 32-bit master transactions.

Table 3–39 shows the sequence of events for a 64-bit zero-wait state burst memory write master transaction. The 64-bit extension signals are not applicable to the `pci_mt32` function.

Table 3–39. Zero-Wait State Burst Memory Write Master Transaction (Part 1 of 3)

Clock Cycle	Event
1	The local side asserts <code>lm_req64n</code> to request a 64-bit transaction.
2	The function outputs <code>reqn</code> to the PCI bus arbiter to request bus ownership. At the same time, the function asserts <code>lm_tsr[0]</code> to indicate to the local side that the master is requesting control of the PCI bus.
3	The PCI bus arbiter asserts <code>gntn</code> to grant the PCI bus to the function. Although Figure 3–31 shows that the grant occurs immediately and the PCI bus is idle at the time <code>gntn</code> is asserted, this action may not occur immediately in a real transaction. Before the function proceeds, it waits for <code>gntn</code> to be asserted and the PCI bus to be idle. A PCI bus idle state occurs when both <code>framen</code> and <code>irdyn</code> are deasserted.
5	The function turns on its output drivers, getting ready to begin the address phase. The function also outputs <code>lm_adr_ackn</code> to indicate to the local side that it has acknowledged its request. During this same clock cycle, the local side should provide the PCI address on <code>l_adi[31..0]</code> and the PCI command on <code>l_cbeni[3..0]</code>. The local side master interface asserts <code>lm_rdyn</code> to indicate that it is ready to send data to the PCI side. The function does not assert <code>irdyn</code> regardless if the local side asserts <code>lm_rdyn</code> to indicate that it is ready to send data, only for the first data phase on the local side. For subsequent data phases, the PCI MegaCore function asserts <code>irdyn</code> if the local side is ready to send data. The PCI MegaCore function continues to assert its <code>reqn</code> signal until the end of the address phase. The function also asserts <code>lm_tsr[1]</code> to indicate to the local side that the PCI bus has been granted.

Table 3–39. Zero-Wait State Burst Memory Write Master Transaction (Part 2 of 3)

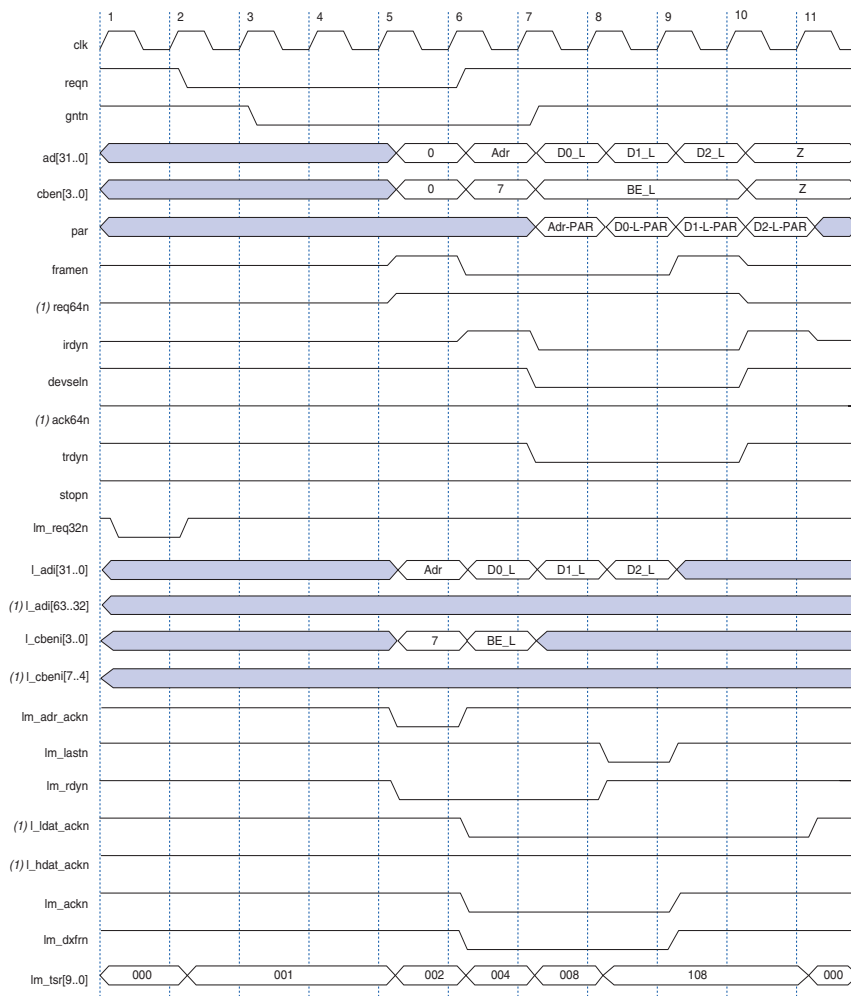
Clock Cycle	Event
6	The PCI MegaCore function begins the 64-bit memory write transaction with the address phase by asserting <code>framen</code> and <code>req64n</code>. At the same time, the local side must provide the byte enables for the transaction on the <code>l_cbeni</code> bus. You can change the byte enables for the successive data words in burst transactions by turning on Allow Variable Byte Enable During Burst Transactions option in the Advanced PCI MegaCore Function Features page of the Parameterize - PCI Compiler wizard. Refer to “Allow Variable Byte Enables During Burst Transactions” on page 2–5 for more information about this option. The PCI MegaCore function asserts <code>lm_ackn</code> to indicate to the local side that it is ready to transfer data. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the PCI MegaCore function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code> and <code>l_hdat_ackn</code> signals indicate to the local side that the PCI MegaCore function has transferred one data word from the <code>l_adi</code> bus. The function asserts <code>lm_tsr[2]</code> to indicate to the local side that the PCI bus is in its address phase. If the arbiter deasserts <code>gntn</code> in less than 3 clock cycles, the PCI MegaCore function does not assert <code>lm_tsr[2]</code> in this clock cycle. For recommendations of how to accommodate scenarios where the arbiter deasserts <code>gntn</code> in less than three clock cycles, refer to “Design Consideration” on page 3–92 for more information.
7	The target claims the transaction by asserting <code>devseln</code>. In this case, the target performs a fast address decode. The target also asserts <code>ack64n</code> to inform the function that it can transfer 64-bit data. The target also asserts <code>trdyn</code> to inform the function that it is ready to receive data. During this clock cycle, the function also asserts <code>lm_tsr[3]</code> to inform the local side that it is in data phase mode. The function deasserts <code>lm_ackn</code> because its internal pipeline has valid data from the local side data transfer during the previous clock cycle but no data was transferred on the PCI side. To ensure that the proper data is transferred on the PCI bus, the function asserts <code>irdyn</code> during the first data phase only after the PCI target asserts <code>devseln</code>.
8	The function asserts <code>irdyn</code> to inform the target that the function is ready to send data. Because the <code>irdyn</code> and <code>trdyn</code> are asserted, the first 64-bit data is transferred to the PCI side on the rising edge of clock cycle 9. The PCI MegaCore function asserts <code>lm_tsr[9]</code> to indicate to the local side that the target can transfer 64-bit data. The function also asserts <code>lm_ackn</code> to inform the local side that the PCI side is ready to accept data. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code>, <code>l_ldat_ackn</code>, and <code>l_hdat_ackn</code> signals indicates to the local side that it has transferred one data word from the <code>l_adi</code> bus.

Table 3–39. Zero-Wait State Burst Memory Write Master Transaction (Part 3 of 3)

Clock Cycle	Event
9	Because <code>irdyn</code> and <code>trdyn</code> are asserted, the second 64-bit data word is transferred to the PCI side on the rising edge of clock cycle 10. The function asserts <code>lm_ackn</code> to inform the local side that the PCI side is ready to accept data. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code>, <code>l_ldat_ackn</code>, and <code>l_hdat_ackn</code> signals indicates to the local side that it has transferred one data word from the <code>l_adi</code> bus. The function asserts <code>lm_tsr[8]</code> in the same clock cycle to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock cycle. The function also asserts <code>lm_tsr[9]</code> to inform the local side that the PCI target has claimed the 64-bit transaction with <code>ack64n</code>.
10	Because <code>irdyn</code> and <code>trdyn</code> are asserted, the third 64-bit data word is transferred to the PCI side on the rising edge of clock 11. The function asserts <code>lm_ackn</code> to inform the local side that the PCI side is ready to accept data. Because <code>lm_rdyn</code> was asserted in the previous cycle and <code>lm_ackn</code> is asserted in the current cycle, the function asserts <code>lm_dxfrn</code>. The assertion of the <code>lm_dxfrn</code>, <code>l_ldat_ackn</code>, and <code>l_hdat_ackn</code> signals indicates to the local side that it has transferred one data word from the <code>l_adi</code> bus. Also, the assertion of the <code>lm_lastn</code> signal indicates to the local side that valid data is expected on the <code>l_adi</code> bus. Also, the assertion of the <code>lm_lastn</code> signal indicates that clock cycle 10 is the last data phase on the local side. The function also asserts <code>lm_tsr[8]</code> in the same clock cycle to inform the local side that a successful data transfer has occurred on the PCI bus during the previous clock cycle.
11	Because <code>lm_lastn</code> was asserted and a data phase was completed in the previous cycle, the function deasserts <code>framen</code> and <code>req64n</code> and asserts <code>irdyn</code> to signal the last data phase. Because <code>trdyn</code> is asserted, the last data phase is completed on the PCI side on the rising edge of clock cycle 12. On the local side, the function deasserts <code>lm_ackn</code> and <code>lm_dxfrn</code> since the last data phase on the local side was completed on the previous cycle. The function continues to assert <code>lm_tsr[8]</code>, informing the local side that a successful data transfer has occurred on the PCI bus during the previous clock cycle.
12	The function deasserts <code>irdyn</code> and tri-states <code>framen</code> and <code>req64n</code>. The PCI target deasserts <code>devseln</code>, <code>ack64n</code>, and <code>trdyn</code>. These actions indicate that the transaction has ended and there will be no additional data phases. The function continues to assert <code>lm_tsr[8]</code>, informing the local side that a successful data transfer has occurred on the PCI bus during the previous clock cycle.
13	The function deasserts <code>lm_tsr[3]</code>, informing the local side that the data transfer mode is completed.

Figure 3–39 shows the same transaction as in Figure 3–42, but the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. This figure applies to both `pci_mt64` and `pci_mt32`, excluding the 64-bit extension signals as noted for `pci_mt32`. The `pci_mt64` function does not assert `req64n` on the PCI side. Therefore, the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid.

Figure 3–39. 32-Bit PCI & 32-Bit Local-Side Burst Memory Write Master Transaction

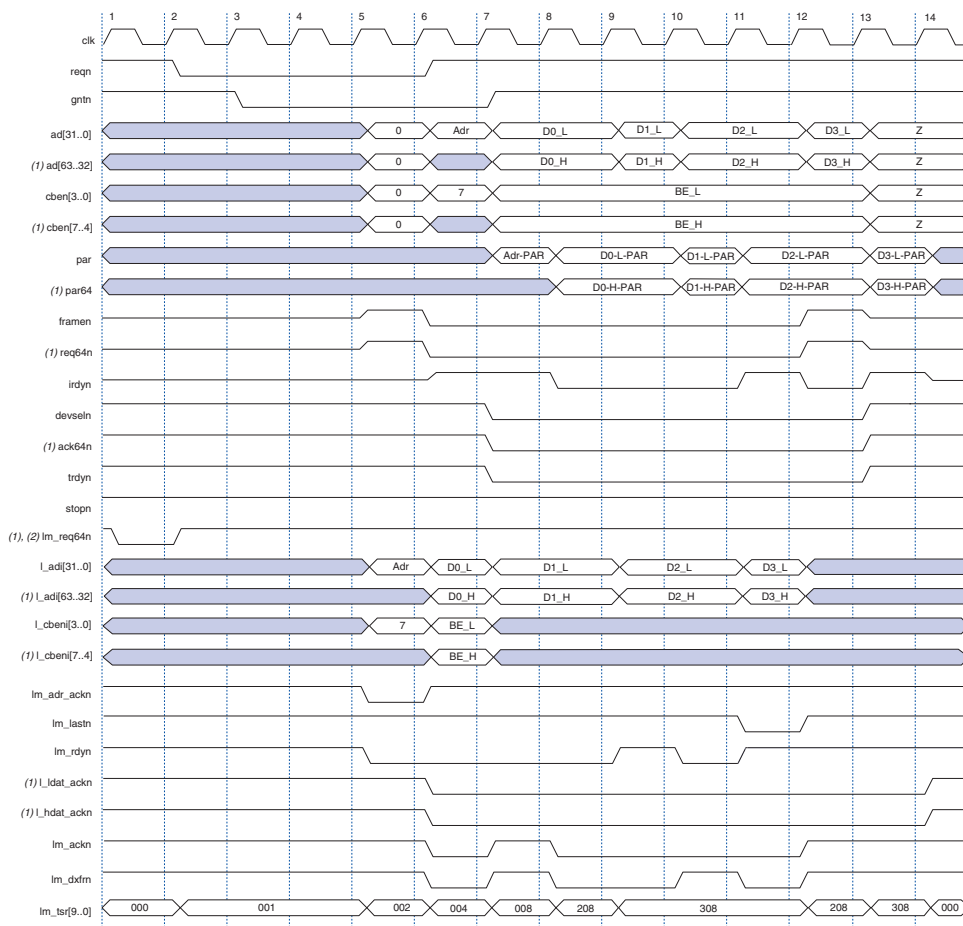


Note to Figure 3–39:

(1) This signal is not applicable to the `pci_mt32` MegaCore function.

Figure 3–40 shows the same transaction as in Figure 3–38 but with the local side inserting a wait state. This figure applies to both the `pci_mt64` and `pci_mt32` functions, except the 64-bit extension signals as noted for `pci_mt32`. The local side deasserts `lm_rdyn` in clock cycle 9. Consequently, on the following clock cycle (clock cycle 10), the `pci_mt64` and `pci_mt32` functions suspend data transfer on the local side by deasserting the `lm_dxfm` signal. Because there is no data transfer on the local side in clock cycle 10, the function suspends data transfer on the PCI side by deasserting the `irdyn` signal in clock cycle 11.

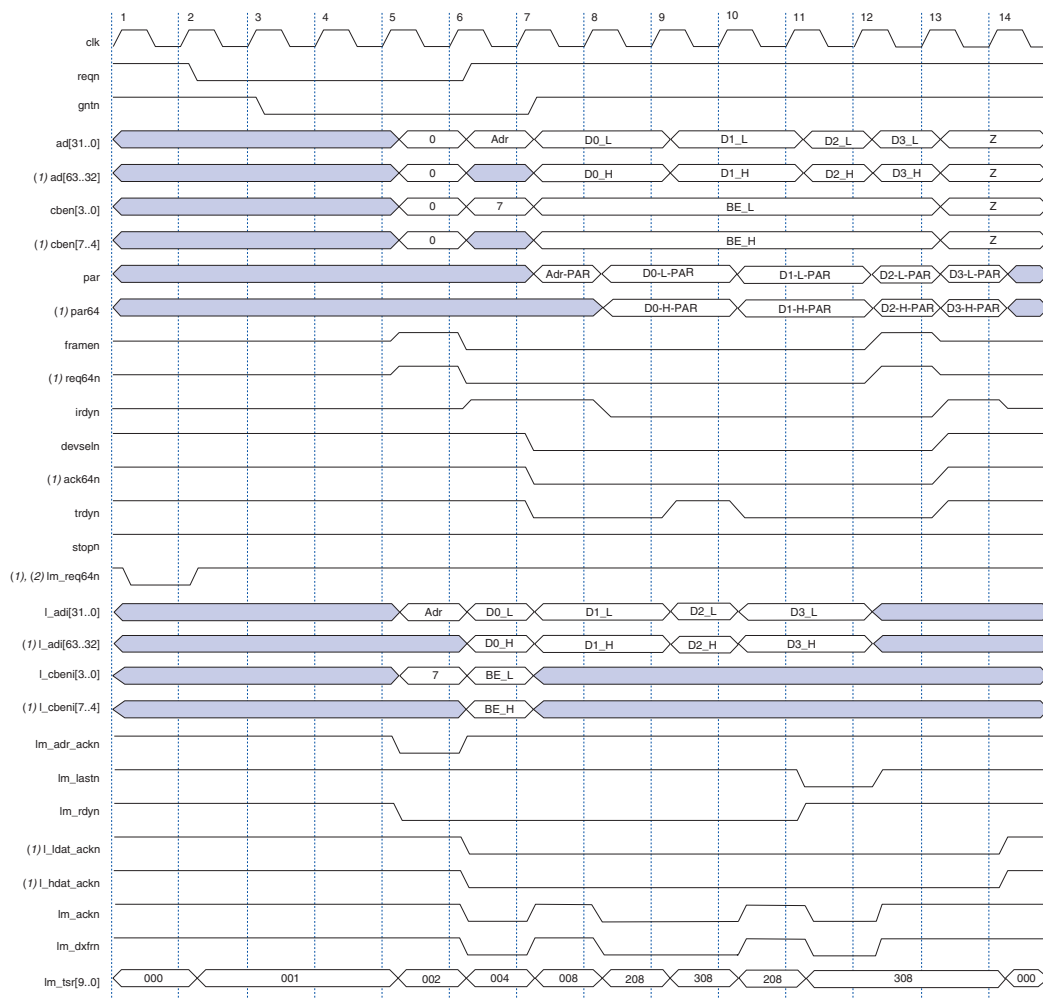
Figure 3–40. Burst Memory Write Master Transaction with Local Wait State



Notes to Figure 3–40:

- (1) This signal is not applicable to the `pci_mt32` MegaCore function.
- (2) For `pci_mt32`, `lm_req32n` should be substituted for `lm_req64n` for 32-bit master transactions.

Figure 3–41 shows the same transaction as in Figure 3–38 but with the PCI bus target inserting a wait state. This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`. The PCI target inserts a wait state by deasserting `trdyn` in clock cycle 9. Consequently, on the following clock cycle (clock cycle 10), the `pci_mt64` and `pci_mt32` functions deassert the `lm_ackn` and `lm_dxfrn` signals on the local side. Data transfer is suspended on the PCI side in clock cycle 9 and on the local side in clock cycle 10. Also, because `lm_lastn` is asserted and `lm_rdyn` is deasserted in clock cycle 11, the `lm_ackn` and `lm_dxfrn` signals remain deasserted after clock cycle 12.

Figure 3–41. Burst Memory Write Master Transaction with PCI Wait State**Notes to Figure 3–41:**

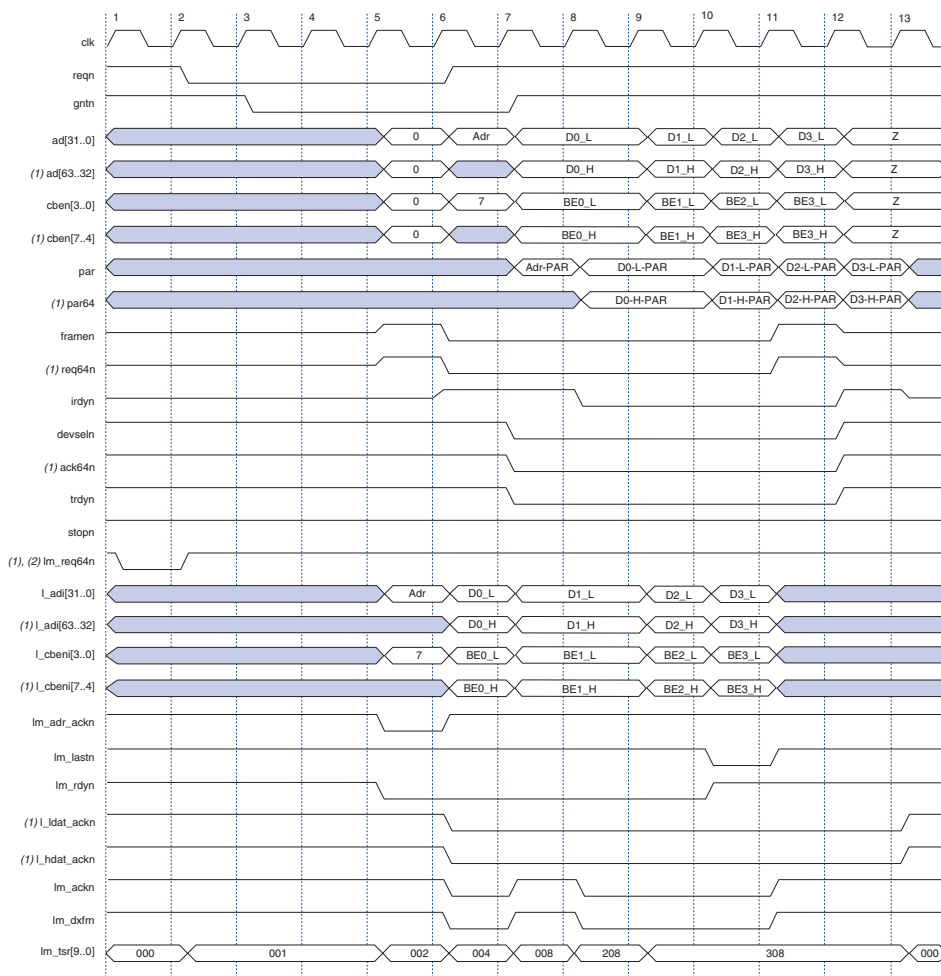
- (1) This signal is not applicable to the pci_mt32 MegaCore function.
- (2) For pci_mt32, lm_req32n should be substituted for lm_req64n for 32-bit master transactions.

Burst Memory Write Master Transactions with Variable Byte Enables

Figure 3–42 shows a burst memory write master transaction using variable byte enables. To allow this type of transaction, turn on **Allow Variable Byte Enables During Burst Transactions** on the Advanced PCI

MegaCore Function Features page of the **Parameterize - PCI Compiler** wizard. This option allows changing byte enables for the successive data words during a burst transaction.

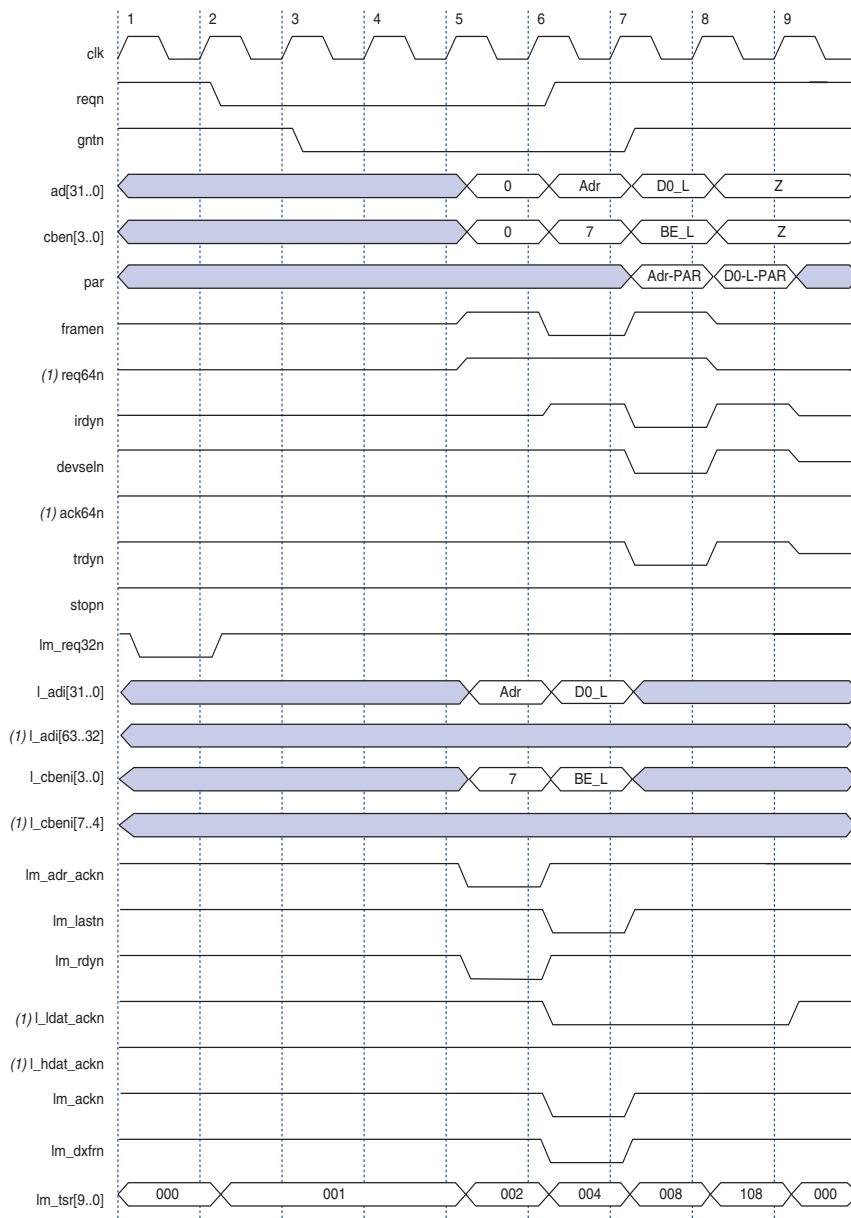
For example, in [Figure 3–38](#) the local side provides byte enables at the rising edge of clock cycle 6. This value of byte enables is used throughout the burst transaction. In [Figure 3–42](#) the same transaction is shown with **Allow Variable Byte Enables During Burst Transactions** turned on and the local side is providing unique byte enables for every data transfer.

Figure 3–42. Burst Memory Write Master Transaction with Variable Byte Enables**Notes to Figure 3–42:**

- (1) This signal is not applicable to the pci_mt32 MegaCore function.
- (2) For pci_mt32, lm_req32n should be substituted for lm_req64n for 32-bit master transactions.

32-Bit Single-Cycle Memory Write Master Transactions

Figure 3–43 shows a 32-bit single-cycle memory write master transaction. The transaction shown in Figure 3–43 is the same as that shown in Figure 3–39, except that the local side master interface transfers only one data word. This figure applies to both the pci_mt64 and pci_mt32 MegaCore functions, excluding the 64-bit extension signals as noted for pci_mt32.

Figure 3–43. 32-Bit PCI & 32-Bit Local-Side Single-Cycle Memory Write Transaction**Note to Figure 3–43:**

(1) This signal is not applicable to the pci_mt32 MegaCore function.

64-Bit Single Cycle Memory Write Master Transactions

This section is only applicable to the `pci_mt64` MegaCore function. The `pci_mt64` MegaCore function performs 64-bit single-cycle master write transactions if the **Assume ack64n Response** option is turned on. (This option is located on the **Advanced PCI MegaCore Function Features** page of the **Parameterize - PCI Compiler** wizard.) This option can be used if both of the following statements are true for your system:

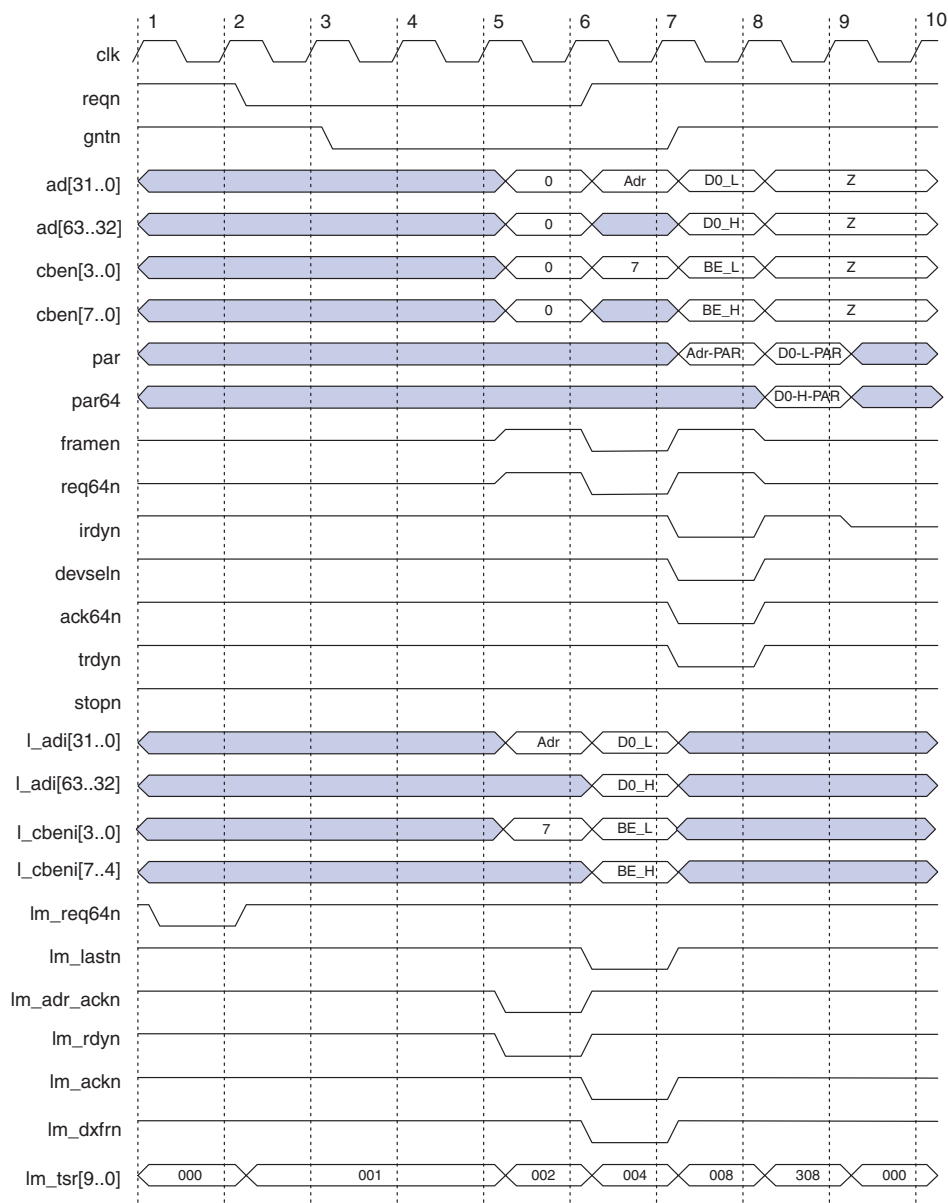
- The bit width of all devices is known, such as in an embedded system
- All 64-bit master transactions are claimed by 64-bit targets that respond with `ack64n` asserted

When you turn on the **Assume ack64n Response** option, the `pci_mt64` master can do the following:

- Perform 64-bit single-cycle master write transactions
- Initiate 64-bit master write transactions with less initial `irdyn` latency

During 64-bit master write transactions in standard operation mode, the `pci_mt64` function waits until the target asserts `devseln` before asserting `irdyn`. This action allows the master to ensure that the correct number of DWORDs are transferred if a 32-bit target claims the transaction.

Standard operation prevents the `pci_mt64` MegaCore function from supporting 64-bit single-cycle master memory write transactions. When the `pci_mt64` master initiates a single-cycle 64-bit write and the target is a 32-bit device, the upper 32-bits of data are not transferred across the PCI bus and are lost from the local side master application. If you turn on the **Assume ack64n Response** option, the `pci_mt64` MegaCore function can support 64-bit single-cycle master write transactions because the target is guaranteed to be a 64-bit device. [Figure 3–44](#) shows an example of a 64-bit single-cycle memory write master transaction where the `pci_mt64` function is the master.

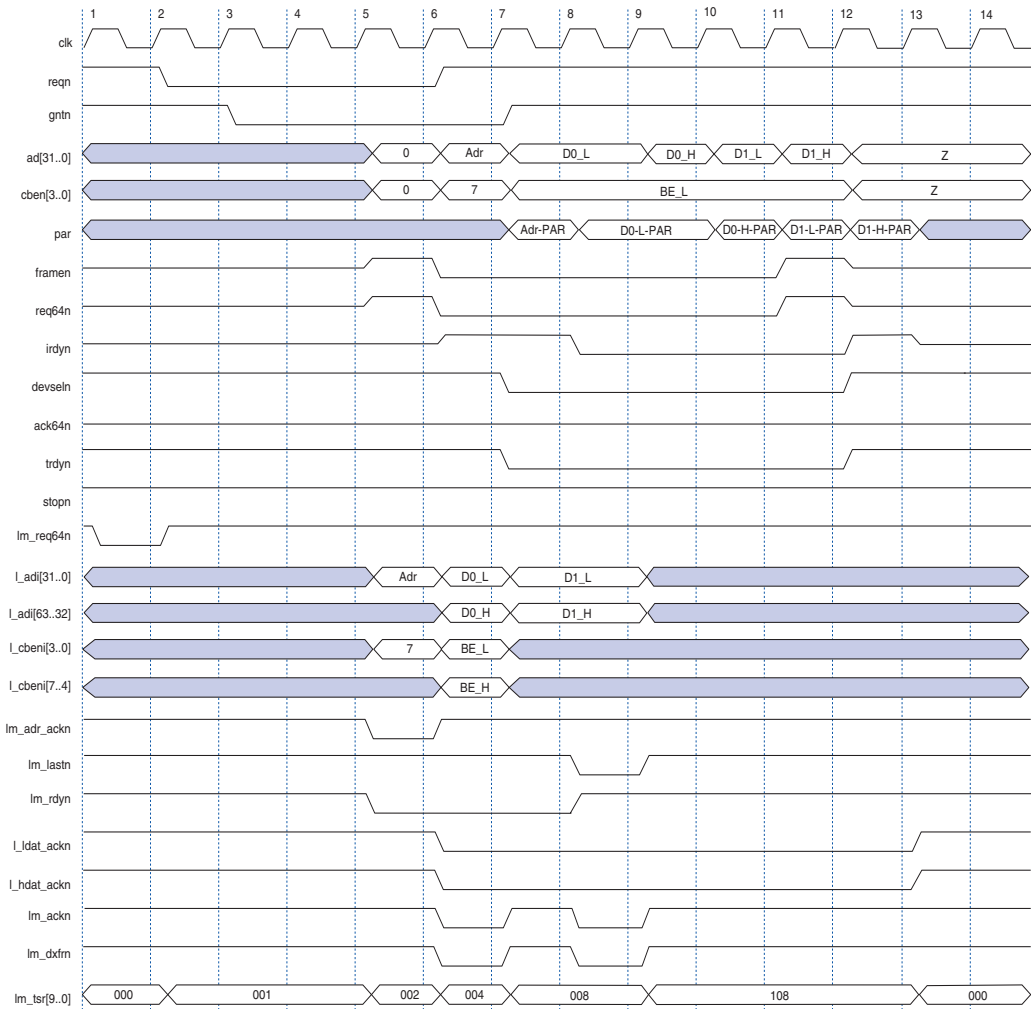
Figure 3–44. PCI 64-Bit Single-Cycle Master Memory Write Operation

Mismatched Bus Width Burst Memory Write Master Transactions

This section is only applicable to the `pci_mt64` MegaCore function. [Figure 3–45](#) shows the same transaction as in [Figure 3–38](#), but the PCI target cannot transfer 64-bit transactions. In this transaction, the local-side master interface requests a 64-bit transaction by asserting `lm_req64n`. The `pci_mt64` function asserts `req64n` on the PCI side. However, the PCI target cannot transfer 64-bit data, and therefore does not assert `ack64n` in clock cycle 7. Because this is the case, the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid.

In this case, the PCI function transfers 64 bits of data from the local side `l_adi[63..0]` bus and automatically transfers 32-bit data on the PCI side. The function automatically inserts wait states on the local side by deasserting the `lm_ackn` signal as necessary.

Also, because the PCI side is 32 bits wide and the local side is 64 bits wide, the `pci_mt64` function assumes that the transactions are within 64-bit boundaries. Therefore, the `pci_mt64` function registers `l_adi[63..0]` on the local side and transfers the lower 32-bit data word on `l_adi[31..0]` on the PCI side first, and the upper 32-bit data word on `l_adi[63..32]` afterwards.

Figure 3–45. 32-Bit PCI & 64-Bit Local-Side Master Burst Memory Write Transaction

I/O & Configuration Write Master Transactions

I/O and configuration write transactions by definition are 32 bits wide. The sequence of events is the same as in a 32-bit single-cycle memory write master transaction, as shown in [Figure 3–43](#). This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`.

Abnormal Master Transaction Termination

An abnormal transaction termination 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_mt64` and `pci_mt32` functions during the following abnormal termination conditions:

- Latency timer expires
- Target retry
- Target disconnect without data
- Target 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_mt64` and `pci_mt32` functions adhere to this rule, and when they end the transaction because the latency timer expired, they assert `lm_tsr[4]` (`tsr_lat_exp`) until the beginning of the next master transaction.



The PCI MegaCore functions allow the option of disabling the latency timer for embedded applications. Refer to “[Disable Master Latency Timer](#)” on page 2–6 for more information.

Retry

The target issues a retry by asserting `stopn` and `devseln` during the first data phase. When the `pci_mt64` and `pci_mt32` MegaCore functions detect a retry condition (refer to “[Retry](#)” on page 3–77 for details), they end the cycle and assert `lm_tsr[5]` until the beginning of the next transaction. This process informs the local-side device that it has ended the transaction because the target issued a retry.



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

Disconnect Without Data

The target device issues a disconnect without data if it is unable to transfer additional data during the transaction. The signal pattern for this termination is described in “[Disconnect](#)” on page 3–79. When the `pci_mt64` and `pci_mt32` functions end a transaction because of a disconnect without data, they assert `lm_tsr[6]` 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 3–79. When the `pci_mt64` and `pci_mt32` functions end a transaction because of a disconnect with data, they assert `lm_tsr[7]` 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 3–86. When the `pci_mt64` and `pci_mt32` functions end a transaction because of a target abort, they assert the `tabort_rcvd` signal, which is the same as the PCI status register bit 12. Therefore, the signal remains asserted until it is reset by the host.

Master Abort

The `pci_mt64` and `pci_mt32` functions terminate 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_mt64` and `pci_mt32` functions inform 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.

Host Bridge Operation

This section describes using the `pci_mt64` and `pci_mt32` MegaCore functions as a host bridge application in a PCI system. The `pci_mt64` and `pci_mt32` functions support the following advanced master features, which should be enabled when using the functions in a host bridge application:

- Use in host bridge application
- Allow internal arbitration logic

The host bridge features can be enabled through the **Advanced PCI MegaCore Function Features** page of the **Parameterize - PCI Compiler** wizard.

Using the PCI MegaCore Function as a Host Bridge

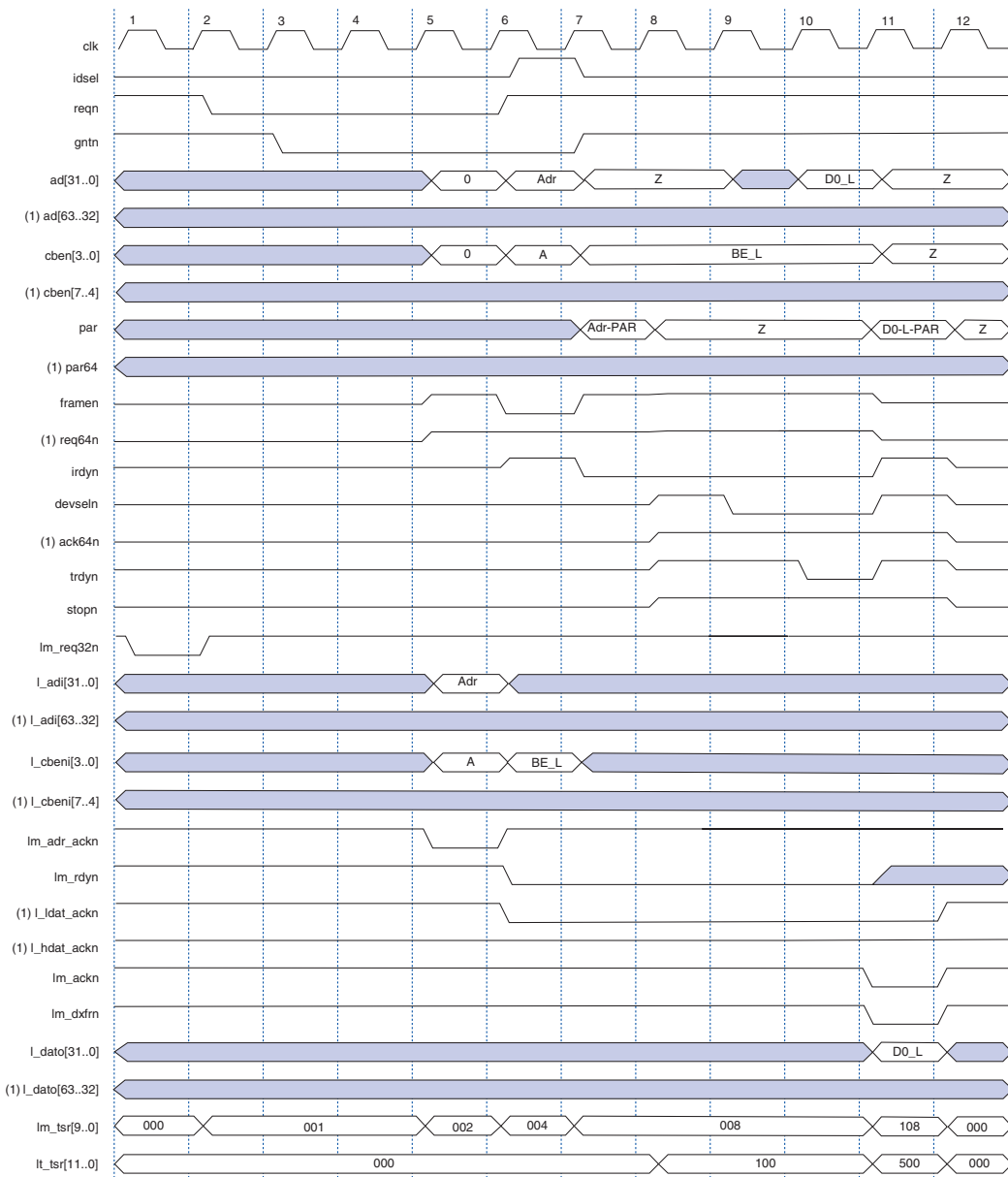
Turning on **Use in Host Bridge Application** hardwires the master enable bit of the command register (`bit [2]`) to a value of 1, which permanently enables the master functionality of the `pci_mt64` and `pci_mt32` MegaCore functions. Additionally, the **Use in Host Bridge Application** option also allows the `pci_mt64` or `pci_mt32` master device to generate configuration read and write transactions to the internal configuration space. With the **Use in Host Bridge Application** option, the same logic and software routines used to access the configuration space of other PCI devices on the bus can also configure the `pci_mt64` or `pci_mt32` configuration space.



To perform configuration transactions to internal configuration space, the `idsel` signal must be connected following the PCI specification requirements.

PCI Configuration Read Transaction from the `pci_mt64` Local Master Device to the Internal Configuration Space

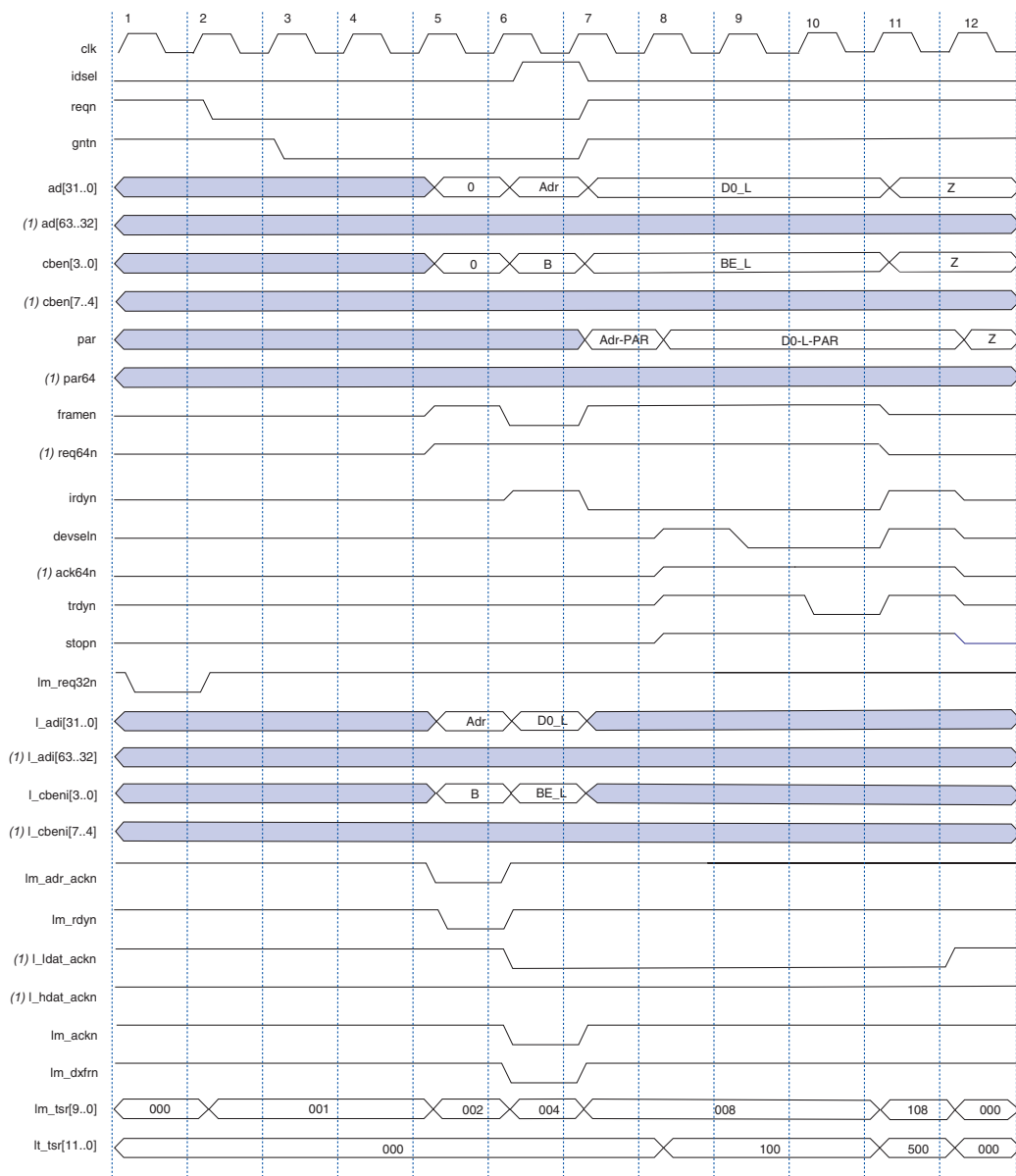
Figure 3–46 shows the behavior of the `pci_mt64` master device performing a configuration read transaction from internal configuration space. The local master requests a 32-bit transaction by asserting the `lm_req32n` signal. When requesting a configuration read transaction, the `pci_mt64` function will automatically perform a single-cycle transaction. The local master signals are asserted as if the `pci_mt64` master is completing a single-cycle, 32-bit memory read transaction, similar to Figure 3–44 in the Master Mode Operation section. The `pci_mt64` function's internal configuration space will respond to the transaction without affecting the local side signals. Figure 3–46 applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for the `pci_mt32` function.

Figure 3–46. Configuration Read from Internal Configuration Space in Self-Configuration Mode**Note to Figure 3–46:**

(1) This signal is not applicable to the pci_mt32 MegaCore function.

PCI Configuration Write Transaction from the pci_mt64 Local Master Device to the Internal Configuration Space

Figure 3–47 shows the behavior of the pci_mt64 master performing a configuration write transaction to internal configuration space. The local master requests a 32-bit transaction by asserting the `lm_req32n` signal. When requesting a configuration write transaction, the pci_mt64 function will automatically perform a single-cycle transaction. The local master signals are asserted as if the pci_mt64 master is completing a single-cycle 32-bit memory write transaction, similar to Figure 3–43 in the Master Mode Operation section. The pci_mt64 function's internal configuration space will respond to the transaction without affecting the local side signals. Figure 3–47 applies to both the pci_mt64 and pci_mt32 MegaCore functions, excluding the 64-bit extension signals as noted for pci_mt32.

Figure 3–47. Configuration Write to Internal Configuration Space in Self-Configuration Mode**Note to Figure 3–47:**(1) This signal is not applicable to `pci_mt32` for 32-bit master write transactions.

64-Bit Addressing, Dual Address Cycle (DAC)

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



This section applies to the `pci_mt64` and `pci_t64` MegaCore functions only.

Target Mode Operation

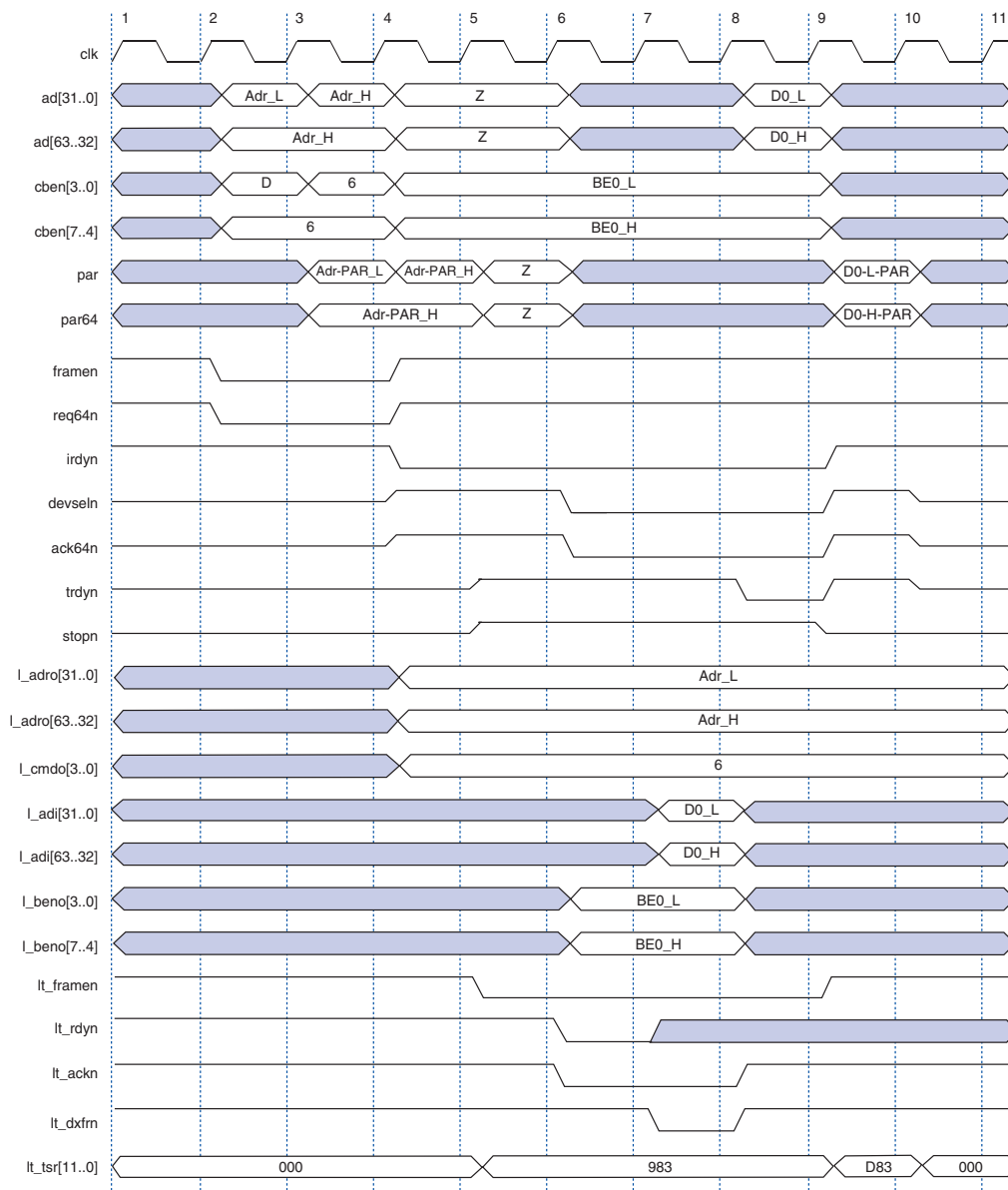
A read or write transaction begins after a master acquires mastership of the PCI bus and asserts `framen` to indicate the beginning of a bus transaction. If the transaction is a 64-bit transaction, the master device asserts the `req64n` signal at the same time it asserts the `framen` signal. The `pci_mt64` and `pci_t64` functions assert the `framen` signal in the first clock cycle, which is called the first address phase. During the first address phase, the master device drives the 64-bit transaction address on `ad[63..0]`, the DAC command on `cben[3..0]`, and the transaction command on `cben[7..4]`. On the following clock cycle, during the second address phase, the master device drives the upper 32-bit transaction address on both `ad[63..32]` and `ad[31..0]`, and the transaction command on both `cben[7..4]` and `cben[3..0]`. During these two address phases, the PCI MegaCore function latches the transaction address and command, and decodes the address. If the transaction address matches the `pci_mt64` and `pci_t64` target, the `pci_mt64` and `pci_t64` target asserts the `devseln` signal to claim the transaction. In 64-bit transactions, `pci_mt64` and `pci_t64` also assert the `ack64n` signal at the same time as the `devseln` signal indicating that `pci_mt64` and `pci_t64` accept the 64-bit transaction. The `pci_mt64` and `pci_t64` functions implement slow decode, i.e., the `devseln` and `ack64n` signals are asserted after the second address phase is presented on the PCI bus. Also, both of the `lt_tsr[1..0]` signals are driven high to indicate that the BAR0 and BAR1 address range matches the current transaction address.

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

Figure 3–48 shows the waveform for a 64-bit address, 64-bit data single-cycle target read transaction. Figure 3–48 is exactly the same as Figure 3–7, except that Figure 3–48 has two address phases (described in the previous paragraph). Also, both `lt_tsr[1..0]` signals are asserted to indicate that the BAR0 and BAR1 address range of `pci_mt64` and `pci_t64` matches the current transaction address. In addition, the current transaction upper 32-bit address is latched on `l_adro[63..32]`, and the lower 32-bit address is latched on `l_adro[31..0]`.



All 32-bit addressing transactions described in “[Target Mode Operation](#)” on page 3–131 are applicable for 64-bit addressing transactions, except for the differences described in the previous paragraph.

Figure 3–48. 64-Bit Address, 64-Bit Data Single-Cycle Target Read Transaction

Master Mode Operation

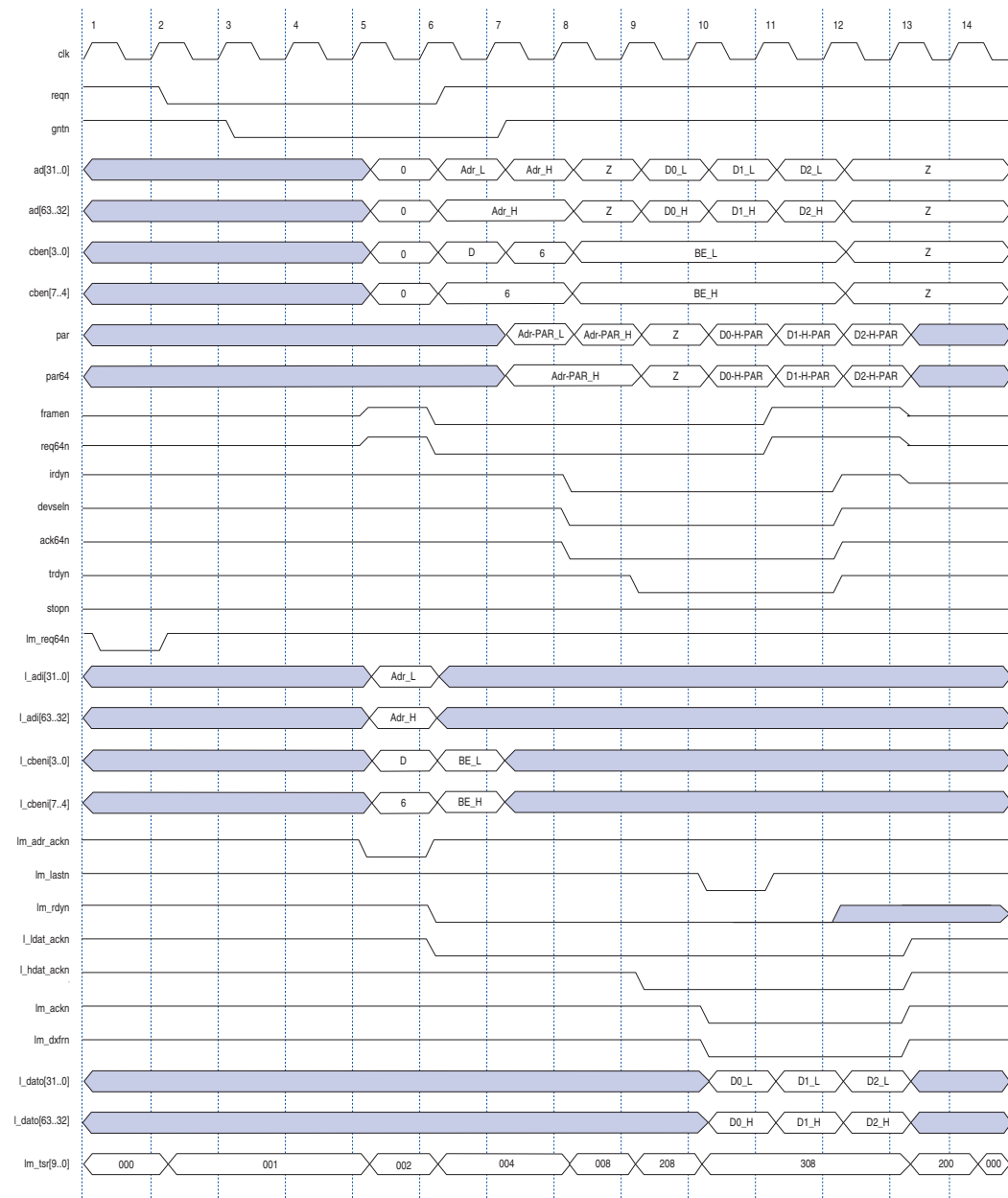
A master operation begins when the local-side master interface asserts the `lm_req64n` signal to request a 64-bit transaction or the `lm_req32n` signal to request a 32-bit transaction. The `pci_mt64` function outputs the `reqn` signal to the PCI bus arbiter to request bus ownership. The `pci_mt64` function also outputs the `lm_adr_ackn` signal to the local side to acknowledge the request. When the `lm_adr_ackn` signal is asserted, the local side provides the PCI address on the `l_adl[63..0]` bus, the DAC command on `l_cbenl[3..0]`, and the transaction command on `l_cbenl[7..4]`. When the PCI bus arbiter grants the bus to the `pci_mt64` function by asserting `gntn`, `pci_mt64` begins the transaction with a dual address phase. The `pci_mt64` function asserts the `framen` signal in the first clock cycle, which is called the first address phase. During the first address phase, the `pci_mt64` function drives the 64-bit transaction address on `ad[63..0]`, the dual address cycle command on `cben[3..0]`, and the transaction command on `cben[7..4]`. On the following clock cycle, during the second address phase, the `pci_mt64` function drives the upper 32-bit transaction address on both `ad[63..32]` and `ad[31..0]`, and the transaction command on both `cben[7..4]` and `cben[3..0]`.

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

Figure 3–49 shows the waveform for a 64-bit address, 64-bit data master burst memory read transaction. Figure 3–49 is exactly the same as Figure 3–31, except that Figure 3–49 has two address phases (as described in the previous paragraph).



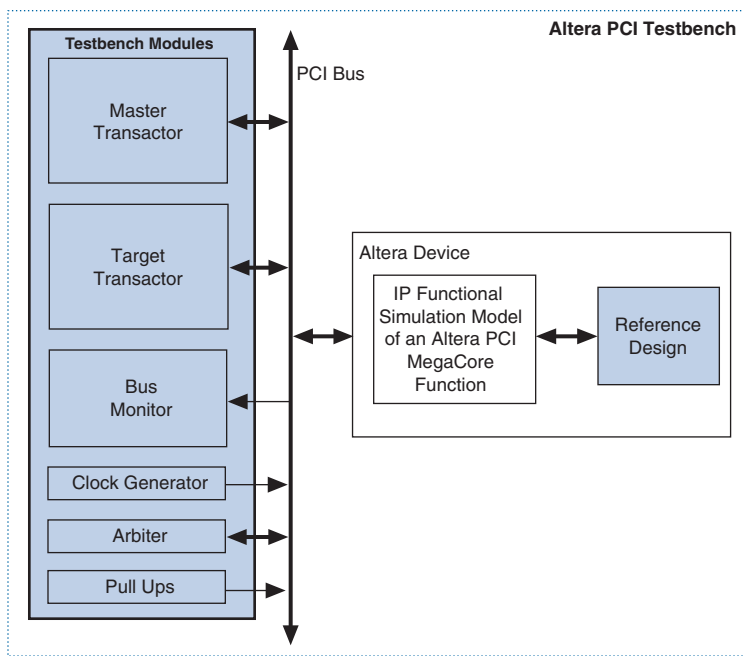
All 32-bit addressing transactions described in “Master Mode Operation” on page 3–134 are applicable for 64-bit addressing transactions, except for the differences described in the previous paragraph.

Figure 3–49. 64-Bit Address, 64-Bit Data Master Burst Memory Read Transaction

General Description

The Altera PCI testbench facilitates the design and verification of systems that implement the Altera `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore functions. You can build a PCI behavioral simulation environment by using the components of the PCI testbench, an IP functional simulation model of any Altera PCI MegaCore function, and your VHDL or Verilog HDL application design. Figure 4–1 shows the block diagram of the PCI testbench. The shaded blocks are provided with the PCI testbench.

Figure 4–1. Altera PCI Testbench Block Diagram



To use the PCI testbench, you should have a basic understanding of PCI bus architecture and operations. This document describes the features and applications of the PCI testbench to help you successfully design and verify your application design.

Features

The PCI testbench includes the following features:

- Easy to use simulation environment for any standard VHDL or Verilog HDL simulator
- Open source VHDL and Verilog HDL files
- Flexible PCI bus functional model to verify your application that uses any Altera PCI MegaCore function
- Simulates all basic PCI transactions including memory read/write operations, I/O read and write transactions, and configuration read and write transactions
- Simulates all abnormal PCI transaction terminations including target retry, target disconnect, target abort, and master abort
- Simulates PCI bus parking
- Includes a simple reference design that performs basic memory and I/O transactions

PCI Testbench Files

The Altera PCI testbench is included and installed with the PCI Compiler. [Figure 4-2](#) shows the directory structure of PCI testbench subdirectory, where *<path>* is the directory in which the PCI Compiler is installed.

Figure 4-2. PCI Testbench Directory Structure

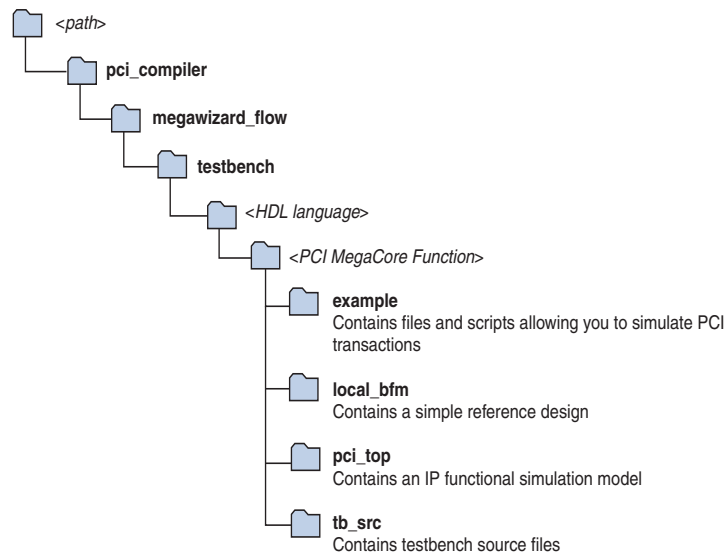


Table 4–1 gives a description of the PCI testbench source files provided in the **tb_src** directory. For more information on these files, refer to “Testbench Specifications” on page 4–6.

Table 4–1. Files Contained in the tb_src Directory	
File(1)	Description
mstr_tranx	The master transactor defines the procedures (VHDL) or tasks (Verilog HDL) needed to initiate PCI transactions in the testbench.
mstr_pkg	The master package consists descriptions of procedures (VHDL) or tasks (Verilog HDL) of the master transactor (mstr_tranx) commands.
trgt_tranx	The target transactor simulates the target behavior in the testbench. It serves to respond to PCI transactions.
trgt_tranx_mem_init.dat	This file is the memory initialization file for the target transactor.
monitor	This module monitors the PCI transactions on the bus and reports the results.
clk_gen	This module generates 33-or 66-MHz clock for the PCI agents.
arbiter	This module contains PCI bus arbiter.
pull_up	This module is used to provide weak pull-up on the tri-stated signals.
altera_tb	This is a sample top-level file that instantiates the testbench modules and the IP functional simulation model of the PCI MegaCore function. You can use this sample top-level file in your application design by replacing the <code>top_local</code> instance from the testbench file with the top level of your application design. Refer to “Simulation Flow” on page 4–20 for more information.

Note to Table 4–1:

(1) All files are provided in both VHDL and Verilog HDL.

Table 4–2 describes the reference design files provided in the **local_bfm** directory. For more information on these files refer to “Local Reference Design” on page 4–15.

Table 4–2. Files Contained in the local_bfm Directory	
File (1)	Description
dma (2)	This module serves as a direct memory access (DMA) engine for the reference design
lm_last_gen (2)	This module generates a <code>lm_lastn</code> signal for <code>local_master</code>
local_master (2)	This module initiates master transactions from the local side. The DMA engine triggers the state machine inside <code>local_master</code> .
local_target	The module consists of a simple target state machine that performs 32 or 64-bit memory read/write transactions with the LPM memory and 32-bit single-cycle IO read/write transactions with an I/O register defined in the local target
prefetch	This module is used to prefetch the data from a memory block during burst target read transactions and burst master write transactions.
lpm_ram_32	This module is used to instantiate 1 KByte of RAM. This RAM is accessible by both <code>local_target</code> and <code>local_master</code> .
local_top	This module instantiates all the local reference design modules.

Notes to Table 4–2:

- (1) All files are provided in both VHDL and Verilog HDL.
- (2) Not applicable to the `pci_t32` and `pci_t64` MegaCore functions.

The **example** directory (Table 4–3) contains the following subgroups of files:

- Testbench files modified to simulate the reference design provided in **local_bfm**
- A top-level design file
- Simulation scripts

Refer to “Simulation Flow” on page 4–20 for more information on the modified testbench files.

Table 4–3. Files Contained in the example Directory

File	Description
Modified Testbench Files	
mstr_tranx (1)	This module contains the master transactor code. The INITIALIZATION section has the parameters set to simulate the Local Reference Design. The USER COMMANDS section has the PCI commands that will be executed during simulation.
trgt_tranx (1)	This module contains the target transactor code. The address_lines and mem_hit_range settings are set to simulate the reference design.
trgt_tranx_mem_init.dat (1)	This file is the memory initialization file for the target transactor.
Top-level Design File	
altera_tb (1)	This top-level file instantiates the testbench module files, the IP functional simulation model of the pci_mt64 MegaCore function, and the reference design file. The idsel signal of the Altera PCI MegaCore function is connected to address bit 28 and the idsel signal of the target transactor is connected to address bit 29.
Simulation Scripts	
run_altera_modelsim.tcl	This script can be used with the Altera-ModelSim simulator. This script compiles all the files provided in <code><path>\pci_compiler\megawizard_flow\testbench\<HDL language>\<PCI MegaCore function>\example</code> and simulates the reference design for the transactions specified in the mstr_tranx file.
run_modelsim.tcl	This script can be used with the ModelSim SE, PE or AE simulators. This script compiles all the files provided in <code><path>\pci_compiler\megawizard_flow\testbench\<HDL language>\<PCI MegaCore function>\example</code> and simulates the reference design for the transactions specified in the mstr_tranx file.
run_vcs.sh	This script is used with VCS simulator. This script compiles the files provided in <code><path>\pci_compiler\megawizard_flow\testbench\<HDL language>\<PCI MegaCore function>\example\<function></code> and simulates the reference design for the transactions specified in the mstr_tranx file.
run_ncverilog.sh	This script must be used with NC-Verilog simulator. This script will compile all the files provided in <code><path>\pci_compiler\megawizard_flow\testbench\<HDL language>\<PCI MegaCore function>\example</code> and simulates the reference design for the transactions specified in the mstr_tranx file.

Note to Table 4–3:

(1) This file is provided in both VHDL and Verilog HDL.

Testbench Specifications

This section describes the modules used by the PCI testbench including master commands, setting and controlling target termination responses, bus parking, and PCI bus speed settings. Refer to [Figure 4-1](#) for a block diagram of the PCI testbench. The Altera PCI testbench has the following modules:

- Master transactor (mstr_tranx)
- Target transactor (trgt_tranx)
- Bus monitor (monitor)
- Clock generator (clk_gen)
- Arbiter (arbiter)
- Pull ups (pull_ups)
- A local reference design

The PCI testbench consists of VHDL and Verilog HDL. If your application requires a feature that is not supported by the PCI testbench, you can modify the source code to add the feature. You can also modify the existing behavior to fit your application needs.

[Table 4-4](#) shows the PCI bus transactions supported by the PCI testbench.

Table 4-4. PCI Testbench PCI Bus Transaction Support				
Transactions	Master Transactor	Target Transactor	Local Master	Local Target
Interrupt acknowledge cycle				
I/O read	✓	✓	✓	✓
I/O write	✓	✓	✓	✓
Memory read	✓	✓	✓	✓
Memory write	✓	✓	✓	✓
Configuration read	✓	✓		
Configuration write	✓	✓		
Memory read multiple		✓		
Memory write multiple				
Dual address cycle				
Memory read line		✓		
Memory write and invalidate		✓		

Table 4-5 shows the testbench's target termination support. The master transactor and the local master respond to the target terminations by terminating the transaction gracefully and releasing the PCI bus.

Table 4-5. PCI Testbench Target Termination Support

Features	Master Transactor	Target Transactor	Local Master	Local Target
Target abort	✓		✓	
Target retry	✓	✓	✓	✓
Target disconnect	✓	✓	✓	✓

Master Transactor (mstr_tranx)

The master transactor simulates the master behavior on the PCI bus. It serves as an initiator of PCI transactions for Altera PCI testbench. The master transactor has three main sections:

- PROCEDURES (VHDL) or TASKS (Verilog HDL)
- INITIALIZATION
- USER COMMANDS

PROCEDURES and TASKS Sections

The PROCEDURES (VHDL) and TASKS (Verilog HDL) sections define the events that are executed for the user commands supported by the master transactor. The events written in the PROCEDURES and TASKS sections follow the phases of a standard PCI transaction as defined by the *PCI Local Bus Specification, Revision 3.0*, including:

- Address phase
- Turn-around phase (read transactions)
- Data phases
- Turn-around phase

The master transactor terminates the PCI transactions in the following cases:

- The PCI transaction has successfully transferred all the intended data
- The PCI target terminates the transaction prematurely with a target retry, disconnect, or abort as defined in the *PCI Local Bus Specification, Revision 3.0*
- A target does not claim the transaction resulting in a master abort

The bus monitor informs the master transactor of a successful data transaction or if the target has terminated the transaction. Refer to the source code to see how the master transactor uses these termination signals from the bus monitor.

The PCI testbench master transactor's PROCEDURES and TASKS sections implement basic PCI transaction functionality. If your application requires different functionality, modify the events to change the behavior of the master transactor. Additionally, you can create new procedures or tasks in the master transactor using the existing events as an example.

INITIALIZATION Section

This user-defined section defines the parameters and reset length of your PCI bus on power-up. Specifically, the system should reset the bus and write the configuration space of the PCI agents. You can modify the master transactor INITIALIZATION section to match your system requirements by changing the time the system reset is asserted and modifying the data written in the configuration space of the PCI agents.

USER COMMANDS Section

The master transactor USER COMMANDS section contains the commands that initiate the PCI transactions you want to run for your tests. The list of events that are executed by these commands is defined in the PROCEDURES and TASKS sections. Customize the USER COMMANDS section to execute the sequence of commands as needed to test your design.

Table 4–6 shows the commands the master transactor supports.

Table 4–6. Supported Master Transactor Commands	
Command Name	Action
cfg_rd	Performs a configuration read
cfg_wr	Performs a configuration write
mem_wr_32	Performs a 32-bit memory write
mem_rd_32	Performs a 32-bit memory read
mem_wr_64	Performs a 64-bit memory write
mem_rd_64	Performs a 64-bit memory read
io_rd	Performs an I/O read
io_wr	Performs an I/O write

cfg_rd

The `cfg_rd` command performs single-cycle PCI configuration read transactions with the address provided in the command argument.

Syntax:	<code>cfg_rd(address)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.

cfg_wr

The `cfg_wr` command performs single-cycle PCI configuration write transactions with the address, data, and byte enable provided in the command arguments.

Syntax:	<code>cfg_wr(address, data, byte_enable)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Transaction data. The data must be in hexadecimal radix.
	<code>byte_enable</code>	Transaction byte enable. The byte enable value must be in hexadecimal radix

mem_wr_32

The `mem_wr_32` command performs a memory write with the address and data provided in the command arguments. This command can perform a single-cycle or burst 32-bit memory write depending on the number of DWORDs provided in the command argument.

- The `mem_wr_32` command performs a single-cycle 32-bit memory write if the DWORD value is 1.

- The `mem_wr_32` command performs a burst-cycle 32-bit memory write if the `DWORD` value is greater than 1. In a burst transaction, the first data phase uses the data value provided in the command. The subsequent data phases use values incremented sequentially by 1 from the data provided in the command argument.

Syntax:	<code>mem_wr_32(address, data, dword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Data used for the first data phase. Subsequent data phases use a value incremented sequentially by 1. This value must be in hexadecimal radix.
	<code>dword</code>	The number of <code>DWORDS</code> written during the transaction. A value of 1 indicates a single-cycle memory write transaction. A value greater than one indicates a burst transaction. This value must be an integer.

mem_rd_32

The `mem_rd_32` command performs a memory read with the address provided in the command argument. This command can perform single-cycle or burst 32-bit memory read depending on the value of the `dword` argument.

- If the `dword` value is 1, the command performs a single-cycle transaction.
- If the `dword` value is greater than 1, the command performs a burst transaction.

Syntax:	<code>mem_rd_32(address, dword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>dword</code>	The number <code>DWORDS</code> read during the transaction. A value of one indicates a single-cycle memory read transaction. A value greater than one indicates a burst transaction. This value must be an integer.

mem_wr_64

The `mem_wr_64` command performs a memory write of the data to the address provided in the command. This command can perform single-cycle or burst 64-bit memory write depending on the value of the `qword` argument.

- This command performs a single-cycle 64-bit memory write if the `qword` value is one.
- This command performs a burst-cycle 64-bit memory write if the `qword` value is greater than one. In a burst transaction, the first data phase uses the data value provided in the command. The subsequent data phases use values incremented sequentially by one from the data provided in the command argument.

Syntax:	<code>mem_wr_64(address, data, qword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Data used for first data phase. Subsequent data phases use a value sequentially incremented by one from this data. This value must be in hexadecimal radix.
	<code>qword</code>	The number <code>QWORDS</code> written in a transaction. A value of one indicates a single-cycle memory write transaction. A value greater than one indicates a burst transaction. This value must be an integer.

mem_rd_64

The `mem_rd_64` command performs memory read transactions with the address provided in the command argument. This command can perform single-cycle or burst 64-bit memory read depending on the value of the `qword` argument.

- If the `qword` value is one the command performs a single-cycle transaction.
- If the `qword` value is greater than one the command performs a burst transaction.

Syntax:	<code>mem_rd_32(address, qword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>qword</code>	The number <code>QWORDS</code> read in the transaction. A one indicates a single-cycle memory read transaction. A value greater than one indicates a burst transaction. This value must be an integer.

io_wr

The `io_wr` command performs a single-cycle memory write transaction with the address and data provided in the command arguments.

Syntax:	<code>io_wr(address, data)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Data written during the transaction. This value must be in hexadecimal radix.

io_rd

The `io_rd` command performs single-cycle I/O read transactions with the address provided in the command argument.

Syntax:	<code>io_rd(address)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.

Target Transactor (trgt_tranx)

The target transactor simulates the behavior of a target agent on the PCI bus. The master transactions initiated by the Altera PCI MegaCore function under test should be addressed to the target transactor. The target transactor operates in 32- or 64-bit mode. The target transactor implements two base address registers BAR0 and BAR1 as shown in [Table 4–7](#).

Table 4–7. Target Transactor Address Space Allocation

Configuration Register	Address Space Type	Block Size	Address Offset
BAR0	Memory Mapped	1 KByte	000-3FF
BAR1	I/O Mapped	16 Bytes	0-F

For definitions of the target transactor address space, refer to the base address registers in [Table 4–8, “Memory Map,”](#) on page 16.

The memory range reserved by BAR0 is defined by the `address_lines` and `mem_hit_range` settings in the target transactor source code.

The target transactor has a 32-bit register that stores data for I/O transactions. This register is mapped to BAR1 of the configuration address space. Because this is the only register that is mapped to BAR1,

any address that is within the BAR1 range results in an `io_hit` action. Refer to the target transactor source code to see how the address is decoded for `io_hit`.



The target transactor ignores byte enables for all memory, I/O, and configuration transactions.

The target transactor `idsel` signal should be connected to one of the PCI address bits in the top-level file of the PCI testbench for configuration transactions to occur on BAR0 and BAR1.

To model different target terminations, use the following three input signals:

- `trgt_tranx_retry`—The target transactor retries the memory transaction if `trgt_tranx_retry` is set to one
- `trgt_tranx_discA`—The target transactor terminates the memory transaction with data if `trgt_tranx_discA` is set to one
- `trgt_tranx_discB`—The target transactor terminates the memory transaction with a disconnect without data if `trgt_tranx_discB` is set to one

The target transactor has two main sections:

- FILE IO
- PROCEDURES (VHDL) and TASKS (Verilog HDL)

FILE IO section

Upon reset, this section initializes the target transactor memory array with the contents of the `trgt_tranx_mem_init.dat` file, which must be in the project's working directory. Each line in the `trgt_tranx_mem_init.dat` file corresponds to a memory location, the first line corresponding to offset "000". The number of lines defined by the `address_lines` parameter in the target transactor source code should be equal to number of lines in the `trgt_tranx_mem_init.dat` file. If the number of lines in `trgt_tranx_mem_init.dat` file is less than the number of lines defined by the `address_lines` parameter, the remaining lines in the memory array are initialized to 0.

PROCEDURES and TASKS sections

The PROCEDURES section (VHDL) and the corresponding TASKS section (Verilog HDL) define the events to be executed for the decoded PCI transaction. These sections are fully documented in the source code. You can modify the procedures or tasks to introduce different variations

in the PCI transactions as required by your application. You can also create new procedures or tasks that are not currently implemented in the target transactor by using the existing procedures or tasks as an example.

Bus Monitor (monitor)

The bus monitor displays PCI transactions and information messages to the simulator's console window and in the **log.txt** file when an event occurs on the PCI bus. The bus monitor also sends the PCI transaction status to the master transactor. The bus monitor reports the following messages:

- Target retry
- Target abort
- Target terminated with disconnect-A (target terminated with data)
- Target terminated with disconnect-B (target terminated without data)
- Master abort
- Target not responding

The bus monitor reports the target termination messages depending on the state of the `trdyn`, `devseln`, and `stopn` signals during a transaction. The bus monitor reports a master abort if `devseln` is not asserted within four clock cycles from the start of a PCI transaction. It reports that the target is not responding if `trdyn` is not asserted within 16 clock cycles from the start of the PCI transaction. You can modify the bus monitor to include additional PCI protocol checks as needed by your application.

Clock Generator (clk_gen)

The clock generator, or `clk_gen`, module generates the PCI clock for the Altera PCI testbench. This module generates a 66-Mhz clock if the `pciclk_66Mhz_enable` parameter is set to true in the PCI testbench top-level file, otherwise, it generates a 33-Mhz clock. The default value of `pciclk_66Mhz_enable` is true.

Arbiter (arbiter)

This module simulates the PCI bus arbiter. The module is a two-port arbiter in which the device connected to port 0 of the arbiter has a higher priority than the device connected to port 1. For example, if device 0 requests the PCI bus while device 1 is performing a PCI transaction, the arbiter removes the grant from device 1 and gives it to device 0. This module allows you to simulate bus parking on devices connected to port 0 by setting the `Park` parameter to true. You can change the value of this parameter in the Altera PCI testbench top-level file.

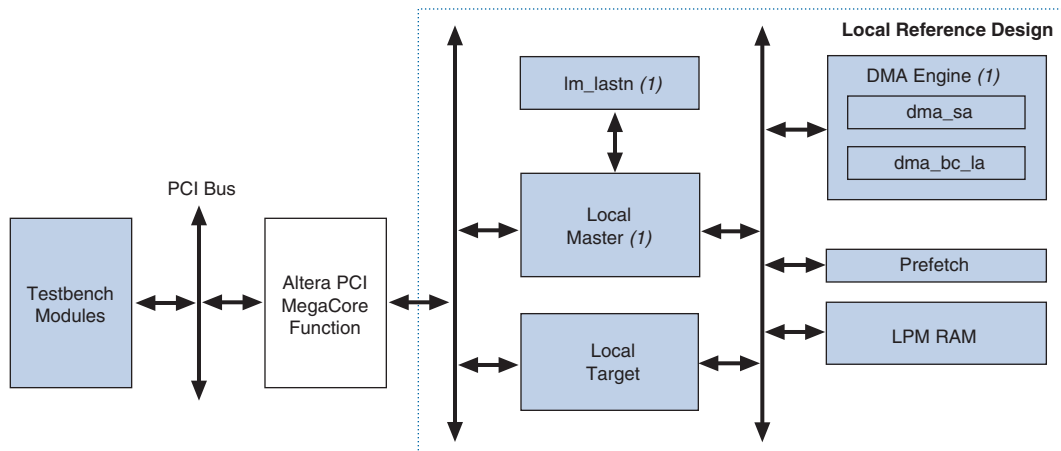
Pull Up (pull_up)

This module simulates the pull up functionality on the PCI signals. The `ad`, `cbe`, `fram`, `irdyn`, `trdyn`, `stopn`, `devseln`, `perrn`, and `serrn` signals of the PCI bus are pulled with a weak high value. This action is necessary to ensure that these signals are never floating or unknown during your simulation.

Local Reference Design

The reference design can be used to quickly evaluate Altera PCI MegaCore functions. This design performs memory read and write transactions with an LPM_RAM library of parameterized modules (LPM) function instantiated in the back end. It also performs I/O read and write transactions with an I/O register that is instantiated locally in the local master and target blocks. You can replace the reference design with your application design to verify PCI transactions with other PCI agents.

Figure 4-3 shows the block diagram of the local reference design. The shaded blocks are provided with the PCI testbench.

Figure 4–3. Local Reference Design**Note to Figure 4–3:**

- (1) The DMA Engine, `lm_lastn` and local master blocks are not applicable for the `pci_t32` and `pci_t64` local reference designs.

Table 4–8 shows the memory map of the Altera PCI MegaCore function required to use the local reference design.

Table 4–8. Memory Map				
Memory Region	Mapping	Block size	Address Offset	Description
BAR0	Memory Mapped	1 KByte	000-3FF	Maps the LPM_RAM function.
BAR1	I/O Mapped	16 Bytes	0-F	Maps the I/O register.
BAR2	Memory Mapped	1 KByte	000-3FF	Maps the <code>trg_termination</code> register and DMA engine registers. Only the lower 24 Bytes of the address space are used.

The reference design has the following elements:

- Local target
- DMA engine
- Local master
- `lm_lastn` generator
- Prefetch
- LPM RAM

Local Target

The local target consists of a simple state machine that performs 32- or 64-bit memory read/write transactions with the LPM memory and 32-bit single-cycle I/O read/write transactions with an I/O register defined in the local target. The local target uses prefetch logic for burst read transactions and ignores byte enables for all memory and I/O transactions. Table 4–9 shows the BAR2 register mapping.

Table 4–9. BAR2 Register Mapping			
Address Space	Range Reserved	Mnemonic	Register Name
BAR2	00h-03h	targ_termination_reg	Target termination register.
BAR2	04h-07h	dma_sa[31:0]	DMA system address register
BAR2	08h-0Bh	dma_bc_la[31:0]	DMA byte count and local address register

Depending on the value of the target termination register, the local target performs the terminations in Table 4–10.

Table 4–10. Target Terminations	
targ_termination_reg Setting	Target Termination
xxxxxxx0	Normal Termination
xxxxxxx1	Target Retry
xxxxxxx2	Disconnect

DMA Engine

The DMA engine has two 32-bit registers, which are mapped to BAR2 in the PCI MegaCore function. Table 4–9 describes the mapping of DMA registers on BAR2.

To initiate a master transaction from a PCI MegaCore function, use the master transactor to perform memory writes to these locations with the appropriate values.

The dma_sa register defines the system address used for the PCI transaction. This address is driven during the address phase of the PCI transaction. Normally, the address written here is the base address register value of the PCI testbench target transactor.

The `dma_bc_1a` register location includes the following 3 fields:

- *Local address*—starting address at which the transaction begins reading or writing data during a PCI transaction.
- *Byte count*—number of DWORDs transferred during a PCI transaction
- *Transaction control*—type of transaction to be initiated



The byte count field is only used for memory transactions. For I/O transactions this value is ignored. Because I/O transactions are always 1 DWORD long, the data is transferred to the 32-bit I/O register.

Figure 4–4 shows the mapping of the `dma_bc_1a` fields.

Figure 4–4. `dma_bc_1a` Fields

Bit 31.....28	27.....16	15.....8	7.....0
Transaction Control	Reserved	Byte Count	Local Address

Table 4–11 shows definition of the transaction control field. Although 8 bits are reserved for the transaction control field, only 4 bits are used:

Table 4–11. Transaction Control Field	
Transaction Control (Binary)	Initiated Transaction Type
0000	32-Bit Memory Read
0010	64-Bit Memory Read
0100	32-Bit I/O Read
0110	32-Bit I/O Write
1000	32-Bit Memory Write
1010	64-Bit Memory Write

To initiate PCI transactions with Altera PCI MegaCore as a master, you must perform a 32-bit single-cycle write transaction to the `dma_sa` register followed by a 32-bit single-cycle write transaction to the `dma_bc_1a` register. The write to the `dma_bc_1a` register triggers the master control unit, which requests a master transaction and executes it as indicated in the transaction control part of the `dma_bc_1a` register.

Local Master

The DMA engine triggers the local master. The local master can perform 32- and 64-bit memory read/write transactions with the LPM RAM block and 32-bit single-cycle I/O read/write transactions with an I/O register defined in the local master. The local master uses prefetch logic for burst memory write transactions and uses the `last_gen` block to generate the `lm_lastn` signal for the `pci_mt64` and `pci_mt32` MegaCore functions. The local master ignores byte enables for all memory and I/O transactions.

Refer to the master transactor (`mstr_tranx`) source code in the `<path>\pci_compiler\megawizard_flow\testbench\<HDL language>\<PCI MegaCore function>\example` directory for examples of PCI transactions using the Altera PCI MegaCore function as a master.

lm_lastn Generator

This module generates the `lm_lastn` signal for the local master when the reference design is in master mode. The `lm_lastn` signal is a local-side master interface control signal that is used to request the end of the current transaction.

Refer to “[Master Local-Side Signals](#)” on page 3–24 for more information on the `lm_lastn` signal.

Prefetch

This module is used to prefetch the data from the LPM RAM block during burst target read transactions and burst master write transactions.

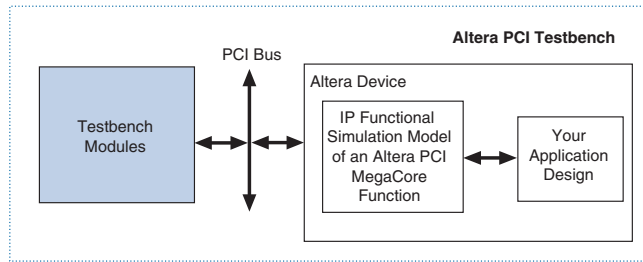
LPM RAM

This module is used to instantiate 1 KByte of RAM for the reference design. LPM RAM is accessible by both local target and local master. The Local target and local master ignore byte enables for all memory transactions.

Simulation Flow

This section describes the simulation flow using Altera PCI testbench. [Figure 4–5](#) shows the block diagram of a typical verification environment using the PCI testbench.

Figure 4–5. Typical Verification Environment Using the PCI Testbench



The simulation flow using Altera PCI testbench comprises the following steps.

1. Use IP Toolbench to specify the PCI MegaCore function configuration space parameters and generate an IP functional simulation model of your custom PCI MegaCore function.
2. Set the initialization parameters, which are defined in the master transactor model source code. These parameters control the address space reserved by the target transactor model and other PCI agents on the PCI bus.
3. The master transactor defines the procedures (VHDL) or tasks (Verilog HDL) needed to initiate PCI transactions in your testbench. Add the commands that correspond to the transactions you want to implement in your tests to the master transactor model source code. At a minimum, you must add configuration commands to set the BAR for the target transactor model and write the configuration space of the PCI MegaCore function. Additionally, you can add commands to initiate memory or I/O transactions to the PCI MegaCore function.

Refer to [Table 4–6 on page 4–8](#) for more information about the user commands.

4. Modify the target transactor model memory range. The target transactor reserves a 1-KByte memory array by default. On reset, this memory array is initialized by the `trgt_tranx_mem_init.dat` file. Refer to [“FILE IO section” on page 4–13](#) for more information about this file.

You can modify the memory instantiated by the target transactor model by changing the `address_lines` value and the `mem_hit_range` value to correspond to the value specified by `address_lines`. For example, if `address_lines` is 1024, the target transactor instantiates a 1-KByte memory array that corresponds to a memory hit range of 000-3FF Hex. Refer to [“Target Transactor \(trgt_tranx\)” on page 4–12](#) for more information.

5. Simulate the testbench for the desired time period.



Section II. PCI Compiler With SOPC Builder Flow

The PCI Compiler with SOPC Builder flow option allows you to build a complete PCI system using an automatically-generated interconnect. The SOPC Builder flow uses the on-chip system interconnect fabric as a bridge between various resident and external peripherals, which dramatically reduces design time.

This section includes the following chapters:

- [Chapter 5, Getting Started](#)
- [Chapter 6, Parameter Settings](#)
- [Chapter 7, Functional Description](#)
- [Chapter 8, Testbench](#)

Design Flow

To create a PCI system that uses the PCI Compiler with SOPC Builder, and to evaluate it using the OpenCore Plus hardware evaluation feature, include the following steps in your design flow:

1. Obtain and install the PCI Compiler.
2. Create a Quartus II project.
3. Use SOPC Builder and the Quartus II software to generate a system that uses the PCI-Avalon bridge.

Use MegaWizard to configure the PCI-Avalon bridge.

4. Use IP functional models to verify your system operation. Although this step is always recommended, it is more critical if you are using your own custom-defined SOPC Builder peripheral.
5. Use an Altera-provided PCI constraint file to ensure your system meets the timing requirements of the PCI specification.



For more information on obtaining and using Altera-provided PCI constraint files in your design, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

6. Use the Quartus II software to compile your design and perform static timing analysis.



You can generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware.

7. Purchase a license for the PCI MegaCore function.

After you have purchased a license for the PCI MegaCore function, the design flow requires the following additional steps:

1. Set up licensing.
2. Generate a programming file for the Altera device(s) on your board.
3. Program the Altera device(s) with the completed design.
4. Complete system verification.

Altera MegaCore functions, including PCI Compiler, can be installed from the MegaCore IP Library CD-ROM either during or after installing the Quartus II software. Alternatively, PCI Compiler can be downloaded individually from the Altera website and installed separately.



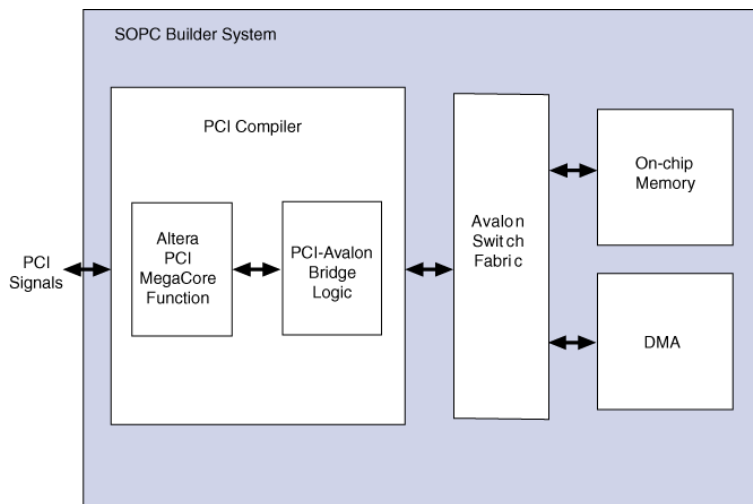
Altera recommends that you occasionally check the Altera website at www.altera.com for updates to the PCI Compiler.

PCI Compiler with SOPC Builder Flow Design Walkthrough

This walkthrough explains how to use SOPC Builder and the Quartus II software to generate a system containing the following components:

- PCI Compiler MegaCore function
- On-chip memory
- DMA controller

Figure 5–1 on page 5–3 shows how SOPC Builder integrates these components using the system interconnect fabric.

Figure 5–1. System Generated Using SOPC Builder

This walkthrough uses Verilog HDL to create a system. You can substitute VHDL for Verilog HDL.

This walkthrough consists of these steps:

- Create a New Quartus II Project
- Set Up the PCI-Avalon Bridge
- Add the Remaining Components to the SOPC Builder System
- Complete the Connections in SOPC Builder
- Generate the SOPC Builder System

Create a New Quartus II Project

You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity. To create a new project follow these steps:

1. To run the Quartus II software, select **Programs > Altera > Quartus II <version>** from the Windows Start menu. You can also use the Quartus II Web Edition software.
2. On the File menu, click **New Project Wizard**.
3. Click **Next** in the **New Project Wizard: Introduction** (the introduction does not display if you turned it off previously).

4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:
 - a. Type `c:\sopc_pci` in the **What is the working directory for this project?** box.
 - b. Type `chip_top` in the **What is the name of the project?** box.



The Quartus II software automatically specifies a top-level design entity that has the same name as the project. Do not change it.

5. Click **Next** to display the **New Project Wizard: Add Files** page.



When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.

6. If you installed the MegaCore IP library in a different directory from where you installed the Quartus II software, add user libraries by following these steps on the **New Project Wizard: Add Files** page:
 - a. Click **User Libraries**.
 - b. Type `<path>\pci_compiler\lib\` into the **Library name** box, where `<path>` is the directory in which you installed the PCI Compiler.
 - c. Click **Add** to add the path to the Quartus II project.
 - d. Click **OK** to save the library path in the project.

7. Click **Next** to display the **New Project Wizard: Family & Device Settings** page.

8. In the **New Project Wizard: Family & Device Settings** page, specify the following target device family and options:

- a. In the **Family** list, select **Stratix II**.



This walkthrough creates a design targeting the Stratix II device family. You can also use these procedures for other supported device families.

- b. Under **Target device**, select **Specific device selected in 'Available devices' list**.

- c. Under **Show in 'Available device' list**, all fields should have the default value of **Any**.
 - d. In the **Available devices:** list, select **EP2S90F1020C3**.
9. Click **Next** to display the **New Project Wizard: EDA Tool Settings** page. You don't need to turn on any settings on this page.
 10. Click **Next** to display the **New Project Wizard: Summary** page.
 11. Check the Summary page to ensure that you have entered all the information correctly.
 12. Click **Finish** to complete the Quartus II project.

Set Up the PCI-Avalon Bridge

To set up a PCI-Avalon bridge in the PCI Compiler, follow these steps:

1. To start SOPC Builder, select **SOPC Builder** from the Tools menu.

The **Altera SOPC Builder** window appears.



For more information on using SOPC Builder, refer to Quartus II Help.

2. In the **Create New System** dialog box, specify the following and click **OK** to continue.
 - a. Type `chip_top` in the **System Name** box
 - b. Under **Target HDL**, ensure **Verilog** is selected.



In this example, you are choosing the SOPC Builder-generated system file to be the same as the project's top level file. This is not required for your own design.

3. Build your system by adding components from the **System Contents** tab.

Expand **Interface Protocols**, followed by **PCI** and select **PCI Compiler**.

4. Click **Add**. The PCI Compiler Parameters Setting is displayed.
5. In the **Systems Options - 1** tab, use the default values.

6. Click **Next** to display the **Systems Options - 2** tab.
7. In the **System Options - 2** tab, specify the following :
 - a. Under **PCI Clock/Reset Settings**, select **Independent PCI and Avalon Clocks**.
 - b. Use the default values for the rest.
8. Click **Next** to display the **PCI Configuration** tab.
9. In the **PCI Configuration** tab, do the following:
 - a. Specify the following for **PCI Base Address Registers (BARs)**:
 - **BAR 0**: Use the default values.
 - **BAR 1**: In the **BAR Type** list, select **32-Bit Non-Prefetchable Memory**. The values for the rest are automatically displayed.
 - b. Use the default values for all **PCI Read-Only Registers**.
10. Click **Next** to display the **Avalon Configuration** tab.
11. In the **Avalon Configuration** tab, specify the following:
 - a. Under **Fixed Address Translation Table Contents**, double-click the value of **PCI Base Address** and type `0x30000000`
 - b. Under **Avalon CRA Port**, turn on **Control Register Access (CRA) Avalon Slave Port**.
 - c. Use the default values for the rest.
12. Click **Finish**. The PCI Compiler is added to your SOPC Builder system.



Normally the PCI base address is configured according to your system requirements. In this example, the value chosen is the same as the one that is used to configure the `trgt_tranx` module in the PCI testbench.



Your system is not yet complete, so you can ignore any error messages generated by SOPC Builder at this stage.

Add the Remaining Components to the SOPC Builder System

You will add the On-chip memory and DMA controller to your design.

1. Expand **Memories and Memory Controllers**, followed by **On-Chip**, and select **On-Chip Memory (RAM or ROM)**. This component contains a slave port.
2. In the **On-chip Memory** dialog box, specify the following:
 - a. Under **Memory Type**, select **MRAM** from the **Block type** list.
 - b. Turn off **Initialize memory content**.
 - c. Under **Size**, specify the **Total memory size** as **128 Kbytes**.
 - d. Use the default values for the rest.
3. Click **Finish**. The On-chip Memory is added to your SOPC Builder system.
4. Expand **Memories and Memory Controllers**, followed by **DMA** , and select **DMA Controller**. This component contains read and write master ports and a control port slave.
5. In the **DMA Controller** dialog box, specify the following in the **DMA Parameters** tab.
 - a. Under **Transfer size**, type 10 in the **Width of the DMA length register (1-32)** box .
 - b. Under **Burst transactions**, turn on **Enable burst transfers**.
 - c. Use the default values for the rest.
6. Click **Finish**. The DMA Controller module is added to your SOPC Builder system.

Complete the Connections in SOPC Builder

You need to connect the ports of the components in your SOPC system.

1. Move the pointer over the **Connections** column. Potential connections are displayed.
2. Connect the following master-slave ports:

<i>Table 5–1. Master-Slave Port Connections</i>	
Master Port	Slave Port
pci_compiler bar0_Prefetchable	onchip_mem s1
pci_compiler bar1_Prefetchable	pci_compiler Control_Register_Access
pci_compiler bar1_Prefetchable	dma control_port_slave
dma read_master	onchip_mem s1
dma read_master	pci_compiler PCI_Bus_Access
dma write_master	onchip_mem s1
dma write_master	pci_compiler PCI_Bus_Access

Figure 5–2 shows the complete connections.

Figure 5–2. Master-Slave Port Connections

Use	Connections	Module Name	Description	Clock	Base	End	...
☒		☒ pci_compiler bar0_Prefetchable bar1_Non_Prefetchable bar1_Prefetchable Control_Register_Acc... PCI_Bus_Access	PCI Compiler Avalon Master Avalon Master Avalon Slave Avalon Slave Avalon Slave	clk			
☒		☒ onchip_mem s1	On-Chip Memory (RAM or ROM) Avalon Slave	clk			
☒		☒ dma control_port_slave read_master write_master	DMA Controller Avalon Slave Avalon Master Avalon Master	clk			
					IRQ 0 0xffffffff 0xffffffff	0x00003ffe 0x000ffffe 0x0000001e	IRQ 0

3. Under **Clock Settings**, double-click the default speed (MHz) for **clk_0** and type 100.

4. From the System menu, select **Auto-Assign Base Addresses**.

SOPC Builder will generate informational messages indicating the PCI MegaCore function you have instantiated and the actual PCI BAR setting.

Generate the SOPC Builder System

You will now generate your SOPC system.

1. In the **SOPC Builder** window, click **Next** to display the **System Generation** tab.
1. In the **System Generation** tab, turn on **Simulation. Create simulator project files**.
2. Click **Generate**. Messages about the PCI MegaCore function that was instantiated and the progress of the system generation are displayed in the SOPC Builder message window.



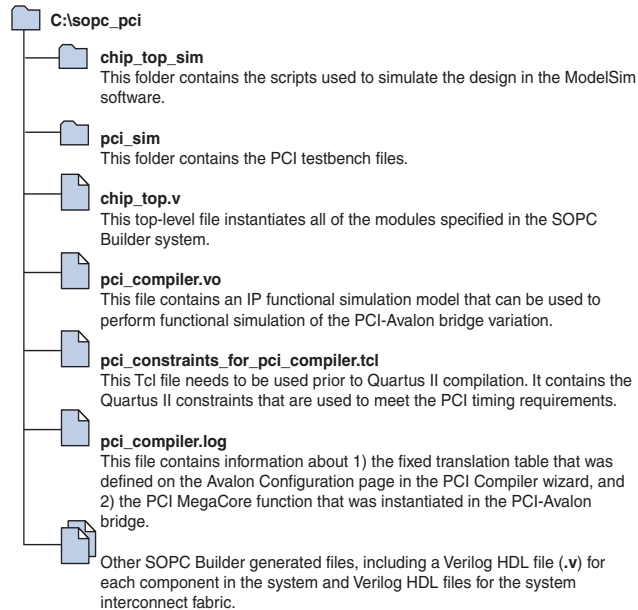
Among the files generated by SOPC Builder is the Quartus II IP File (**.qip**). This file contains information about a generated IP core or system. In most cases, the **.qip** file contains all of the necessary assignments and information required to process the MegaCore function or system in the Quartus II compiler. Generally, a single **.qip** file is generated for each SOPC Builder system. However, some more complex SOPC Builder components generate a separate **.qip** file. In that case, the system **.qip** file references the component **.qip** file.

You can now simulate the system using any Altera-supported third party simulator, compile the system in the Quartus II software, and configure an Altera device.

Files Generated by SOPC Builder

When SOPC Builder generates your system, it creates a number of new folders and files in the project directory. [Figure 5–3](#) describes some of these files.

Figure 5–3. Files Generated By SOPC Builder



Simulate the Design

SOPC Builder automatically sets up the simulation environment for the PCI Compiler.

SOPC Builder creates the **pci_sim** directory in your project directory and copies the testbench files from

`<path>/pci_compiler/testbench/sopc/<language>/<core>` to the **pci_sim** directory.



The testbench files must be edited to add the PCI transactions that will be performed on the system. If you regenerate your system, SOPC Builder will not overwrite the testbench files in the **pci_sim** directory. If you want to use the default testbench files, first delete the **pci_sim** directory and then regenerate your system.

This section of the walkthrough uses the following components:

- The system you created using SOPC Builder
- Scripts created by SOPC Builder in the `c:\sopc_pci\chip_top_sim` directory
- The ModelSim software



You can also use any other supported third-party simulator.

- The PCI testbench files located in the `c:\sopc_pci\pci_sim` directory

SOPC Builder creates IP functional simulation models for all the system components. The IP functional simulation models are the **.vo** or **.vho** files generated by SOPC Builder in your project directory.



For more information on IP functional simulation models, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

The SOPC Builder-generated top-level file also integrates the simulation modules of the system components and testbenches (if available), including the PCI testbench. The Altera-provided PCI testbench can be used to verify the basic functionality of your PCI compiler system. SOPC Builder automatically configures the PCI testbench and copies the necessary files to your project directory in the **pci_sim** directory. The default configuration of the PCI testbench is predefined to run basic PCI configuration transactions to the PCI device in your SOPC Builder generated system. You can edit the PCI testbench **mstr_tranx.v** or **mstr_tranx.vhd** file to add more interesting PCI transactions.

You can also copy the `mstr_tranx.v` file from the SOPC Builder example directory located at `<path>/pci_compiler/sopc_flow_examples/verilog/pci_sim/verilog/mt32`.



For more information on the PCI testbench files refer to [Chapter 8, Testbench](#).

For this walkthrough, perform the following steps:

1. Start the ModelSim simulator.
2. In the simulator, change your working directory to `c:\sopc_pci\chip_top_sim`.
3. To run the script, type the following command at the simulator command prompt:

```
source setup_sim.do ↵
```

4. To compile all the files and load the design, type the following command at the simulator prompt:

```
s ↵
```

5. To see all the signals in the wave-default window, type the following command at the simulator prompt:

```
w ↵
```



Not all of the signals show up in the **Wave - Default** window. You need to explicitly add other signals of interest to the **Wave - Default** window.

6. To simulate the design, type the following command at the simulator prompt:

```
run 10000ns ↵
```

View the following PCI transactions in the ModelSim **Wave - Default** window:

- Configuration write operations on the command registers BAR0 and BAR1 of the Altera PCI MegaCore function, followed by configuration write operations on BAR0 and BAR1 of the PCI testbench target device (`trgt_tranx`).

- Configuration read operations on command registers BAR0 and BAR1 of the Altera PCI MegaCore function, followed by configuration read operations on BAR0 and BAR1 of the PCI testbench `trgt_trnx`.

You can view the following PCI transactions in the ModelSim **Wave - Default** window if you made the additions to the `mstr_trnx` file.

- A 32-bit burst memory write operation to the on-chip memory.
- A 32-bit burst memory read operation to the on-chip memory.
- Configure DMA to perform burst memory read transaction to move data from the PCI testbench `trgt_trnx` to on-chip memory.
- Configure DMA to perform burst memory write transaction to move data from the on-chip memory to PCI testbench `trgt_trnx`.

Compile the Design

Compile your design in the Quartus II software. Refer to Quartus II Help for instructions on performing compilation.

Altera provides constraint files to ensure that the PCI MegaCore function achieves PCI specification timing requirements in Altera devices. This walkthrough incorporates a constraint file included with PCI Compiler.



For more information on using Altera-provided constraint files in your design, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

For this walkthrough perform the following steps:

1. Open `c:\sopc_pci\chip_top.qpf` (the **chip_top** project) in the Quartus II software.



This is the same project you created in “PCI Compiler with SOPC Builder Flow Design Walkthrough” on page 5–2.

2. Choose **Utility Windows > Tcl Console** (View menu).
3. Source the generated constraint file by typing the following commands at the Quartus II Tcl Console command prompt:

```
source pci_constraints_for_pci_compiler.tcl ←
```

```
add_pci_constraints -pin_suffix _pci_compiler ←
```



You must use the `-pin_suffix` option for the PCI constraints to be applied correctly. This option appends the PCI Compiler instance name to the default PCI pin names. You must specify the leading underscore character in front of the instance name.

In this example, the PCI Compiler instance is named `pci_compiler`.

4. Monitor the Quartus II Tcl Console to see the actions performed by the script.

To verify the PCI timing assignments in your project, perform the following steps:

1. Choose **Start Compilation** (Processing menu) in the Quartus II software.
2. After compilation, expand the **Timing Analyzer** folder in the **Compilation Report** by clicking the + symbol next to the folder name. Note the values in the **Clock Setup**, **tsu**, **th**, and **tco** report sections.

Program a Device

After you have compiled the design, you must program the targeted Altera device and verify the design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the PCI-Avalon bridge before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model and produce a time-limited programming file.



For more information on IP functional simulation models, refer to the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

For more information on OpenCore Plus hardware evaluation using the PCI MegaCore functions, refer to *"Compliance Summary" on page 10*.

Upgrading Systems from a Previous Version

Follow the steps below to upgrade a system created in the previous version of SOPC Builder.

1. Start SOPC Builder and open the existing system.
2. Follow the instructions on the screen to upgrade your system.



The connections for prefetchable and non-prefetchable master BARs are lost when your system is upgraded.

3. On the Edit menu, click **Remove Dangling Connections**.
4. Re-connect the prefetchable and non-prefetchable master BARs.

This chapter describes the parameters available to configure PCI Compiler, including:

- "System Options-1"
- "Value of Multiple Pending Reads"
- "System Options-2"
- "PCI Configuration"
- "Avalon Configuration"

System Options-1

The **System Options - 1** tab in the PCI Compiler wizard allows you to specify the PCI-Avalon bridge's device mode and performance profile.



When you select a device mode and performance profile, a brief description displays just to the right of the selection, which can help you determine the appropriate option for your system.

PCI Device Mode

There are three available device type modes. The wizard uses your selections to define the operation of the targeted Altera device. For example, if you want the targeted Altera device to operate as a target-only peripheral, select the **PCI Target-Only Peripheral**.

This section defines the following PCI device type modes:

- PCI Master/Target Peripheral
- PCI Target-Only Peripheral
- PCI Host-Bridge Device

PCI Master/Target Peripheral

This mode allows Avalon-MM master devices to access PCI target devices via the PCI bus master interface, and PCI bus master devices to access Avalon-MM slave devices via the PCI bus target interface.

Select this option if you are constructing an SOPC Builder system with:

- Avalon-MM peripherals initiating accesses to PCI devices
- PCI devices accessing Avalon-MM peripherals
- The system Host processor located on the PCI bus side

Selecting the **PCI Master/Target Peripheral** mode results in the PCI-Avalon bridge instantiating either the Altera `pci_mt32` or `pci_mt64` MegaCore function. Your **PCI Data Bus Width** selections in the **System Options - 2** tab will make the final determination between MegaCore functions.

PCI Target-Only Peripheral

This mode allows PCI bus mastering devices to access Avalon-MM slave devices via the PCI bus target interface. Select this option if you are constructing an SOPC Builder system that does not have Avalon-MM master devices accessing PCI bus devices. Refer to [Figure 7-2 on page 7-7](#) and [Figure 7-3 on page 7-8](#).

Selecting the **PCI Target-Only Peripheral** mode results in the PCI-Avalon bridge instantiating either the `pci_t32` or `pci_t64` MegaCore function. Your **PCI Data Bus Width** selections in the **System Options - 2** tab will make the final determination between MegaCore functions.

PCI Host-Bridge Device

In addition to the same features provided by the **PCI Master/Target Peripheral** mode, the PCI Host-Bridge Device mode provides host bridge functionality including hardwiring the master enable bit to 1 in the PCI command register and allowing self configuration. Hardwiring the master enable bit to 1 allows the PCI master device to initiate master transactions upon power up. Self configuration is needed to enable the PCI-Avalon bridge's PCI master device to access its own PCI configuration header for the PCI bus enumeration.

Select **PCI Host-Bridge Device** mode when the Host processor is on the Avalon-MM side of the PCI-Avalon bridge. Selecting this mode results in the PCI-Avalon bridge instantiating either the `pci_mt32` or `pci_mt64` MegaCore function. Your **PCI Data Bus Width** selections in the **System Options - 2** tab will make the final determination between MegaCore functions.

PCI Target Performance

This field lists the three available PCI target performance profile options. The wizard uses your selections to determine read and write operation throughput generated by PCI bus mastering devices to Avalon-MM slave peripherals.

Performance profile options allow you to balance trade-offs between preserving device resources and managing overall performance. For example, if you are constructing a target peripheral for a low-latency, low-bandwidth application, selecting the **Single-Cycle Transfers Only** performance profile suits the application's requirements while preserving resources for other system needs.

This section defines the following PCI target performance profile options:

- **Single-Cycle Transfers Only**
- **Burst Transfers with Single Pending Read**
- **Burst Transfers with Multiple Pending Reads**
- **Maximum Target Read Burst Size**

Single-Cycle Transfers Only

The option provides a low-latency and low-throughput option yielding to the smallest resource utilization and using no internal RAM blocks for PCI target accesses. When selecting this option, all PCI target transactions are automatically broken into single data phase transactions and utilize the Non-prefetchable Avalon-MM master port on the PCI-Avalon bridge. [Figure 7-2 on page 7-7](#) illustrates a system using this performance profile.

With this option, all accesses from the PCI bus utilize the non-prefetchable data path shown in [Figure 7-2 on page 7-7](#).

Selecting the **Single-Cycle Transfers Only** option is generally appropriate when you are constructing a system needing either no memory access or minimum memory access performance from the PCI bus to the Avalon-MM bus peripherals.

Burst Transfers with Single Pending Read

This option is typical for many PCI bus systems where PCI bus devices access one or more Avalon-MM peripherals. This option allows for burst and single cycle accesses from the PCI bus. Because accesses to non-prefetchable BARs are serviced as **Single-Cycle Transfers Only**, only accesses to prefetchable BARs benefit from the increase in performance.

In fact, if the PCI-Avalon bridge has no prefetchable BARs, it defaults to the **Single-Cycle Transfers Only** performance profile—even if you select the **Burst Transfers with Single Pending Read** performance profile.

This option maximizes PCI write transaction performance, but only allows one single pending read. A pending read is the same as an in-progress delayed read in PCI bus terminology and is defined as a PCI read transaction that was retried on the PCI bus while the PCI-Avalon bridge retrieves the data from the appropriate Avalon-MM peripheral. In this mode, only one pending read is processed at a time; all other read transactions are retried without being processed.

[Figure 7–3 on page 7–8](#) illustrates a system using this performance profile and has both prefetchable and non-prefetchable BARs defined. Accesses to non-prefetchable BARs utilize the non-prefetchable data path, while accesses to prefetchable BARs utilize the prefetchable data path. If no non-prefetchable BARs are used, the non-prefetchable data path logic will be removed from the bridge.

Select this option for systems that require burst and single-cycle accesses to Avalon-MM peripherals but do not require the highest performance for memory reads.

Burst Transfers with Multiple Pending Reads

This option is similar to the **Burst Transfers with Single Pending Read** option except that it allows up to four pending reads, which provides a significantly higher throughput for PCI memory read operations. Additionally, this option also provides the highest throughput for PCI target accesses and is the most resource intensive of the three target performance profile options.

For more information on the value of multiple pending reads, refer to [“Value of Multiple Pending Reads” on page 6–6](#).

Maximum Target Read Burst Size

This option allows you to configure the maximum read burst. The maximum dword is prefetched from the Avalon-MM regardless of the amount of data required on the PCI bus. If the application constantly requests large burst read on the PCI bus, set this option to a larger burst size .

PCI Master Performance

This field lists the two available PCI master performance profile options. The wizard uses your selections to determine read and write operation throughput generated by Avalon-MM master devices to PCI target devices.

This section defines the following PCI master performance profile options:

- Burst Transfers with Single Pending Read
- Burst Transfers with Multiple Pending Reads

Burst Transfers with Single Pending Read

This option allows burst and single-cycle accesses from Avalon-MM master devices, which includes memory, I/O, and configuration transactions.

The PCI-Avalon bridge contains either a dynamic or fixed Avalon-to-PCI address translation table. Depending on the address translation entry, the corresponding PCI address and command is generated by the PCI-Avalon bridge.

In this mode, only one pending read is serviced at a time. All subsequent reads are stored in a temporary queue holding up to eight transactions until the current pending read transaction is finished. If read transactions can be stored in the temporary read queue, write transactions are allowed to pass the read transactions.

Select the **Burst Transfers with Single Pending Read** performance profile either for:

- General purpose systems
- Data-intensive systems that utilize write operations to move data and use minimum read operations

Burst Transfers with Multiple Pending Reads

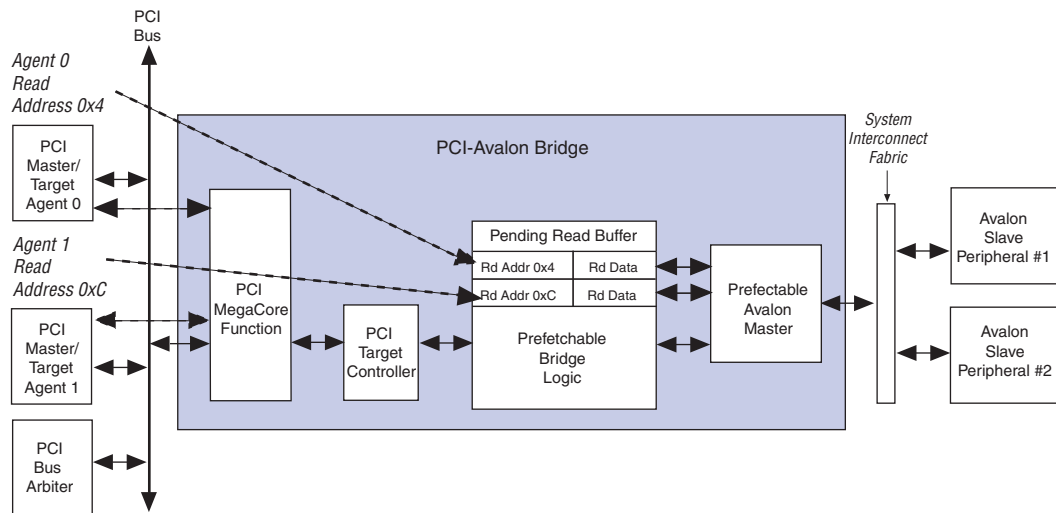
This option is similar to the **Burst Transfers with Single Pending Read** option except that it allows up to four pending reads. In other words, instead of issuing one read at a time, up to four simultaneous reads can be issued on the PCI bus. This allows PCI target devices to return read data while also reducing the read completion times.

Although up to four pending reads can be issued on the PCI bus, read data is returned in the order it is issued to the interconnect. This performance profile provides a significant improvement in PCI read operations for systems that rely on read operations to transfer data from PCI devices to Avalon-MM devices.

Value of Multiple Pending Reads

This section explains the enhanced performance that is possible with multiple pending read transactions. [Figure 6–1](#) illustrates the following burst transfer with multiple pending reads example:

1. PCI Agent 0 requests a read transaction (R0) to address 0x4. The PCI-Avalon bridge issues a PCI retry, stores the necessary information from the R0 transaction, and begins to retrieve the requested data from the PCI-Avalon bridge.
2. Before the R0 transaction completes, PCI Agent 1 requests a read transaction (R1) to address 0xC. The PCI-Avalon bridge issues another PCI retry and stores the necessary information to retrieve the data for R1. Meanwhile, the PCI-Avalon bridge continues to retrieve the data for R0.
3. At some point the data for R0 is returned, and the PCI-Avalon bridge immediately begins retrieving the data for R1.
4. PCI Agent 0 issues R0 again, and the PCI-Avalon bridge provides the requested data and completes R0. Meanwhile, the data for R1 is returned from the Avalon-MM peripheral.
5. PCI Agent 1 issues R1 again, and the PCI-Avalon bridge provides the requested data and completes R1.

Figure 6–1. PCI-Avalon Bridge Burst Transfer with Multiple Pending Reads

In contrast, the same two reads will complete in the following sequence if multiple pending read transactions are not allowed:

1. PCI Agent 0 requests a read transaction (R0) to address 0x4. The PCI-Avalon bridge issues a PCI retry, stores the necessary information from the R0 transaction, and begins to retrieve the requested data from the Avalon-MM peripheral.
2. Before the R0 transaction completes, PCI Agent 1 requests a read transaction (R1) to address 0xC. Because the PCI-Avalon bridge is currently servicing the R0 transaction, it simply issues a retry in response to R1 and continues to retrieve the data for R0. In this case, no information about R1 is stored in the PCI-Avalon bridge.
3. At some point the data for R0 is returned by the Avalon-MM peripheral. Meanwhile, all read transactions are automatically retried by the PCI-Avalon bridge.
4. PCI Agent 0 reissues the R0 transaction, and the PCI-Avalon bridge provides the requested data and completes the R0 transaction.
5. At a later time, PCI Agent 1 attempts R1 again. The PCI-Avalon bridge issues a PCI retry, stores the necessary information from the R1 transaction, and begins to retrieve the requested data the from Avalon-MM peripheral.

6. At some point the data for R1 is returned by the Avalon-MM peripheral. Meanwhile, all read transactions are automatically retried by the PCI-Avalon bridge.
7. PCI Agent 1 issues R1 again, the PCI-Avalon bridge provides the requested data and completes R1.

For more information on multiple pending read transactions, refer to [“Prefetchable Read Operations” on page 7-23](#) and [“Avalon-to-PCI Read Requests” on page 7-32](#).

System Options-2

The **System Options - 2** tab in the PCI Compiler wizard defines the complexity of the PCI-to-Avalon bridge. On this tab, you specify the following PCI bus configurations:

- PCI Bus Speed
- PCI Global Reset Signal
- PCI Data Bus Width
- Clock Domains
- PCI Bus Arbiter

PCI Bus Speed

PCI Bus Speed selections determine the system's maximum clock rate. If you select the 66-MHz clock rate, the capable bit (`pci_66mhz_capable`)—which is bit 5 of the PCI status register—is set to 1. Refer to [Table 3-17 on page 3-33](#).

PCI Data Bus Width

PCI Data Bus Width selections determine the system's PCI bus width. Additionally, the PCI-Avalon bridge automatically configures the Avalon-MM data width to be the same as the PCI data bus width. Therefore, selecting a 64-bit PCI data bus instantiates either the `pci_mt64` or `pci_t64` MegaCore function, depending on the **PCI Device Mode** you specified in the **System Options - 1** tab. Likewise, selecting a 32-bit PCI data bus instantiates either the `pci_mt32` or `pci_t32` MegaCore function, depending on the **PCI Device Mode** you specified in the **System Options - 1** tab.

PCI Clock/Reset Settings

Turning on the **Enable Independent Avalon System Reset** enables a second reset signal named `reset_n` in the SOPC system. When triggered, this signal resets the entire SOPC system except the PCI Compiler, thus preserving its configuration.

The **Independent PCI and Avalon Clocks** option allows the PCI bus and Avalon-MM interface to use independent clocks that can run at the same or different speeds. Choosing this option provides maximum flexibility but also requires more logic resources and adds latency. When you select this option, the SOPC Builder generates a system with two clock pins driving the PCI-Avalon bridge. One clock pin has the PCI-Avalon bridge's instance name and should be connected to the PCI clock source. The other clock pin has the PCI-Avalon bridge's instance name appended to it and should be connected to the system's clock source.

The **Shared PCI and Avalon Clocks** option allows the PCI bus and Avalon-MM interface to use a single clock, resulting in a simpler system that uses fewer logic resources. Additionally, the resulting system's latency is lower than is possible with separate clocks. When you select this option, the SOPC Builder generates a system with only one clock pin driving the PCI-Avalon bridge. The clock pin name will not have the PCI-Avalon bridge instance name, and should be connected to the PCI clock source.

PCI Bus Arbiter

The **PCI Bus Arbiter** options are disabled if you select **PCI Target-Only Peripheral** mode. If you select either **PCI Master/Target Peripheral** or **PCI Host-Bridge Device** mode, you can define how the `reqn` and `gntn` signals are routed. The default is to route the signals to external pins.

The following defines the **PCI Bus Arbiter** selections:

- **Arbiter External to Device**—this is the most common option and is also the default setting. Because the PCI bus arbitration is done outside of the FPGA device, the **Arbiter External to Device** option is common for all PCI add-on applications. Selecting this option routes the `reqn` and `gntn` signals to pins.
- **User-Defined Arbiter Internal to Device**—Selecting this option allows you to connect the `reqn` and `gntn` signals of the PCI-Avalon bridge to internal logic and not drive them to pins. This option disables the tri-state buffer on the `reqn` signal.
- **Altera-Provided Arbiter Internal to Device**—This option enables the arbiter shipped with the PCI-Avalon bridge. Similar to the **User-Defined Arbiter Internal to Device** option, this option disables the tri-state buffer on `reqn` signal. Additionally, this option wires the `reqn` and `gntn` signals to the provided arbiter as device #0, i.e., the `reqn` and `gntn` signals are internally connected to `ArbReq_n_i[0]` and `ArbGnt_n_o[0]` respectively.

If you select this option, you need to specify the number of PCI devices supported by the MegaCore function. The provided arbiter can support up to eight PCI devices. The number of devices supported includes the PCI-Avalon bridge; therefore, if you select two devices, there will be one PCI device in addition to the PCI-Avalon bridge device on the PCI bus.



To implement a Host bridge device with no other PCI master-capable devices, select the **User-Defined Arbiter Internal to Device** option and connect the PCI gntn input port to a physical 0. The bus will always be granted to the Host Bridge.

PCI Configuration

The third tab of the PCI Compiler wizard sets up the **PCI Base Address Registers (BARs)** and the **PCI Read-Only Registers**.

PCI Base Address Registers

In the **PCI Base Address Register** box you define the number and type of BARs as well as the system's BAR address range. Up to six 32-bit BARs can be defined. If you select **64 Bit PCI Bus** in the **PCI Data Bus Width** field (**System Options - 2** tab), you have the option of defining one 64-bit BAR and up to four 32-bit BARs. Refer to [“Setting the PCI Base Address Register Values” on page 6–11](#).

You have the option to disable the I/O ordering between the non-prefetchable master and prefetchable master BARs in Target Only mode. If you turn on **Disable IO Ordering between Non-Prefetchable and Prefetchable BARs**, the non-prefetchable master write/read is no longer dependent on the prefetchable master write/read, resulting in low latency non-prefetchable master write/read.



Disabling the I/O ordering between the non-prefetchable master and prefetchable master BARs violates the PCI ordering rules for bridge specification. It should therefore be used only in embedded applications where designers can control the ordering via the applications.

PCI Read-Only Registers

The values in the **PCI Read-Only Registers** box can all be edited in place. When you change a value, the validity of the new value is automatically checked. If the new value is out of range, the previous legal value is substituted. For example, if you enter an illegal value such as 0xFFFF for the Vendor ID, the value will be returned to its previous value. In addition, a message is displayed that explains why the edit was ignored. This message must be dismissed before you can proceed with other edits.

Setting the PCI Base Address Register Values

For transactions initiated on the PCI bus with a destination on the Avalon-MM bus, the PCI bus address must be translated into an Avalon-MM address. The PCI-Avalon bridge claims the PCI transaction

if the address matches one of the BARs. The PCI-Avalon bridge then translates the PCI address into an Avalon-MM address before it initiates the equivalent transaction on the interconnect. The PCI BAR settings in the PCI Compiler wizard are used to set the appropriate options for the PCI BAR, so the transactions from PCI can accurately flow to the interconnect.

For each BAR you must set the following options:

- **BAR Type**
- **BAR Size**
- **Avalon Base Address**
- **Hardwired PCI Address**

The following sections explain how to select the appropriate settings for each option.

BAR Type—The PCI-Avalon bridge supports three BAR types:

- **32-Bit Prefetchable Memory:** This type of BAR is typical for most systems. It is used for Avalon-MM I/O. Implementing at least one 32-bit prefetchable BAR enables a prefetchable master port for the PCI-Avalon bridge—except if you select the Single-Cycle Transfers Only performance profile. This enables both burst and single cycle access to Avalon-MM peripherals for both read and write transactions.

You can use this option for all types of Avalon-MM peripherals except those that do not support prefetchable read transactions. Peripherals that support prefetchable read transactions do not modify the state of the data when a read operation is performed. A RAM or ROM is a typical example of a prefetchable peripheral. Peripherals where a read operation changes the state of the data, such as a FIFO buffer or a clear-on read register, are called non-prefetchable and should not be accessed by a prefetchable base address register.

- **64-Bit Prefetchable Memory:** This BAR is similar to the 32-bit prefetchable memory BAR except that it supports 64-bit PCI addressing. This option is only available if you select **64 Bit PCI Bus** in the **System Options - 2** tab. The requirement for your device to support 64-bit addressing is usually apparent from your system architecture and is driven by the amount of system memory. Because the Avalon-MM address space can only support 32-bit addresses, you are limited to the amount of address space that you can reserve

in this BAR type to 2 GBytes. In other words, this BAR type allows your device to reside anywhere in the 64-bit address space, but does not allow you to reserve more than 2 GBytes.

Only one 64-bit prefetchable BAR is allowed in a system. All other BARs you define are 32-bit BARs.

- **32-Bit Non-Prefetchable Memory:** Non-prefetchable memory address is normally used for register or memory space where the read operation can modify the state of the data you read. Implementing one or more 32-bit non-prefetchable BARs enables a non-prefetchable master port for the PCI-Avalon bridge. If the transaction address matches a non-prefetchable memory BAR, only single cycle transactions for both read and write operations are performed.
- **I/O:** Implementing at least one IO BAR enables a non-prefetchable master port for the PCI-Avalon bridge. Only single-cycle reads and writes are supported. All I/O reads and writes are non-posted, and handled as delayed operations. The amount of address space a device requests can range between 4 and 256 bytes, inclusively. I/O address space decoding for legacy devices is supported, as described in Appendix G of the PCI Specifications.

BAR Size—The BAR size must be set to encompass the addresses of all the Avalon-MM peripherals that you wish to access with the BAR. The BAR size must be greater than or equal to the range of Avalon-MM address that it accesses.

You can configure the BAR size either manually or automatically. To automatically set the BAR size, select **Auto** for BAR size. **Auto** will automatically set the BAR size to encompass the entire address space for the Avalon-MM peripherals that are addressed by the BAR.

Avalon Base Address—The Avalon-MM base address corresponds to the PCI base address. Based on the BAR size setting, the PCI-Avalon bridge replaces the PCI base address with the Avalon-MM base address. In other words, the read/write bits of the PCI Base Address Register are replaced with the equivalent Avalon-MM base address bits.

You can configure the Avalon-MM base address either manually or automatically. To automatically set the Avalon-MM base address, select **Auto** for the BAR size. Refer to [“Manual Setting of the BAR Size & Avalon Base Address”](#) for information on setting the size of the BAR.

Hardwired PCI Address—The hardwired PCI address setting allows you to hardwire the PCI BAR so the system software does not configure it at run time. To use this option, select **YES** and enter a hardwired BAR value.

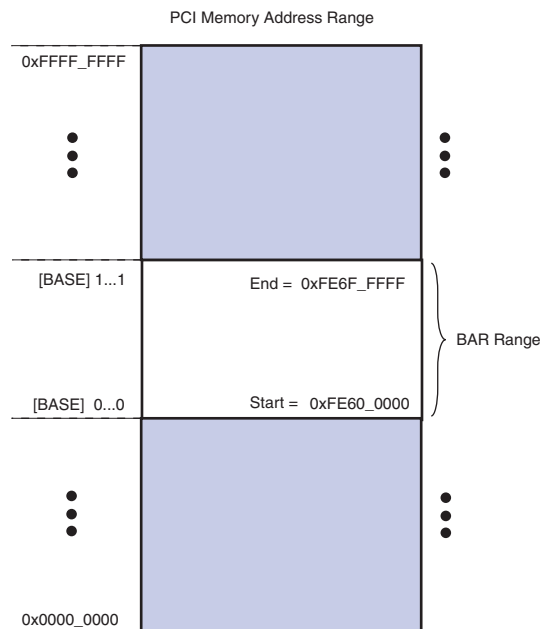


The Hardwired PCI Address option is generally useful for the PCI Host-Bridge Device mode and embedded applications, because system designers have complete control over system configuration. This option is not recommended for other applications.

Manual Setting of the BAR Size & Avalon Base Address

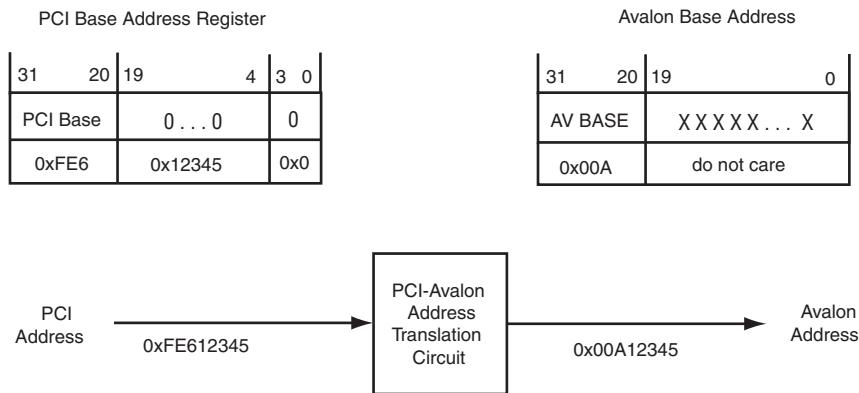
The number of high-order bits (starting from the most significant bit) in the BAR that the PCI device implements as read/write bits determines the size of address to which it will respond. A 32-bit BAR can be implemented to support a contiguous memory size that is a power of 2 from 1 KByte to 2 GBytes. If you select a base address size of 1 MByte (using a 32-bit BAR), the PCI-Avalon bridge will implement the 12 most significant bits (bits 31-20) of the base address registers as read/write and hardwire the remaining bits to 0 (except for the least significant 4 bits because they have special use). [Figure 6–2](#) shows a pictorial view of the PCI BAR address range.

Figure 6–2. Pictorial View of the 32-Bit PCI BAR



The PCI-Avalon address translation circuit requires that you supply the PCI-Avalon bridge with the Avalon-MM base address value that corresponds to the PCI base address value. The PCI-Avalon address translation circuit replaces the most significant bits of the PCI address (the read/write bits of the PCI BAR) and replaces them with the corresponding most significant bits of the Avalon-MM base address. [Figure 6–3](#) shows a pictorial view of the process, assuming a 1 MByte PCI BAR size.

Figure 6–3. Pictorial View of the PCI-to-Avalon Address Translation Circuit



A simplified way to determine the size and Avalon-MM base address is shown in the following procedure ([Table 6–1](#)):

1. Record the base and end addresses of each Avalon-MM peripheral you want to address with one BAR. List the addresses vertically with the lowest base address in the bottom and the highest end address on the top.
2. Starting from the most significant bit (bit 31), count the number of bits (w) that remain constant for all the addresses you wrote. If the number of bits is w , the size of the BAR is 2^{32-w} .
3. The corresponding base value is the smallest base address for the Avalon-MM peripherals.

Because you can only address a 32-bit space on the Avalon-MM side, the size calculation is the same for both the 32-bit BAR and 64-bit BAR.

In some cases, you should try to adjust the Base Address of some peripherals to help reduce the amount of address space you reserve with the PCI BAR. In [Table 6–1](#) the only difference between Example B and Example C is the location of the second peripheral. In Example B the BAR size is 16 MBytes, while in Example C the size is 64 KBytes.

Table 6–1. Determining the Size and Avalon Base Address of a BAR

Example	Avalon Peripheral Setting	Calculations	PCI BAR Setting
A	End1 = 0x0001_FFFF Base1 = 0x0001_0000	W = 16 Size = $2^{(32-16)} = 2^{16} = 64K$ Base = 0x0001_0000	BAR size = 64 KBytes – 16 bits Avalon Base Address = 0x0001_0000
B	End2 = 0x0A80_OFFF Base2 = 0x0A80_0000 End 1 = 0x0A00_7FFF Base1 = 0x0A00_4000	W = 8 Size = $2^{24} = 16$ MBytes Base = 0x0A00_4000	BAR Size = 16 MBytes – 24 bits Avalon Base Address = 0x0A00_0000
C	End2 = 0x0A00_8FFF Base2 = 0x0A00_8000 End 1 = 0x0A00_7FFF Base1 = 0x0A00_4000	W = 16 Size = $2^{16} = 64$ KBytes Base = 0x0A00_4000	BAR Size = 64 KBytes – 16 bits Avalon Base Address = 0x0A00_0000

Avalon Configuration

The **Avalon Configuration** tab of the PCI Compiler wizard is used to configure the address mapping from Avalon-MM addresses to PCI addresses. This map can be dynamically configured at run time or hardwired. If you choose to hardwire the map, the controls on this page are used to define the hardwired map.



If you select the **PCI Target-Only Peripheral** mode from the **System Options - 1** tab, all options on the **Avalon Configuration** tab, except **Avalon CRA Port**, will be disabled.

Under **Address Translation Table Configuration**, selecting either **Dynamic Translation Table** or **Fixed Translation Table** options determines the maximum number of pages for the window from Avalon to PCI. If you select **Fixed Translation Table**, the number of pages is limited to 16. The maximum number of pages for a dynamically configured address map is 512.

In the **Address Translation Table Size** field, you set the number and size of address pages. The lower limit on the size of the pages is 4 KBytes and the upper limit is 2 GBytes. The maximum number of pages is limited by either the decision to hardwire the map or the size of the page. The maximum number of pages is the smaller of:

- 16 if hardwired or 512 if dynamically configured
- 2 GBytes divided by the size of each page

For example, if you select a page size of 512 MBytes, the maximum number of pages is four for both hardwired and dynamically configured.

If you select a smaller page size that yields more than 16 pages and then change to a hardwired address map, the number of pages is automatically decreased to 16.

The **Fixed Address Translation Table Contents** field is disabled if you select **Dynamic Translation Table**. If you select **Fixed Translation Table**, the number of rows that appear in the **Contents** panel is the same as the value you enter in the **Number of Address Pages** box. For each **Avalon Address Offset**, you can set a **PCI Base Address** and **Type** value. The choices for **Type** are 32-bit/64-bit memory, I/O, and configuration. Refer to [“Avalon-to-PCI Address Translation” on page 7–35](#).

Turning on **Control Register Access Avalon Slave Port** option under **Avalon CRA Port** allows read/write accesses to the bridge’s registers. Disabling this option means that no read/write accesses will be granted to the bridge’s registers. There are two cases in which this option is always enabled:

- If **Dynamic Translation Table** is selected in the **Translation Table** panel
- If **PCI Host-Bridge Device** is chosen as the device type on the **System Options - 1** page

This chapter provides specification details for the PCI Compiler with SOPC Builder flow, including:

- [Functional Overview](#)
- [Interface Signals](#)
- [PCI Bus Commands](#)
- [PCI Target Operation](#)
- [PCI Master Operation](#)
- [PCI Host-Bridge Operation](#)
- [Altera-Provided PCI Bus Arbiter](#)
- [Interrupts](#)
- [Control & Status Registers](#)

Functional Overview

The PCI Compiler with SOPC Builder flow uses the PCI-Avalon™ bridge to connect the PCI bus to the on-chip system interconnect fabric, allowing you to easily create simple or complex PCI systems that include one or more of the SOPC Builder components.

Because the PCI Compiler with SOPC Builder flow uses a predefined interconnect, system development is completely driven by the SOPC Builder graphical user interface (GUI). For example, you specify bridge and component parameterization options, and the PCI Compiler wizard sets up the bridge's structure, component features, and system-wide interconnect.

To make a complete PCI-Avalon bridge, the PCI Compiler instantiates the Altera `pci_t32`, `pci_t64`, `pci_mt32`, or `pci_mt64` MegaCore function as needed per user specifications.

The interconnect—which contains logic to manage system-wide connectivity—connects all components that make up the user-specified SOPC Builder system. For example, when the targeted Altera device is operating in the PCI Master/Target Peripheral mode, the PCI-Avalon bridge (via the interconnect) is managing the connectivity of multiple master and slave components.

This section discusses:

- PCI-Avalon bridge module blocks
- PCI operational modes
- Performance profiles
- OpenCore Plus time-out behavior

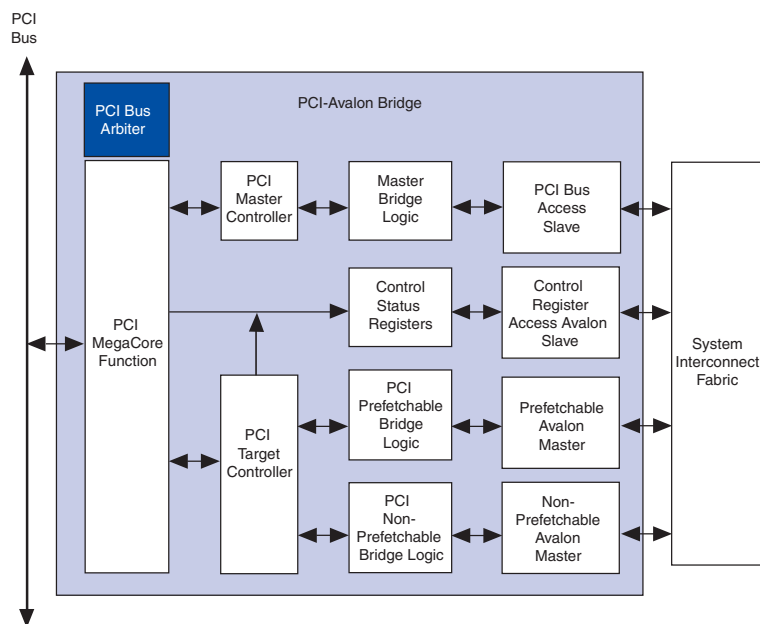
PCI-Avalon Bridge Blocks

The PCI-Avalon bridge's blocks provide a feature-rich foundation that enables the bridge to manage the connectivity for all three PCI operational modes:

- PCI Target-Only Peripheral
- PCI Master/Target Peripheral
- PCI Host-Bridge Device

Depending on the operational mode, the PCI-Avalon bridge uses some or all of the predefined Avalon-MM ports. [Figure 7–1](#) shows a generic PCI-Avalon bridge block diagram, which includes the following blocks:

- Four predefined Avalon-MM ports
- Control/status registers
- PCI master controller (when applicable)
- PCI target controller
- PCI bus arbiter (for Master/Target and Host bridge mode)
- Instantiated Altera PCI MegaCore function

Figure 7–1. Generic PCI-Avalon Bridge Block Diagram

Avalon-MM Ports

The Avalon bridge is comprised of up to four (depending on device operating mode) predefined ports to communicate with the interconnect.

This section discusses the four Avalon-MM ports:

- Prefetchable Avalon-MM master
- Non-Prefetchable Avalon-MM master
- PCI bus access slave
- Control register access Avalon-MM slave

Prefetchable Avalon-MM Master

The prefetchable Avalon-MM master port provides a high bandwidth PCI memory request access to Avalon-MM slave peripherals. This master port is capable of generating Avalon-MM burst requests for PCI requests that hit a prefetchable base address register (BAR). You should only connect prefetchable Avalon-MM slaves to this port, typically RAM or ROM memory devices.

This port is optimized for high bandwidth transfers as a PCI target and is optional for PCI master/target peripherals that do not need to support burst transactions as a PCI target.

This port is enabled when you perform both of the following:

- Select one of the following target performance settings:
 - Burst Transfers with Single Pending Read
 - Burst Transfers with Single or Multiple Pending Reads
- Implement at least one prefetchable BAR

Non-Prefetchable Avalon-MM Master

The Non-Prefetchable Avalon-MM Master port provides a low latency PCI memory request access to Avalon-MM slave peripherals. Burst operations are not supported on this master port. Only the exact amount of data needed to service the initial data phase will be read from the interconnect. Therefore, the PCI byte enables (for the first data phase of the PCI read transaction) are passed directly to the interconnect.

This Avalon-MM master port is also optimized for low latency access from PCI-to-Avalon-MM slaves. This is optimal for providing PCI target access to simple Avalon-MM peripherals. This port is optional for implementations that do not need non-prefetchable access to peripherals.

If you select Single-Cycle Transfers Only target performance profile, this port will be the only Avalon-MM master port instantiated.

PCI Bus Access Slave

This Avalon-MM slave port is used to propagate the following transactions from the interconnect to the PCI bus:

- Single cycle memory read and write requests
- Burst memory read and write requests
- I/O read and write requests
- Configuration read and write requests

Burst requests from the interconnect are the only way to create burst transactions on the PCI bus.

This slave port is not implemented in the PCI Target-Only Peripheral mode.

Control Register Access Avalon-MM Slave

This Avalon-MM slave port is available to all three PCI device modes and is used to access various control and status registers in the PCI-Avalon bridge. To provide external PCI master access to these registers, one of the

bridge's master ports must be connected to this port. There is no internal access inside the bridge from the PCI bus to these registers. You can only read-from and write-to these registers from the interconnect.

Control/Status Register Module

The PCI-Avalon bridge provides a rich set of control and status registers including mailbox registers. To access these registers, you must enable the `Control Register Access Avalon Slave port`.

Mailbox Registers

The PCI-Avalon bridge provides two sets of mailbox registers. These registers enable PCI and Avalon-MM masters to pass one DWORD of data and assert an interrupt. To use the mailbox registers, you must enable the `Control Register Access Avalon Slave port`.

One set of mailbox registers is used by external PCI masters. When a PCI master writes a 32-bit value to a mailbox register, an Avalon-MM interrupt is asserted. The number of available mailbox registers is determined by the PCI-to-Avalon performance profile.

The second set of mailbox registers is used by Avalon-MM masters. When an Avalon-MM master writes a 32-bit value to an Avalon-PCI mailbox register, a PCI interrupt is generated. The number of Avalon-PCI mailbox registers depends on the target performance profile. Refer to [Table 7-1](#).

Table 7-1. Target Performance Profiles & Mailbox Registers Used

Target Performance Profile	Number of PCI-Avalon Mailbox Registers	Number of Avalon-PCI Mailbox Registers
Single-Cycle Transfers Only	1	1
Burst Transfers with Single Pending Read or Burst Transfers with Multiple Pending Reads	8	8

PCI MegaCore Function

The PCI-Avalon bridge instantiates the appropriate PCI MegaCore function per user specifications. For example, if you select **64 Bit PCI Bus** from the **PCI Data Bus Width** field (**System Options - 2** tab), your system will use a 64-bit MegaCore function. Refer to [“Value of Multiple Pending Reads” on page 6-6](#).



To use the PCI-Avalon bridge you must license one of the Altera PCI MegaCore functions.

PCI Bus Arbiter

The PCI-Avalon bridge has an optional, integrated PCI bus arbiter that can be used in both the PCI Master/Target Peripheral and the PCI Host-Bridge Device operating modes. When using the PCI bus arbiter, the PCI-Avalon bridge will be automatically connected to requests and grants for device zero.

Other PCI-Avalon Bridge Modules

The remaining PCI-Avalon bridge modules contain embedded memory blocks, PCI MegaCore control modules, and bridge logic. These modules provide the circuitry to enable the bridge's functionality, i.e., transaction translation, clock domain crossing, and transaction ordering.

PCI Operational Modes

The targeted Altera device can operate in any one of the following modes:

- PCI Target-Only Peripheral
- PCI Master/Target Peripheral
- PCI Host-Bridge Device



MAX II devices only support the PCI Target-Only Peripheral mode and only Single-Cycle Transfers Only performance profile.

PCI Target-Only Peripheral Mode Operation

Figure 7–2 shows the block diagram of the PCI-Avalon bridge managing the connectivity of the PCI Target-Only Peripheral mode with the Single-Cycle Transfers Only performance profile. The configuration uses the Non-Prefetchable Master port and has a Host processor and bus arbiter on the PCI bus side. In the Single-Cycle Transfers Only performance profile, all PCI transactions are transferred via the Non-prefetchable Avalon-MM master port including access to prefetchable BARs.

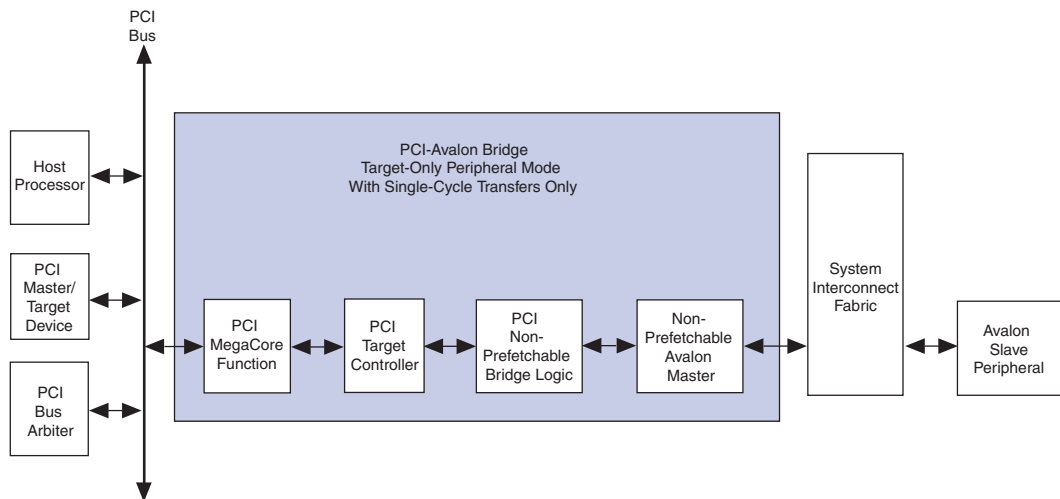
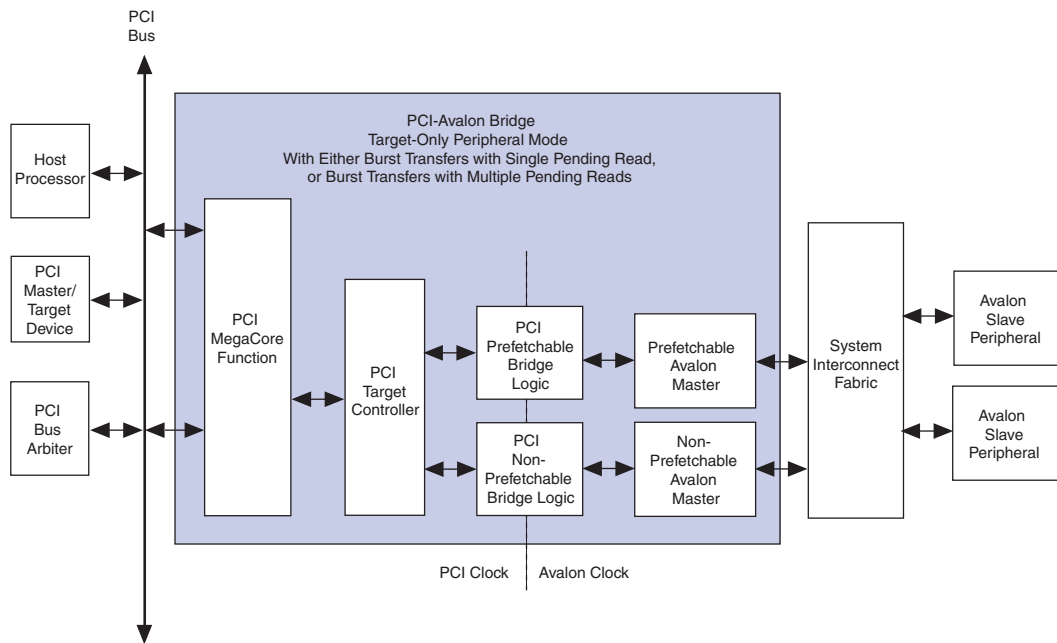
Figure 7-2. PCI-Avalon Bridge Managing the PCI Target-Only Peripheral Mode, Single-Cycle Transfers Only

Figure 7-3 shows the block diagram of the PCI-Avalon bridge managing the connectivity of the PCI Target-Only Peripheral mode with either the Burst Transfers with Single-Pending Read profile or the Burst Transfers With Multiple Pending Reads performance profile. The configuration uses two of the four Avalon-MM ports and has a Host processor and bus arbiter on the PCI side.



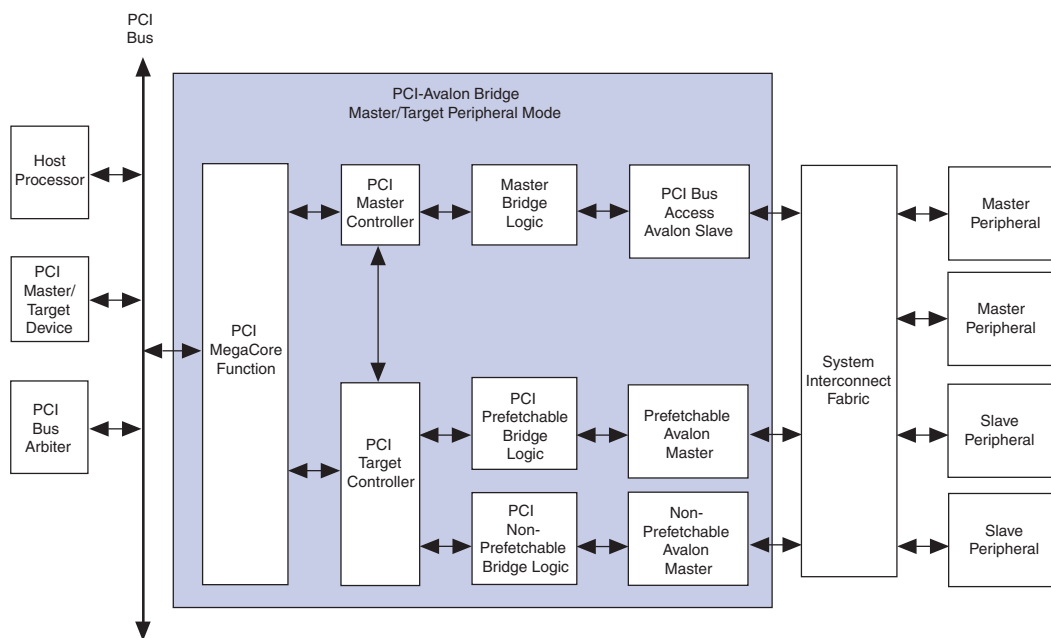
Because both the Prefetchable and Non-Prefetchable Avalon-MM master ports are instantiated, the Avalon bridge must have at least two memory BARs; one prefetchable memory BAR and one non-prefetchable memory BAR.

Figure 7–3. PCI-Avalon Bridge Managing the PCI Target-Only Peripheral Mode, Burst Transfers

You can customize the Target-Only mode by specifying one of the performance profiles. Refer to [“Performance Profiles”](#) on page 7–11.

PCI Master/Target Peripheral Mode Operation

[Figure 7–4](#) shows the block diagram of the PCI-Avalon bridge managing the connectivity of the PCI Master/Target Peripheral mode. The PCI Master/Target Peripheral mode uses at least one Avalon-MM master port, the PCI Bus Access Slave port, and has a Host processor and bus arbiter on the PCI bus side. [Figure 7–4](#) shows an example when both Avalon-MM master ports are used.

Figure 7–4. PCI-Avalon Bridge Block Diagram Managing the PCI Master/Target Peripheral Mode

You can customize the Master/Target mode by specifying one of the performance profiles. Refer to [“Performance Profiles” on page 7–11](#).

Control Register Access Avalon Slave

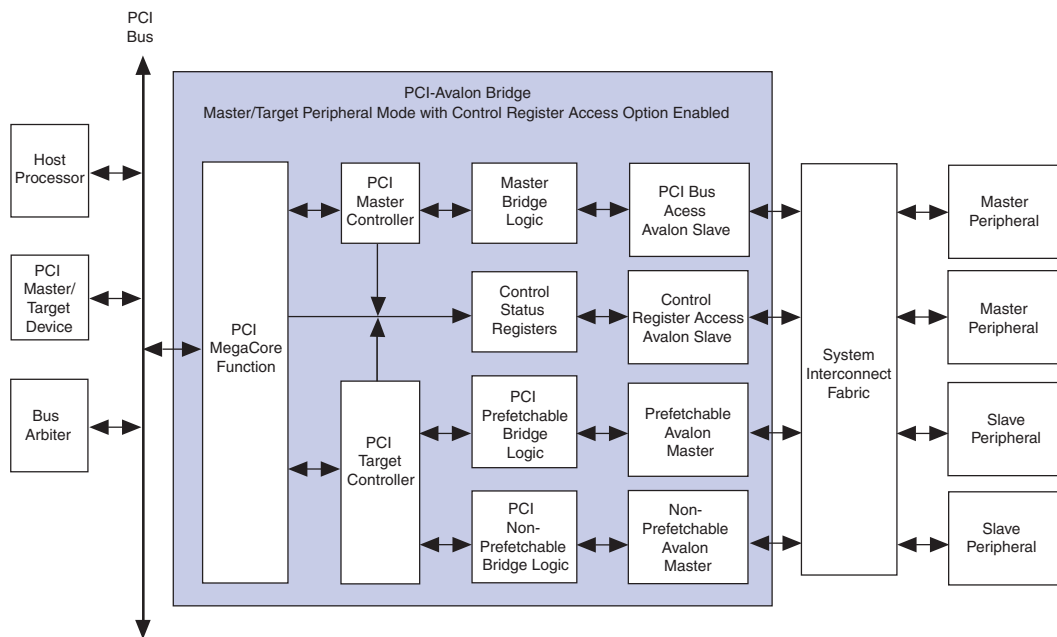
The PCI-Avalon bridge provides a rich set of user-accessible control/status registers. Implementing the registers is optional except when using the:

- PCI Host-Bridge Device mode
- Dynamic Avalon-to-PCI address translation option

The **Avalon Configuration** tab of the PCI Compiler wizard allows you to enable the control/status registers and specify access to them via the interconnect. Refer to [“Avalon Configuration” on page 6–16](#).

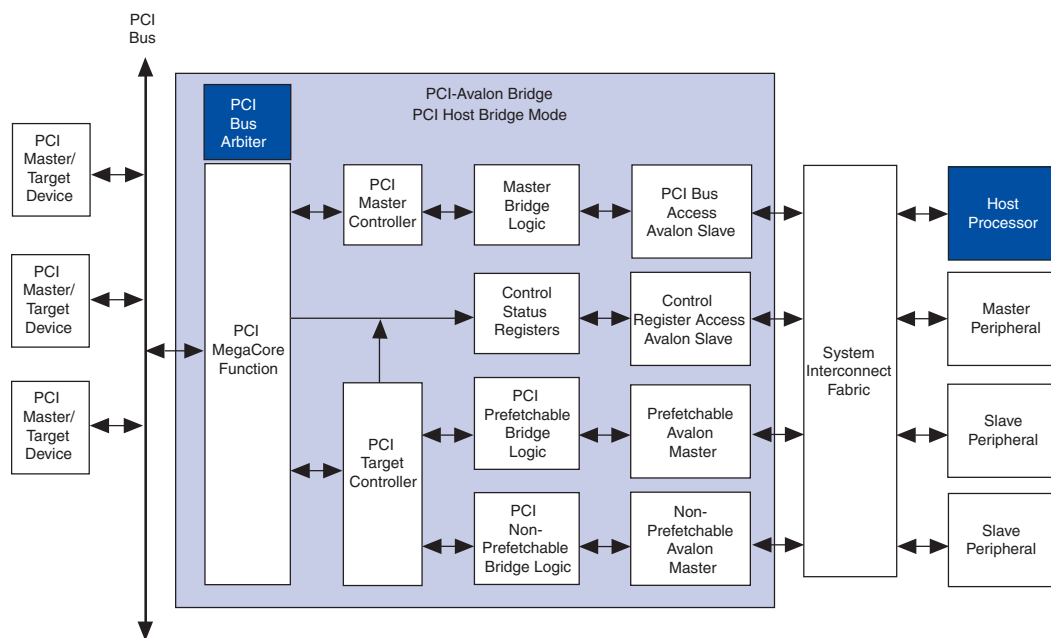
The control/status registers can be accessed from any Avalon-MM master device including PCI-Avalon bridge master ports. If you want to access the control/status registers from a PCI bus master device, you must use the SOPC Builder GUI to connect the Avalon-MM ports to the Control Register Access Avalon Slave port. Refer to [Figure 7–5](#).

Figure 7–5. PCI-Avalon Bridge Managing the PCI Master/Target Mode with Control Register Access Option Enabled



PCI Host-Bridge Device Mode Operation

Figure 7–6 shows the block diagram of the PCI-Avalon bridge managing the connectivity of the PCI Host-Bridge Device mode.

Figure 7–6. PCI-Avalon Bridge Block Diagram Managing the PCI Host-Bridge Device Mode

You can customize the PCI Host-Bridge Device mode by specifying a performance profile. The PCI Host-Bridge Device mode performance profiles are the same as the PCI Master/Target Peripheral mode.

Performance Profiles

The performance profiles are designed to provide trade-offs between performance and resource usage, which also provides a deeper level of system control. Therefore, you are able to customize system features to suit application requirements. For example, if you need a target-only PCI system component for a low bandwidth and low latency application, specify that the targeted Altera device operate in the PCI Target-Only Peripheral mode and use the Single-Cycle Transfers Only performance profile.

The PCI-Avalon bridge provides the ability to modify the PCI master and target performances independently. The PCI target performance setting applies to all PCI device operating modes, while the PCI master performance setting applies only to the PCI Master/Target Peripheral and PCI Host-Bridge Device operating modes.

Within each of the PCI operating modes, the targeted Altera device can use any of the performance profiles; the performance profiles are slightly different per device operating mode.

Target Performance

The PCI-Avalon bridge provides the following three target performance options:

- Single-Cycle Transfers Only
- Burst Transfers with Single Pending Read
- Burst Transfers with Multiple Pending Reads

Single-Cycle Transfers Only

This profile uses the least amount of resources and does not require embedded RAM blocks. This profile provides low latency and low bandwidth connectivity for Avalon-MM slave peripherals. Only the Non-Prefetchable Avalon-MM master port is enabled.

Burst Transfers With Single Pending Read

This profile allows high throughput read/write operations to Avalon-MM slave peripherals. Read/write accesses to prefetchable base address registers (BARs) use dual-port buffers to enable burst transactions on both the PCI and Avalon-MM sides. This profile also allows access to non-prefetchable PCI BARs to use the Non-prefetchable Avalon-MM master port to initiate single-cycle transfers to Avalon-MM slave peripherals. All PCI read transactions are completed as delayed reads. However, only one delayed read is accepted and processed at a time.

Burst Transfers With Multiple Pending Reads

This profile is exactly the same as the target Burst Transfers with Single Pending Read performance profile except that it allows up to four pending reads to be simultaneously processed.

Master Performance

The PCI-Avalon bridge provides the following two master performance options:

- Burst Transfers with Single Pending Read
- Burst Transfers with Multiple Pending Reads

Burst Transfers With Single Pending Read

This profile provides high throughput for transactions initiated by Avalon-MM master devices to PCI target devices via the PCI bus master interface. This profile uses embedded RAM blocks to enable clock domain crossing and efficient processing of single-cycle and burst transfers. Avalon-MM read transactions are implemented as latent read transfers. The PCI master devices issue only one read transaction at a time.

Burst Transfers With Multiple Pending Reads

This profile is exactly the same as the Burst Transfers with Single Pending Read for both the PCI Master/Target Peripheral and PCI Host-Bridge Device operational modes except that it can simultaneously process up to four pending reads. This allows higher throughput for read operations, but also requires more device resources.

For more information on multiple pending read transactions, refer to [“Value of Multiple Pending Reads” on page 6–6 and page 7–23.](#)

Interface Signals



For information about Avalon-MM interface signals and their functionality, refer to the [System Interconnect Fabric for Memory-Mapped interfaces](#) chapter in volume 4 of the *Quartus II Handbook*.

The SOPC Builder appends the instance name of the PCI Compiler component to all of the corresponding PCI-Avalon bridge component's signal names. For example, if the instance name of the PCI Compiler component is `pci_compiler`, all of PCI-Avalon bridge component's signal names will be `<signal name>_pci_compiler`, where `<signal name>` is the default signal name.

The SOPC Builder system containing the PCI-Avalon bridge has one asynchronous reset signal named `rstn_<pci_compiler_instance_name>`. This signal is used for the entire SOPC Builder system, including the PCI-Avalon bridge. A second reset signal, `reset_n`, can be enabled by selecting **Independent Avalon Reset Signal** when setting up the PCI-Avalon bridge. This signal is used for the entire SOPC Builder system except the PCI-Avalon bridge.

The connection of the reset signal, `rstn_<pci_compiler_instance_name>`, is generally a system-specific requirement and is outside the scope of this document. However, for most PCI applications, this reset signal should be connected to the PCI reset signal and must meet all PCI reset

requirements. If you use the PCI constraint files as recommended, the SOPC Builder reset signal will be assigned to the PCI reset signal and all PCI settings will automatically be made in your Quartus II project file.

Depending on the selected clock option, you may have one or more clock signals in your SOPC Builder system. There are two clock options (refer to [“Value of Multiple Pending Reads” on page 6–6](#)):

- If you select **Shared PCI and Avalon Clocks**, the resulting SOPC Builder system will have only one clock signal, `clk`. This pin must be connected to your device’s PCI clock signal and must have all of the appropriate PCI assignments in your Quartus II project. The PCI constraint files do not make the appropriate assignments.
- If you select **Independent PCI and Avalon Clocks**, the resulting SOPC Builder system will have at least two clock signals, `clk` and `clk_<pci_compiler_instance_name>`. The latter of these signals must be connected to your device’s PCI clock signal on your device and must have the appropriate PCI assignments in your Quartus II project. You can use the PCI constraint files to make all of the appropriate PCI assignments.



For more information on using PCI constraint files, refer to [Appendix A, Using PCI Constraint File Tcl Scripts](#).

PCI Bus Arbiter Signals

[Table 7–2](#) lists the PCI arbiter interface signals. These signals are only present when the **Altera-Provided Arbiter Internal to Device** option is selected in the PCI bus arbiter field (refer to [“Value of Multiple Pending Reads” on page 6–6](#)).

<i>Table 7–2. PCI Arbiter Ports</i>		
Name	Type	Description
<code>ArbReq_n_i[N-1]</code>	Input	Bus request inputs. These signals are asserted when the connected agent wants to master the PCI bus. Where <i>N</i> is the number of PCI devices supported by the arbiter.
<code>ArbGnt_n_o[N-1]</code>	Output	Bus grant outputs. These signals are asserted when the bus is granted to one of the attached devices. Where <i>N</i> is the number of PCI devices supported by the arbiter.

PCI Bus Commands

Table 7–3 shows the PCI bus commands support for PCI-Avalon bridge. The bus commands are discussed in greater detail in “PCI Target Operation” on page 7–15 and “PCI Master Operation” on page 7–27.

Table 7–3. PCI Bus Command Support Summary

Command Value	Bus Command Cycle	PCI Master (1)	PCI Target (2)
0b0000	Interrupt acknowledge	No	Ignored
0b0001	Special cycle	No	Ignored
0b0010	I/O read	Yes	Yes
0b0011	I/O write	Yes	Yes
0b0100	Reserved	Ignored	Ignored
0b0101	Reserved	Ignored	Ignored
0b0110	Memory read	Yes	Yes
0b0111	Memory write	Yes	Yes
0b1000	Reserved	Ignored	Ignored
0b1001	Reserved	Ignored	Ignored
0b1010	Configuration read	Yes	Yes
0b1011	Configuration write	Yes	Yes
0b1100	Memory read multiple	Yes	Yes
0b1101	Dual address cycle (DAC)	Yes (3)	Yes (3)
0b1110	Memory read line	Yes	Yes
0b1111	Memory write and invalidate	No	Yes

Notes to Table 7–3:

- (1) Refers to the ability of the PCI-Avalon bridge to initiate a PCI transaction with the indicated command.
- (2) Refers to the ability of the PCI-Avalon bridge to accept a PCI transaction with the indicated command.
- (3) This command is not supported in 32-bit PCI bus width applications.

PCI Target Operation

Because it is used with all device types, the PCI target mode is the most basic operational mode for the PCI-Avalon bridge. In PCI target mode, the PCI-Avalon bridge supports the following PCI bus transactions:

- Memory read/write
- Configuration read/write
- IO read/write

PCI configuration read and write operations are automatically handled by the PCI MegaCore function block of the PCI-Avalon bridge. The functions provide access to all PCI configuration registers and behave exactly as described in the PCI Compiler with MegaWizard flow section. Refer to “[Configuration Read Transactions](#)” on page 3–62 and “[Configuration Write Transactions](#)” on page 3–76.



When discussing PCI-Avalon bridge functionality, all PCI memory write transactions are referred to as *write* and all PCI memory read transactions are referred to as *read*. The specific PCI command will be indicated only when the behavior of the bridge is dependent on the actual PCI bus command.

Additionally, *request* and *completion* are used to describe operations handled by the PCI-Avalon bridge. *Request* is used to indicate that the command is being issued for the first time, and *completion* is used to indicate that actual data is being transferred. In a write operation, *request* and *completion* occur within the same PCI transaction. However, in a read operation, *request* and *completion* are usually two different transactions separated by a significant amount of time.

The PCI-Avalon bridge has two distinct data paths: prefetchable and non-prefetchable. Depending on the performance profile and type of BARs used, a transaction is routed to one of the two data paths. If a transaction hits a non-prefetchable BAR it will be handled by the non-prefetchable data path. Additionally, if you select Single-Cycle Transfers Only target performance profile, all PCI memory transactions are routed to the non-prefetchable data path and the prefetchable data path will be removed.

Transactions handled by a non-prefetchable data path have the following key characteristics:

- Are always handled as single data phase transactions
- Read requests will be initially retried and completed as delayed read operations
- The requests will be directed to the Non-prefetchable Avalon-MM master port. This path consists of single address and data registers, and therefore, will have minimal latency. However, the path will not support burst behavior.

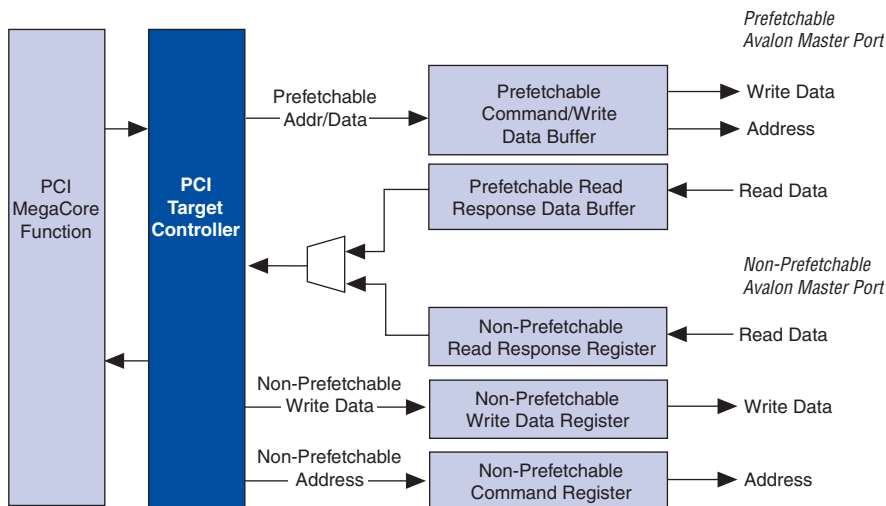
Transactions that hit a prefetchable BAR will be routed to the prefetchable data path and have the following characteristics:

- Burst transactions are supported.

- Read requests will always be initially retried and completed as delayed read operations.
- The requests will be directed to the prefetchable Avalon-MM master port. The data path between the PCI bus and this Avalon-MM port will be optimized to support higher bandwidth that results in higher latency to transition through the required RAM buffers.

Figure 7–7 shows the bridge logic between the PCI target controller and Avalon-MM master ports.

Figure 7–7. PCI Target-to-Avalon-MM Master Block Diagram



Non-Prefetchable Operations

Non-prefetchable operations are defined as either transactions that hit:

- A non-prefetchable BAR
- A prefetchable BAR if the Single-Cycle Transfers Only target performance profile is used

As previously noted, PCI write operations involve only one PCI transaction where the address/command and data is transferred. The read operation involves at least two PCI transactions. In the first PCI transaction (request), the address and data are transferred to the PCI-Avalon bridge, and in the second transaction (completion), the PCI-Avalon bridge transfers the data.

To ensure the lowest possible latency, the PCI-Avalon bridge can handle just one PCI-to-Avalon, non-prefetchable request at a time. A single command register holds the command, address, and byte enables for either the current read or the current write operation. If the register is still busy with the previous operation, no additional read or write requests are accepted and a retry is signaled on the PCI interface.

Non-Prefetchable Write Operations

The non-prefetchable bridge data path handles both the memory write command and the memory write and invalidate command if they hit either:

- A non-prefetchable BAR.
- A prefetchable BAR when the Single-Cycle Transfers Only target performance profile is selected.
- An I/O BAR.

When PCI write requests are claimed from the PCI bus, they are passed to Avalon-MM as write requests. Both PCI memory write and PCI memory write and invalidate PCI bus commands are treated identically inside the non-prefetchable PCI-Avalon bridge logic. The first data phase worth of data is accepted from the PCI bus and written to the PCI-to-Avalon write data register. A target disconnect is signaled as the first data phase is accepted from the PCI bus.

The PCI-to-Avalon address translation circuit is used to compute the appropriate Avalon-MM address. The non-prefetchable Avalon-MM master port will then issue a single-cycle Avalon-MM write transaction to transfer data.

Table 7-4 shows all of the possible termination conditions for non-prefetchable PCI target write operations.

Table 7-4. Non-Prefetchable Write Operation	
Termination Condition	Resulting Action
PCI-to-Avalon non-prefetchable command already in progress	The target controller retries the operation on the PCI bus. Nothing is remembered about the retried PCI write operation. When the PCI write operation is subsequently re-issued, it is treated as a new operation.
Normal master initiated termination of single data phase transaction	Data is accepted and written to the PCI-to-Avalon non-prefetchable data register and then written to Avalon.

Table 7–4. Non-Prefetchable Write Operation

Termination Condition	Resulting Action
Target disconnect if initiator attempts to burst beyond the first data phase	The target controller issues a target disconnect on the PCI bus if the PCI initiator attempts to burst beyond the first data phase. The data is accepted and written to the PCI-to-Avalon non-prefetchable data register and then written to the interconnect.
Target abort	Not applicable when a non-prefetchable master BAR is hit. The target controller will not terminate a PCI write operation with a target abort. When an I/O BAR is hit, target abort is signalled according to the I/O space address decoding.

I/O Write Operations

The non-prefetchable bridge data path handles the I/O write command that hits an I/O BAR.

PCI I/O writes are handled as delayed write operations. When PCI I/O write requests are claimed from the PCI bus, they are passed to Avalon-MM as write requests. The first data phase worth of data is accepted from the PCI bus and written to the PCI-to-Avalon write data register. A target disconnect is signaled as the first data phase is accepted from the PCI bus.

The PCI-to-Avalon address translation circuit is used to compute the appropriate Avalon-MM address. The non-prefetchable Avalon-MM master port will then issue a single-cycle Avalon-MM write transaction to transfer data.

Non-Prefetchable Read Operations

The non-prefetchable data path handles PCI read transactions that hit either:

- A non-prefetchable BAR.
- A prefetchable BAR with the Single-Cycle Transfers Only target performance profile selected.
- An I/O BAR.

Non-prefetchable read operations are handled as delayed read operations. PCI memory read, memory read line, memory read multiple commands and I/O read are treated identically in the non-prefetchable PCI-Avalon bridge logic.

A PCI read operation handled by the non-prefetchable bridge data path has the following sequence of events:

1. In the request phase, the PCI bus issues a read transactions that matches one of the BARs. The PCI-Avalon bridge claims the transaction, stores its address, command and byte enables, and issues a retry. The PCI-Avalon bridge claims transactions only if there are no other transactions pending in the non-prefetchable data path.
2. The PCI-Avalon bridge translates the PCI address to Avalon-MM and passes the transaction to the non-prefetchable Avalon-MM master port, which issues the transaction to the interconnect.
3. The data retrieved from Avalon is stored in a read response register inside the PCI-Avalon bridge. The PCI-Avalon bridge will then wait for the PCI bus to issue the same read transaction.
4. Finally, during the completion phase, the PCI bus issues the same read transaction with exactly the same address, command and byte enables, and then the PCI-Avalon bridge transfers the data, which signals the end of the non-prefetchable read operation.

While the PCI-Avalon is processing the non-prefetchable read, all transactions are retried and not remembered.

Due to the required ordering rules, if there is a pending write transaction in the opposite direction (Avalon-to-PCI), the non-prefetchable read operation's completion phase will be delayed. In other words, if a write operation (flowing in the opposite direction of the current read operation) is in the Avalon bridge first, the PCI-Avalon bridge will not complete the read operation until the write operation completes.

If the non-prefetchable read latency timer expires before the read transaction is complete, the non-prefetchable read transaction's data is discarded. The non-prefetchable read latency counter is set to 32,768 clocks.

Table 7–5 shows all of the termination conditions that are possible for non-prefetchable PCI target read operations.

Table 7–5. Non-Prefetchable Read Operation	
Request/Termination Condition	Resulting Action
PCI-to-Avalon non-prefetchable command register is full. Current command, address and byte enables do not match this register.	The target controller retries the operation on the PCI bus. Nothing is remembered about the retried PCI read operation. When the PCI read operation is subsequently re-issued, it is treated as a new operation.
PCI-to-Avalon non-prefetchable command register is full. Current command, address and byte enables match this register. However, response data from the interconnect is not available.	The target controller retries the operation on the PCI bus.
PCI-to-Avalon non-prefetchable command register is full. Current command, address and byte enables match this register. Response data from the interconnect is valid.	The data is returned to the PCI bus and a disconnect is signaled. The PCI-to-Avalon non-prefetchable command register is made available.
PCI-to-Avalon non-prefetchable command register is available.	Address, command, and byte enables are captured in the PCI-to-Avalon non-prefetchable command register. The read request is forwarded to the interconnect. A retry is signaled on the PCI bus.
PCI-to-Avalon non-prefetchable command register is available. Avalon-to-PCI write operation is already pending.	Address, command, and byte enables are captured in the PCI-to-Avalon non-prefetchable command register. The read request is forwarded to the interconnect. A retry is signaled on the PCI bus. The returned read data is not made available until the previously pending Avalon-to-PCI write operations are complete.
Target abort	Not applicable. The target controller will not terminate a PCI read operation with a target abort.

Prefetchable Operations

If you select either of the burst performance profiles (Burst Transfers with Single Pending Read or Burst Transfers with Multiple Pending Reads), requests that hit prefetchable BARs are handled by the prefetchable data path. At the same time, request that hit non-prefetchable BARs are handled by the non-prefetchable data path as previously described.

The prefetchable data path supports both single-cycle and burst operations and allows multiple writes to be internally pipelined. Additionally, if you select Burst Transfers with Multiple Pending Reads target performance profile, the prefetchable data path will support up to four pending read operations. The Burst Transfers with Single Pending Read target performance profile allows only one pending read at a time.

These features result in higher bandwidth, but introduce higher latency and require more resources.

Prefetchable Write Operations

When PCI write requests that hit prefetchable BARs are claimed from the PCI bus, they are passed to the interconnect as write requests. Both PCI memory write and PCI memory write and invalidate commands are treated identically inside the prefetchable PCI-Avalon bridge logic.

When a PCI memory write is claimed from the PCI interface, the BAR hit information, BAR offset address, and initial data are written to the PCI-to-Avalon command/write data buffer. Memory write transfers are broken into 32-byte boundaries before they are issued on the interconnect. The memory write command is committed to the buffer when either the PCI write command ends on the PCI interface or the burst data reaches a 32-byte boundary. Once the command is committed to the buffer, it becomes visible to the Avalon-MM side of the buffer and the Avalon-MM write operation can begin. Therefore, long PCI memory burst write transactions are broken into 32-byte Avalon-MM transfers. However, depending on the PCI address, the first and/or last resulting Avalon-MM write transaction can be less than 32-bytes. The PCI-Avalon bridge calculates the appropriate Avalon-MM address for all transfers.

If the incoming PCI write specifies the "cacheline wrap code" burst order, the request is target disconnected on the first data phase. The single data phase worth of write data is committed to the PCI-to-Avalon command/write data buffer.

For all data phases of PCI-to-Avalon write requests, the PCI byte enables are passed through to the Avalon-MM byte enables unchanged.

PCI write bursts can be terminated for a number of reasons. The reasons and resulting actions by the PCI target controller are enumerated in [Table 7-6](#).

<i>Table 7-6. Termination of PCI Writes That Hit a Prefetchable BAR as a PCI Target (Part 1 of 2)</i>	
Termination Condition	Resulting Action
Normal master-initiated termination	The current transaction is committed to the PCI-to-Avalon command/write data buffer at its current length.
Some bytes are disabled in the current data phase	The transaction will continue and the byte enables will be passed along to Avalon-MM unchanged.

Table 7–6. Termination of PCI Writes That Hit a Prefetchable BAR as a PCI Target (Part 2 of 2)

Termination Condition	Resulting Action
PCI-to-Avalon command/write data buffer full, no data transferred yet	The target controller retries the operation on the PCI bus. Nothing is remembered about the retried PCI write operation. When the PCI write operation is subsequently re-issued, it is treated as a new operation.
PCI-to-Avalon command/write data buffer full, some data transferred	The target controller issues a disconnect on the PCI bus. The current transaction is committed to the PCI-to-Avalon command/write data buffer at its current length. If and when the PCI write operation is subsequently resumed, it is treated as a new operation.
Prefetchable target burst write with cacheline wrap mode	One data phase worth of data is transferred and the request is disconnected.
Target abort	Not applicable. The target controller will not terminate a PCI write operation with a target abort.

Prefetchable Read Operations

All prefetchable PCI read requests that are claimed are initially retried. The number of retried reads that can be remembered and passed on to the Avalon-MM interface as read requests depends on the performance profile selected in the PCI Compiler wizard. For burst transfers with Single-Cycle Transfers Only performance profile, only one pending read is handled at a time.

For Burst Transfers with Multiple Pending Reads, up to four delayed read transactions can be in progress at the same time. The PCI-Avalon bridge accepts up to four reads and forwards them to the interconnect. The reads are completed in the order requested on the PCI bus.

If additional reads arrive after the maximum number of pending reads are stored in the queue, the additional reads are retried and no information is stored. After one of the pending reads is completed (or discarded due to expiration of a timer), an additional read can be stored in the queue and passed to the interconnect.

For every possible PCI-to-Avalon pending read request, there is a set of registers that store the PCI memory address, command, and byte enables. Therefore, the command can be matched on a subsequent retry. In addition, there is dedicated PCI-to-Avalon read response buffer space and buffer management logic for every possible pending read request.

The read requests are passed through the PCI-to-Avalon command/write data buffer, so that they maintain their ordering with respect to the previous write requests.

Memory read requests that match a prefetchable BAR are forwarded to the interconnect as burst read requests. The size of the burst depends on the PCI command used:

- The cacheline wrap mode reads are treated as Single-Cycle Transfers. Only and are always set to a burst length of one. Therefore, one DWORD is transferred in 32-bit mode, and two DWORDs are transferred in 64-bit mode.
- The PCI memory read and memory read line commands set the burst count to transfer data up to the next 32-byte address boundary. Therefore, the burst count is set from 1-8 in 32-bit mode and 1-4 in 64-bit mode. For example, if the least significant byte of the PCI address is 0x08, the burst count used for 32-bit mode will be 6 and 64-bit mode will be 3.
- The PCI memory read multiple command sets the burst count to transfer data up to the second 32-byte boundary. Thus, the burst count is set from 9-16 in 32-bit mode and 5-8 in 64-bit mode. So, the maximum number of bytes transferred in the PCI memory read multiple command is 64-bytes where the end address must be 32-byte aligned. For example, if the least significant byte of the PCI address is 0x08, the burst count used for 32-bit is 14 and burst count used for 64-bit is 7. Refer to [Table 7-7](#).

Table 7-7. Burst Size for PCI Target Prefetchable Read Requests

PCI Command	Transfers	32-Bit		64-Bit	
		Burst Count	Transfer Size	Burst Count	Transfer Size
Any read that specifies cacheline wrap mode	1	1	1 DWORD	1	2 DWORDS
Memory read and memory read line	As many to reach address that is aligned to 32-bytes.	1-8	1-8 DWORDS	1-4	2-8 DWORDS
Memory read multiple	Transfers up to 64-bytes so that the address reaches the second 32-byte boundary.	9-16	9-16 DWORDS	5-8	10-16 DWORDS

- The PCI memory read command initiates burst transaction to the Avalon bus with the burst value based on the maximum target read burst size (refer to “[Maximum Target Read Burst Size](#)” on page 6–4). For example, the read transaction will return maximum number of 32 DWORDS when the maximum target read burst value is 32.

To optimize overall performance, PCI memory read line and memory read multiple requests (that use linear burst order) always have their burst lengths rounded to a burst boundary. This is done so that subsequent reads will be naturally aligned to burst boundaries.

When a previously remembered PCI read request is claimed again, it is retried until there is at least one data phase worth of data in the response buffer. When there is at least one data phase worth of data in the response buffer, the burst data transfer will begin.

Every attempt is made to keep the read burst transfer going for as long as possible. If additional data has been requested from Avalon-MM, but is not yet available in the response buffer, wait states are inserted on the PCI bus up to the PCI specified maximum target subsequent latency of eight cycles. As soon as the response data is available, it is transferred to the PCI bus. If the data is not available within the eight clock cycles, a target disconnect is issued.

Any time the PCI read operation ends (via either the master or the target device), any prefetched data remaining in the response buffer—or still expected from the interconnect—is discarded. This is done to be consistent with the PCI-SIG specifications.

For memory read requests that hit a prefetchable BAR, all bytes are enabled in the Avalon-MM requests regardless of the actual byte enables signaled by the PCI master.

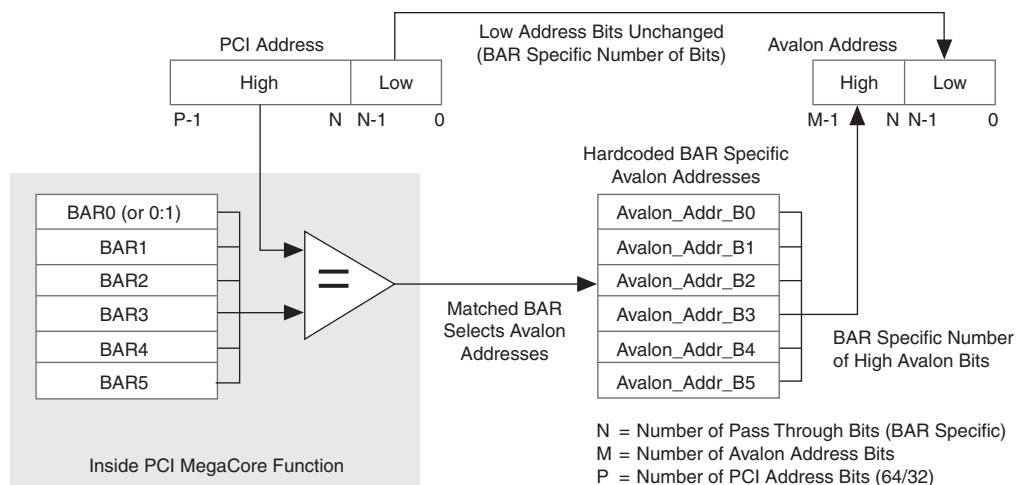
Associated with each pending read response buffer is a timer that determines if and when to discard the data read from the interconnect and free the pending read response buffer. When the initial data for a prefetchable read request is returned from the interconnect, the timer is initialized to 2047 and the timer begins counting down. If the timer reaches 0 before the matching PCI read request is repeated, the data is discarded and the pending read request buffer is freed. Discarding prefetchable read data is not considered an error and no status bit is set to indicate that this has happened.

Table 7–8 lists the reasons for which a burst transfer can be terminated and the resulting actions.

Table 7–8. Termination of Prefetchable Target Burst Reads	
Termination Condition	Resulting Action
Response buffer is empty and no more data is expected from the interconnect	The target controller issues a disconnect and the response buffer is available for re-use.
Response buffer is empty, more data is expected from the interconnect, and less than eight cycles have elapsed since the last data phase.	Wait states are inserted on the PCI bus in an attempt to extend the burst transaction.
Response buffer is empty, more data is expected from the interconnect, and eight cycles have elapsed since the last data phase	The target controller issues a target disconnect. The data is discarded when returned from the interconnect, and the response buffer is available for re-use after all expected data from the interconnect is discarded.
Normal master completion	• Data in the response buffer is discarded. • Data already requested from the interconnect is discarded when returned. • After all expected data from the interconnect is discarded, the response buffer is available for re-use.
Prefetchable target burst read crosses the BAR boundary	One data phase worth of data is read and returned and the request is disconnected. This happens when the burst count exceeds the PCI BAR boundary (Table 7–7 on page 7–24).
Prefetchable target burst read with cacheline wrap mode	One data phase worth of data is transferred and the request is disconnected.
Target abort	Not applicable. The target controller will not terminate a PCI write operation with a target abort.

PCI-to-Avalon Address Translation

Figure 7–8 shows the PCI-to-Avalon address translation. The bits in the PCI address that are used in the BAR matching process are replaced by an Avalon-MM base address that is specific to that BAR. The Avalon-MM base addresses are hardwired from the CB_P2A_AVALON_ADDR_B [0 : 5] parameters for each BAR.

Figure 7–8. PCI-to-Avalon Address Translation

PCI Master Operation

This section describes the PCI master mode operation. Because the PCI Target-Only Peripheral mode is a subset of the PCI Master/Target Peripheral mode, the information in the previous section also applies to the target side of the PCI Master/Target Peripheral mode.

The PCI master mode operation applies to Avalon-to-PCI transactions. The PCI-Avalon bridge automatically accepts Avalon-MM read and write operations targeting the PCI Bus Access Slave port interface and translates them into PCI master transactions. Transaction progress and error conditions are stored in the control/status registers that can be accessed via the control access port.

The Avalon-to-PCI address translation module (refer to [“Avalon-to-PCI Address Translation” on page 7–35](#)) controls whether Avalon-MM read and write requests are issued on PCI as memory, I/O, or configuration space transactions. Except for the command used, accesses to the different spaces operate identically. Burst transactions may even be attempted to configuration space, but this is unusual behavior and target devices may not operate correctly.

You have the option to make the Avalon-to-PCI address translation module either fixed at compile time or dynamically-configured at run time. A fixed Avalon-to-PCI address translation module is very useful for embedded systems with very few PCI devices. If the dynamic address translation table is used, you need to write to it using the `Control Register Access Avalon Slave` port.

The PCI-Avalon bridge uses the burst count to select the best PCI command that provides the best performance on the PCI bus. For example, if the burst count is greater than 1, but the request spans more than one cacheline size (as define by the PCI cacheline size register) the PCI-Avalon bridge issues a PCI memory read multiple command.

Because the interconnect does not support non-posted write operations, the PCI-Avalon bridge cannot report the status of I/O or configuration write operations back to Avalon-MM. So, if you want to emulate a non-posted write behavior, you can either:

- Issue a write immediately followed by a read transaction. When the read returns, you will know that the write transaction is finished.
- Use the current PCI status register bit, `A2P_WRITE_IN_PROGRESS`, to determine if the write transaction is still pending in the bridge.

There are two performance options available in the master mode operation:

- **Burst Transfers with Single Pending Read.** This performance profile initiates both PCI single-cycle and burst transactions, depending on the Avalon-MM burst count. Each PCI delayed read transaction must complete before a new one is initiated. This selection maximizes data throughput, but does not minimize PCI read latencies.
- **Burst Transfers with Multiple Pending Reads.** This performance profile initiates both PCI single-cycle and burst transactions, depending on the Avalon-MM burst count. A maximum of four pending PCI delayed read transactions are allowed. This selection maximizes data throughput and minimizes PCI read latencies.

Avalon-To-PCI Read & Write Operation

The PCI Bus Access Slave port is a burst capable slave that attempts to create PCI bursts that match the bursts requested from the interconnect.

The PCI-Avalon bridge is capable of handling bursts up to 512 bytes with a 32-bit PCI bus and 1024 bytes with a 64-bit PCI. In other words, the maximum supported Avalon-MM burst count is 128.

Bursts from Avalon-MM can be received on any boundary. However, when internal PCI-Avalon bridge bursts cross the Avalon-to-PCI address page boundary, they are broken into two pieces. This is because the address translation can change at that boundary, resulting in a different PCI address needing to be used for the second portion of the burst with a burst count greater than 1.



Avalon-MM burst read requests are treated as if they are going to prefetchable PCI space. Therefore, if the PCI target space is non-prefetchable, you should not use read bursts.

There are several factors that control how Avalon-MM transactions (bursts or single cycle) are translated to PCI transactions. These cases are discussed in [Table 7–9](#). Remember that some optimizations are put in place for situations where 32-bit Avalon-MM masters (for example, the Nios® II processor) talk to 32-bit PCI targets through a 64-bit PCI-Avalon bridge.

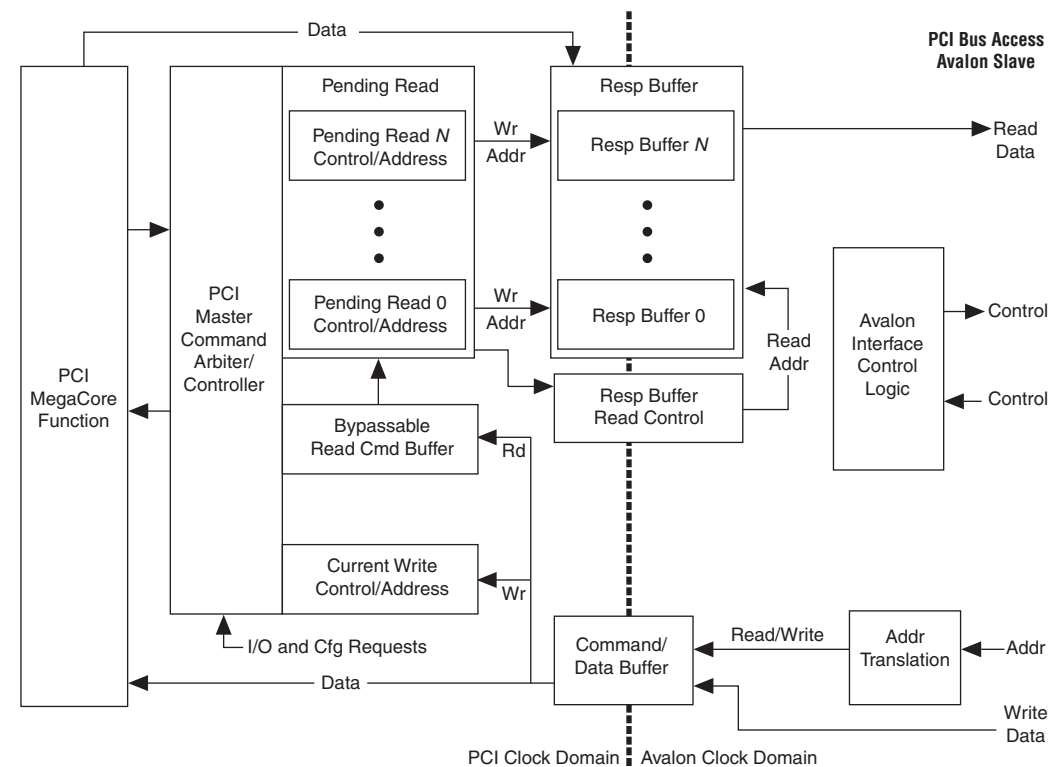
Table 7–9. Translation of Avalon Requests to PCI Requests (Part 1 of 2)

Data Path Width	Avalon Burst Count	Type of Operation	Avalon Byte Enables	Resulting PCI Operation and Byte Enables
32	1	Read or write	Any value	Single data phase read or write, PCI byte enables identical to Avalon byte enables
32	>1	Read	Any value	Attempt to burst on PCI. All data phases will have all PCI bytes enabled.
32	>1	Write	Any value	Attempt to burst on PCI. All data phases will have PCI byte enables identical to the Avalon byte enables.
64	1	Read and write	Upper 4 bytes disabled; lower 4 bytes any value	Only a single 32-bit data phase (<code>req64n</code> not asserted) with the lower 4 byte enables sent to PCI, and lower 32 bits of data if a write.
64	1	Read and write	Upper 4 bytes any value; lower 4 bytes disabled	Only a single 32-bit data phase (<code>req64n</code> not asserted) to the odd DWORD with the upper 4 byte enables sent to PCI (and upper 32 bits of data if a write).
64	1	Read	Bytes enabled in both the upper and lower DWORD	A single 64-bit data phase is attempted (<code>req64n</code> asserted) with the Avalon byte enables sent to PCI. If the target does not assert <code>ack64n</code> and disconnects after a single data phase, the transaction is resumed as a single cycle 32-bit request (<code>req64n</code> not asserted). The PCI byte enables will be the upper 4 byte enables from the original Avalon request.
64	1	Write	Bytes enabled in both the upper and lower DWORD	A 32-bit two data phase burst is attempted (<code>req64n</code> not asserted) with the lower and upper byte enables from Avalon sent in consecutive PCI data phases. If the target disconnects after the first data phase, the request will be resumed as a 32-bit single data phase transfer.

Table 7–9. Translation of Avalon Requests to PCI Requests (Part 2 of 2)

Data Path Width	Avalon Burst Count	Type of Operation	Avalon Byte Enables	Resulting PCI Operation and Byte Enables
64	>1	Read	Any value	Attempt to do a 64 bit burst on PCI (<code>req64n</code> asserted). All data phases will have all PCI byte enables asserted. Note: If the target address space is only 32-bit (<code>ack64n</code> not asserted) and the device disconnects on an odd DWORD boundary, the transaction is resumed as a 32-bit burst (<code>req64n</code> not asserted).
64	>1	Write	Any value	Attempt to do a 64-bit burst on PCI (<code>req64n</code> asserted). All data phases will have PCI byte enables identical to the Avalon byte enables. Note: If the target address space is only 32-bit (<code>ack64n</code> not asserted) and the device disconnects on an odd DWORD boundary (<code>req64n</code> not asserted), a single cycle 32-bit write operation will be issued to get back on an even DWORD boundary. This is followed by an attempt at a 64-bit burst that is converted to a 32-bit burst if the device doesn't acknowledge 64-bit bursts.

Figure 7–9 shows the basic data paths and control structures in the Avalon-to-PCI direction. There is an Avalon-MM slave port that provides access to the PCI bus.

Figure 7–9. Avalon-to-PCI Block Diagram

Avalon-to-PCI Write Requests

For write requests from the interconnect, the write request is pushed on to the PCI bus as a configuration write, I/O write, or memory write. When the Avalon-to-PCI command/write data buffer either has enough data to complete the full burst or 8 data phases (32 bytes on a 32-bit PCI bus or 64 bytes on a 64-bit bus) are exceeded, the PCI master controller will issue the PCI write transaction.

The PCI write is issued to configuration, I/O, or memory space based on the address translation table. Refer to [“Avalon-to-PCI Address Translation”](#) on page 7–35.

For all PCI memory write commands, a linear incrementing burst order is used.

The PCI-Avalon bridge will not combine multiple Avalon-MM writes to consecutive locations into a single PCI write transaction. No attempt is made to byte merge separate Avalon-MM writes that write to separate bytes in the same DWORD or QWORD into a single PCI write operation.

The PCI interface will attempt to burst as long as it can. A PCI write burst can be terminated for various reasons. [Table 7–10](#) describes the resulting action for the PCI master write request termination condition.

Table 7–10. PCI Master Write Request Termination Conditions

Termination condition	Resulting Action
Burst count satisfied	Normal master-initiated termination on PCI bus, command completes, and the master controller proceeds to the next command.
Latency timer expiring during configuration, I/O, or memory write command	Normal master-initiated termination on PCI bus, the continuation of the PCI write is requested from the master controller arbiter.
Avalon-to-PCI command/write data buffer running out of data	Normal master-initiated termination on the PCI bus. Master controller waits for the buffer to reach 8 DWORDs on a 32-bit PCI or 16 DWORDs on a 64-bit PCI, or there is enough data to complete the remaining burst count. Once enough data is available, the continuation of the PCI write is requested from the master controller arbiter.
PCI target disconnect	The continuation of the PCI write is requested from the master controller arbiter.
PCI target retry	
PCI target-abort	The PCI interrupt status register bit, <code>ERR_PCI_WRITE_FAILURE</code> (bit 0), is set to 1. The rest of the write data is read from the buffer and discarded.
PCI master-abort	

Avalon-to-PCI Read Requests

For read requests from the interconnect, the request is pushed on the PCI bus by a configuration read, I/O read, memory read, memory read line, or memory read multiple command. The PCI read is issued to configuration, I/O, or memory space based on the address translation table entry. Refer to [“Avalon-to-PCI Address Translation” on page 7–35](#).

If a memory space read request can be completed in a single data phase, it is issued as a memory read command. If the memory space read request spans more than one data phase but does not cross a cacheline boundary (as defined by the cacheline size register), it is issued as a memory read line command. If the memory space read request crosses a cache line boundary, it is issued as a memory read multiple command.

Single-cycle, 64-bit Avalon-to-PCI read requests that have only the upper or lower 32 bits enabled, need to be issued as single-cycle, 32-bit read requests on the PCI bus.

Avalon-MM requires that read response data be returned in the order requested. Typically, read requests on PCI are initially retried. Usually, a PCI master will issue additional PCI reads after one has been retried—this routine is done so that the PCI targets can start the internal actions for servicing the reads in parallel. However, this leaves the PCI master with little control over the order in which the reads complete. In bridging to Avalon-MM, this can be a particular problem when a PCI read is issued and gets retried, while a second read is issued and data is immediately provided. The bridge needs to hold on to that data until data for the first read is returned.

To solve this problem with the best possible performance, the PCI-Avalon bridge has four Avalon-to-PCI read response buffers for holding pending reads. The multiple response buffers are used when the Burst Transfers with Multiple Pending Reads performance profile is chosen.

There is also a buffer for holding additional Avalon-to-PCI read commands before they are allocated to a pending read buffer and issued on the PCI bus. This buffer allows writes to pass reads before they are allocated an Avalon-to-PCI read response buffer.

When a PCI read request is read from the Avalon-to-PCI bypassable read buffer, it is assigned to the first available Avalon-to-PCI read response buffer. If an Avalon-to-PCI read response buffer is not available, the PCI read request is held in the Avalon-to-PCI bypassable read buffer.

To return the read data to Avalon-MM in the correct order, the Avalon-MM side of the Avalon-to-PCI read response buffers is always read from the buffers in a first-in, first-out order.

No attempt is made to combine multiple Avalon-MM reads to consecutive locations into a single PCI Read burst.

Table 7–11 shows PCI master read request termination conditions.

Table 7–11. PCI Master Read Request Termination Conditions (Part 1 of 2)	
Termination Condition	Resulting Action
Burst count satisfied	Normal master initiated termination on the PCI bus. Master controller proceeds to the next command.
Latency timer expired	Normal master initiated termination on PCI bus. The continuation of the PCI read is made pending as a request the master controller arbiter.

Table 7–11. PCI Master Read Request Termination Conditions (Part 2 of 2)

Termination Condition	Resulting Action
PCI target disconnect	The continuation of the PCI read is requested from the master controller arbiter.
PCI target retry	
PCI target-abort	PCI interrupt status register <code>ERR_PCI_READ_FAILURE</code> (bit 1) is set to 1. Dummy data is returned to complete the Avalon-MM read request. The next operation is then attempted in a normal fashion.
PCI master-abort	

Arbitration Among Pending PCI Master Requests

The transaction arbiter is responsible for managing all pending PCI master requests. To manage the continuous cycle of requests, the transaction arbiter uses priority guidelines to determine which master request to service first. This section discusses how a master request is issued as well as the transaction arbiter's priority guidelines.

A PCI master request can be issued via any of the following:

- Continuation of a previously interrupted command
- A new read from the Avalon-to-PCI bypassable read buffer
- A new write command from the Avalon-to-PCI command/write data buffer

Figure 7–8 on page 7–27 shows that a command can be serviced either from the pending read queue (i.e., for read requests), or directly from the command/data buffer (i.e., for write requests). The transaction arbiter selects one of the "eligible" commands to service. In the Avalon-to-PCI command/data buffer, only the head-of-line command can be eligible.

PCI write and read command eligibility are defined below:

- PCI write commands are "eligible" if either:
 - There are eight data phases of data
 - There is enough data in the Avalon-to-PCI command/write data buffer to satisfy the remaining burst count
- PCI read commands are "eligible" if:
 - They have already been assigned to an Avalon-to-PCI read response buffer
 - They have not been assigned and there is an Avalon-to-PCI read response buffer available



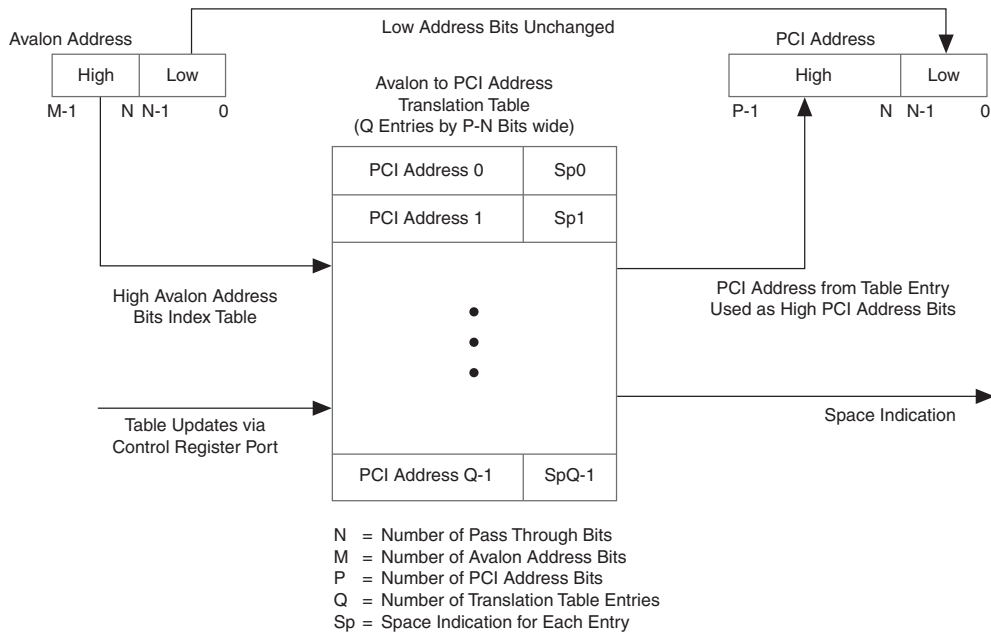
The head-of-line read command in the pending read queue is the one that is issued first and its read data must be returned before the data for all other reads is returned. Because the interconnect is waiting for the head-of-line command data to be returned first, it is given a special priority level.

The transaction arbiter issues eligible commands in the following order:

1. Head-of-line previously retried or disconnected read request that was not the last command issued.
2. Previously disconnected or never issued eligible write request.
3. Not head-of-line previously retried or never issued read request. If there are multiple not head-of-line retried or never issued reads, this priority slot is given to each of them one at a time in a rotating order, so that the head-of-line read request is issued at least once every other command.
4. Head-of-line previously retried read request that was the last command issued.

Avalon-to-PCI Address Translation

Avalon-to-PCI address translation is done through a translation table. Low order Avalon-MM address bits are passed to PCI unchanged; higher order Avalon-MM address bits are used to index into the address translation table. The value found in the table entry is used as the higher order PCI address bits. [Figure 7–10](#) depicts this process.

Figure 7–10. Avalon-to-PCI Address Translation

Address Translation Table Size (refer to “[Avalon Configuration](#)” on [page 6–16](#)) selections determine both the number of entries in the Avalon-to-PCI address translation table, and the number of bits that are passed through the transaction table unchanged.

Each entry in the address translation table also has two address space indication bits, which specify the type of address space being mapped. If the type of address space being mapped is memory, the bits also indicate

whether the resulting PCI address is a 32- or 64-bit address. [Table 7–12](#) shows the address space field's format of the address translation table entries.

Table 7–12. Address Space Bit Encodings

Address Space Indicator (Bits 1:0)	Description
00	Memory space, 32-bit PCI address. Address bits 63:32 of the translation table entries are ignored.
01	Memory space, 64-bit PCI address, dual address cycle (DAC) command will be issued on the PCI bus. Due to PCI Compiler restrictions, this setting is only possible when a 64-bit PCI data path is in use. When a 32-bit PCI data path is in use, the hardware will not let this value be set.
10	I/O space. The address from the translation table process is modified as described in Table 7–13 .
11	Configuration space. The address from the translation table process is treated as a type 1 configuration address and is modified as described in Table 7–13 .

If the space indication bits specify configuration or I/O space, subsequent modifications to the PCI address are performed. Refer to [Table 7–13](#).

Table 7–13. Configuration & I/O Space Address Modifications

Address Space	Modifications Performed
I/O	• PCI address bits 2:0 are set to point to the first enabled byte according to the Avalon byte enables. (Bit 2 only needs to be modified when a 64-bit data path is in use.) • PCI address bits 31:3 are handled normally.
Configuration address bits 23:16 == 0, (Type 0 transaction)	• PCI address bits 1:0 are set to "00" to indicate a type 0 configuration request. • Address bits 10:2 are passed through as normal. • PCI address bits 31:11 are set to be a one-hot encoding of the device number field of the address bits 15:11. For example, if the device number shows 0x00, PCI address bit 11 is set to 1 and bits 31:12 are set to 0. If the device number shows 0x01, PCI address bit 12 is set to 1 and bits 31:13, 11 are set to 0. • Address bits 31:24 are ignored.
Configuration address bits 23:16 > 0 (Type 1 transaction)	• PCI address bits 1:0 are set to "01" to indicate a type 1 configuration request. • Address bits 31:2 are passed through unchanged.

The Avalon-to-PCI address translation table has two configurations (refer to [“Avalon Configuration” on page 6–16](#)):

- Dynamic
- Fixed

If Dynamic Translation Table is specified, translation table entries can be modified, as needed, by the software at run time. If Fixed Translation Table is specified, the translation table entries are fixed at compile time.

The dynamic Avalon-to-PCI address translation table has the following properties:

- The table can be written to via the control register access port
- Entries must be set up before read or write requests are issued to the corresponding Avalon-MM addresses
- The table cannot be preinitialized upon reset

Some applications can use a fixed Avalon-to-PCI address map. Because Host Bridge applications program the BAR registers on the attached PCI devices, they can control which PCI address ranges are used. If a large enough range of the Avalon-MM address space can be set aside to map all of the BARs in a system, then a single fixed Avalon-to-PCI map can be used. Even in PCI Target-Only Peripheral mode or PCI Master/Target Peripheral mode, many embedded applications will know enough about the required PCI addressing to get by with a small number of fixed Avalon-to-PCI translations.

Fixed address translations operate identically to the modifiable scheme described above, except that the translation table effectively becomes a read only memory (ROM). The same parameters control the size of the now fixed translation table.

Ordering of Requests

The PCI-Avalon bridge handles the following types of requests:

- PMW—Posted memory write
- DRR—Delayed read request
- DWR—Delayed write request. DWRs are I/O or configuration write operation requests. The PCI-Avalon bridge does not handle DWRs as delayed writes. As a:
 - PCI master, I/O or configuration writes are generated from posted Avalon-MM writes. If required to verify completion, you must issue a subsequent read request to the same target.

- PCI target, configuration writes are the only requests accepted, which are never delayed. These requests are handled directly by the PCI core.
- DRC—Delayed read completion
- DWC—Delayed write completion. These are never passed through to the core in either direction. Incoming configuration writes are never delayed. Delayed write completion status is not passed back at all.

The following sections describe the special ordering logic and adherence to the PCI-SIG specifications for each direction through the PCI-Avalon bridge.

Ordering of Avalon-to-PCI Operations

Read and write requests in the Avalon-to-PCI direction are handled in a first-in, first-out order through the Avalon-to-PCI command/write data buffer. As read commands are read out of the Avalon-to-PCI command/write data buffer they can be placed in the Avalon-to-PCI bypassable read buffer, which allows them to be passed by writes.

To preserve the producer/consumer ordering model, delayed read completions (for reads handled as a PCI target) cannot pass writes in the PCI-to-Avalon direction. To preserve this ordering relationship, the valid flag for data returned from the interconnect is passed through the Avalon-to-PCI command/write data buffer. Because the buffer is handled in a first-in, first-out order, the read response data valid flag will not be indicated on the PCI side until all previous Avalon-to-PCI write commands are finished.

Figure 7-11 shows the ordering logic used in the Avalon-to-PCI direction.

Figure 7–11. Ordering Logic for Avalon-to-PCI Direction

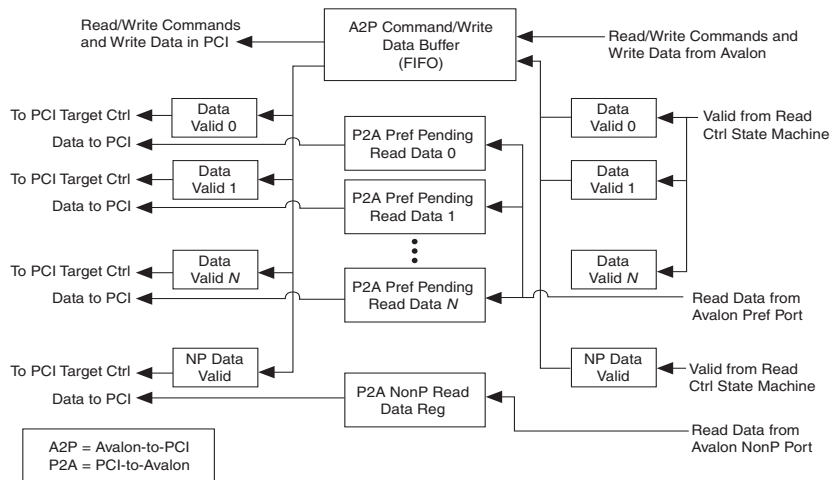


Table 7–14 specifies the ordering rules and behavior of the PCI-Avalon bridge for the Avalon-to-PCI direction. The entries in this table describe whether a type in a row may pass a type in a column. The table uses the following terminology: "No" means a type may not pass another type, "Yes/No" means a type may pass the other type, but does not have to, and "Yes" means that a type must pass another type to avoid deadlocks.

Table 7–14. Summary of Ordering in the Avalon-to-PCI Direction

	PMW		DRR		DWR		DRC		DWC	
	Spec(1)	Impl(2)	Spec	Impl	Spec	Impl	Spec	Impl	Spec	Impl
PMW	No	No(3)	Yes	Yes(6)	Yes	No(8)	Yes	Yes(10)	Yes	N/A(5)
DRR	No	No(3)	Yes/ No	No(7)	Yes/ No	No(3)	Yes/ No	Yes(10)	Yes/ No	N/A(5)
DWR	No	No(3)	Yes/ No	Yes(6)	Yes/ No	No(3)	Yes/ No	Yes(10)	Yes/ No	N/A(5)
DRC	No	No(4)	Yes	Yes(6)	Yes	No(9)	Yes/ No	Yes/ No(11)	Yes/ No	N/A(5)
DWC	Yes/ No	N/A(5)	Yes	N/A(5)	Yes	N/A(5)	Yes/ No	N/A(5)	Yes/ No	N/A(5)

Notes to Table 7–14:

- (1) **Spec** refers to the *PCI Local Bus Specification, Revision 3.0*, published by PCI-SIG.
- (2) **Impl** refers to the implementation of this passing rule in the PCI-Avalon bridge.
- (3) PMWs, DRRs, and DWRs will not pass other PMWs or DWRs since a request at the head of the Avalon-to-PCI Command/Write Data buffer will be handled first before any subsequent requests.
- (4) Ordering logic will make sure that PCI-to-Avalon Read Completion data is not indicated as available on the PCI side until a previous PMW or DWR is issued.
- (5) DWCs are never passed through the PCI-Avalon bridge. The PCI-Avalon bridge can only be the target of a Configuration Write and these are never delayed.
- (6) DRRs can be held pending in the pending read logic or the Avalon-to-PCI Bypassable Read Buffer, allowing them to be passed by PMWs, DWRs, or DRCs.
- (7) Avalon-MM requires that all read data be returned in the order requested. It is possible they can complete on PCI in a different order if there are multiple Avalon-to-PCI Read Response buffers.
- (8) PMWs cannot pass I/O Writes or Configuration Writes (DWRs). However, since the PCI-Avalon bridge does treat I/O Writes or Configuration Writes in a non-posted fashion, the deadlock avoidance required by the PCI specification is not required.
- (9) DRCs cannot pass I/O Writes or Configuration Writes (DWRs). However, since the PCI-Avalon bridge does treat I/O Writes or Configuration Writes in a non-posted fashion, the deadlock avoidance required by the PCI specification is not required.
- (10) DRCs can be held in the PCI-to-Avalon Read Response buffers, allowing them to be passed by PMWs, DRRs or DWRs.
- (11) If multiple PCI-to-Avalon Read Response buffers are implemented, then one DRC can pass another. Otherwise, only one delayed read can be in progress at a time.

Ordering PCI-to-Avalon Operations

For requests that hit a prefetchable BAR, ordering is maintained among the requests by the PCI-to-Avalon command/write data buffer. Both read and write requests pass through this buffer and are handled in a first-in, first-out order.

For requests that hit a non-prefetchable BAR, only one of these requests can be in progress at a time. This request is ordered against any prefetchable requests by passing the non-prefetchable request valid indication through the prefetchable PCI-to-Avalon command/write data buffer.

When a non-prefetchable write request is made valid on the Avalon-MM side, it must interlock prefetchable write and read requests from being passed to the Avalon-MM side until the non-prefetchable request has been accepted by the interconnect. This preserves the ordering of the non-prefetchable write with respect to prefetchable requests that come later.

Read response data for Avalon-to-PCI reads must not be allowed to pass either prefetchable or non-prefetchable writes. To enforce this requirement, the read response data valid indications are passed from the PCI bus to the interconnect through both the prefetchable PCI-to-Avalon command/write data buffer (as sideband data) and the non-prefetchable command processing logic. The read response data is not made valid until the prior commands have been passed to the interconnect.

Figure 7–12 shows the ordering logic used in the PCI-to-Avalon direction. If the prefetchable port is not implemented, the valid indications are propagated through clock synchronization logic (if needed) instead of the prefetchable PCI-to-Avalon command/write data buffer (FIFO).

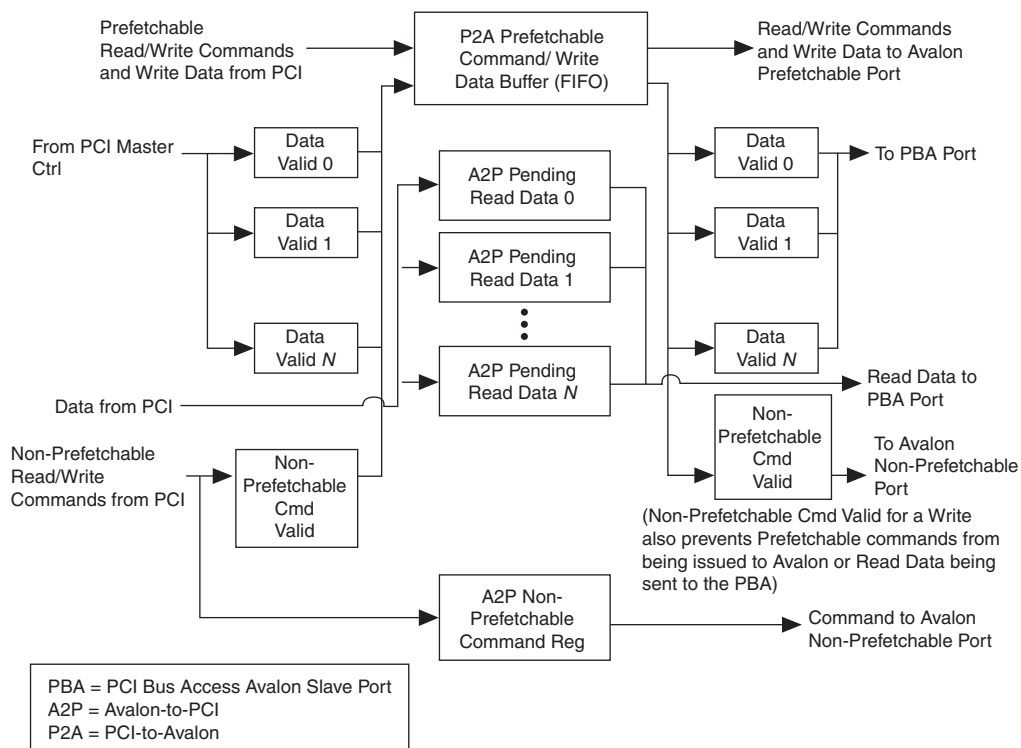
Figure 7–12. Ordering Logic for PCI-to-Avalon Direction

Table 7–15 specifies the ordering rules and behavior for the PCI-to-Avalon direction. The entries in this table describe whether a type in a row may pass a type in a column. The table uses the following terminology: "No" means a type may not pass another type, "Yes/No" means a type may pass the other type, but does not have to, and "Yes" means that a type must pass another type to avoid deadlocks.

Table 7–15. Summary of Ordering in the PCI-to-Avalon Direction

	PMW		DRR		DWR		DRC		DWC	
	Spec(1)	Impl(2)	Spec	Impl	Spec	Impl	Spec	Impl	Spec	Impl
PMW	No	No(3)	Yes	No(5)	Yes	N/A	Yes	Yes(6)	Yes	Yes
DRR	No	No(3)	Yes/ No	No(5)	Yes/ No	N/A	Yes/ No	Yes(6)	Yes/ No	Yes
DWR	No	No(3)	Yes/ No	No(5)	Yes/ No	No(5)	Yes/ No	Yes(6)	Yes/ No	No
DRC	No	No(4)	Yes	No(5)	Yes	N/A	Yes/ No	No(7)	Yes/ No	No
DWC	Yes/ No	No(4)	Yes	No(5)	Yes	No	Yes/ No	No	Yes/ No	No

Notes to Table 7–15:

- (1) **Spec** refers to the *PCI Local Bus Specification, Revision 3.0*, published by PCI-SIG.
- (2) **Impl** refers to the implementation of this passing rule in the PCI-Avalon bridge.
- (3) PMWs and DRRs cannot pass other PMWs in the PCI-to-Avalon Command/Write Data buffer. Ordering logic will prevent PMWs that hit prefetchable BARs and PMWs that hit non-prefetchable BARs from passing each other.
- (4) The ordering logic prevents DRCs from passing PMWs
- (5) Avalon-MM does not provide any mechanism to stop accepting reads separately from writes, so there is no way to make PMWs, DRCs (or other DRRs) pass DRRs. However, since Avalon-MM provides completely separate paths for master and slave transactions, the PCI requirements for this passing do not apply.
- (6) PMWs and DRRs in the PCI-to-Avalon Buffers are allowed to pass DRCs in the Read Response buffers.
- (7) Avalon-MM requires the DRCs (Read Responses) be returned in the order the requests were made. Note, however, that DRCs may actually be returned in a different order than they completed on the PCI bus.

PCI Host-Bridge Operation

You can use the PCI Host-Bridge Device operating mode when the host processor resides on the Avalon-MM side. The PCI Host processor is responsible for configuring all of the PCI devices. In PCI Host-Bridge Device mode, the PCI-to-Avalon bridge operates the same as the PCI Master mode except for the following situations:

- The Control Register Access Avalon Slave port is no longer optional.
- The PCI command register's bus master enable bit is hardwired to 1 to enable PCI master operations at power up.
- You must connect the PCI-Avalon bridge's `idse1` signal to one of the address bus signals to enable self configuration.

Typically, you will need to implement the PCI bus arbitration logic. However, you can still use any of the provided options for the PCI bus arbitration. Refer to [“PCI Bus Arbiter” on page 7–6](#).

Connecting the `idse1` signal to one of the `ad(31:11)` bus lines is an operation that is not automatically performed when using the PCI-Avalon bridge. This can be done in your top level Quartus II design or you can use a resistive coupling on the board. Generating PCI configuration transactions that access the PCI-Avalon bridge's configuration space is accomplished in the same way that all configuration transactions are initiated.

Altera-Provided PCI Bus Arbiter

The Altera-provided arbiter can be enabled from the PCI Compiler wizard. In addition, you can choose to support from two up to eight PCI devices.

The number of external ports for the Altera-provided arbiter is dependent on the number of supported PCI devices. The arbiter's external ports are: `ArbReq_n_i[(N-1):1]` and `ArbGnt_n_o[(N-1):1]`, where N is the number of devices specified. The `ArbReq_n_i[0]` and `ArbGnt_n_o[0]` will automatically be connected to the PCI-Avalon bridge's `reqn` and `gntn` signals respectively.



If you choose to support two devices, the PCI test bench will be automatically configured. If you choose to support more than two devices, additional manual configuration is necessary.

The arbiter is a fair, single level arbiter. Once an `ArbGnt_n_o` signal is asserted in response to the corresponding `ArbReq_n_i`, the grant is maintained at least until one of the following occurs:

- `framen` transitions from deasserted to asserted
- 16 cycles elapse with `framen` deasserted
- The corresponding `ArbReq_n_i` is deasserted

When no `ArbReq_n_i` lines are asserted, `ArbGnt_n_o[0]` will be asserted, parking the bus on the internal PCI-Avalon bridge master.

Interrupts

This section discusses the generation and reception of PCI and Avalon-MM interrupts.

Generation of PCI Interrupts

There are several events that can cause a PCI interrupt. However, for each event there is a specific bit that enables the interrupt. The following events can cause a PCI interrupt:

- Avalon asserts the `IRQ` signal
- Avalon writes to one of the mailbox registers
- An error condition is detected

In any SOPC Builder constructed PCI-Avalon system, either the prefetchable or the non-prefetchable master port has an Avalon-MM interrupt (`IRQ`) input. The non-prefetchable master port always has the `IRQ` interrupt input. However, if the non-prefetchable master is not implemented in a system, the prefetchable port will have the `IRQ` interrupt input.

The Avalon-MM `IRQ` input causes a bit to be set in the PCI interrupt status register. When you need to assert a PCI interrupt, this bit can be enabled.

PCI interrupts can also be generated by writing to the Avalon-to-PCI mailbox registers and having the appropriate enable bit set.

PCI interrupts can also be signaled under a variety of error conditions. Refer to the PCI interrupt status register ([Table 7–18 on page 7–50](#)) and the PCI interrupt enable register ([Table 7–19 on page 7–52](#)) for a complete list of possible interrupt conditions.

Reception of PCI Interrupts

If it is enabled, the PCI-Avalon bridge can signal an interrupt on the interconnect—in response to the assertion or deassertion of the PCI interrupt signal. The PCI-Avalon bridge provides register bits that can signal either a falling- or a rising-edge of the PCI interrupt signal `inta`. You can specify to signal an Avalon-MM interrupt in response to either of the two events or both events.

The Avalon-MM interrupt status register contains two bits that indicate whether a rising- or falling-edge is detected on `intan`. Similarly, the Avalon-MM interrupt enable register has two bits that enable the signaling of an Avalon-MM interrupt on either a rising- or falling-edge of the Avalon-MM interrupt enable register. For a complete description of the Avalon-MM interrupt status register and Avalon-MM interrupt enable register, refer to [Table 7-26 on page 7-57](#) and [Table 7-28 on page 7-60](#).

MSI interrupts can be received by using the PCI-to-Avalon mailbox registers - read/write as the target of the PCI MSI messages. MSI interrupts can also be received by another Avalon-MM slave specifically designed to process them.

Generation of Avalon-MM Interrupts

Avalon-MM interrupts (the `CraIrq_o` signal) can be generated by a variety of error conditions, mailbox writes, or PCI interrupt signals. For a complete list of Avalon-MM interrupts, refer to the Avalon-MM interrupt status register ([Table 7-26 on page 7-57](#)) and the Avalon-MM interrupt enable register ([Table 7-28 on page 7-60](#)).

Control & Status Registers

These registers are accessible from the Control Register Access Avalon Slave port. If you do not enable the Control Register Access Avalon Slave port (refer to [“Avalon Configuration” on page 6-16](#)), none of the control and status registers will be implemented.

The control and status register space is spread over a 16-KByte region, with each 4-KByte sub-region containing a specific set of functions that may be specific to accesses from either:

- PCI processors only
- Avalon processors only
- From both types of processors

Because all accesses come from Avalon-MM (requests from the PCI bus are routed through Avalon-MM), there is no hardware controlling which processors access which regions. However, enforcement via processor software is designed to be straightforward. [Table 7–16](#) describes the four sub-regions.

Table 7–16. Control and Status Register Address Spaces

Address Range	Address Space Usage
0x0000-0x0FFF	Registers typically intended for access by PCI processors only. This includes PCI interrupt enable controls, write access to the PCI-to-Avalon mailbox registers, and read access to Avalon-to-PCI mailbox registers.
0x1000-0x1FFF	Avalon-to-PCI address translation tables. Depending on the system design, these may be accessed by PCI processors, Avalon processors, or both.
0x2000-0x2FFF	Read only registers that reflect various configuration parameters of the implementation. Depending on the system design these may be accessed by PCI processors, Avalon processors, or both.
0x3000-0x3FFF	Registers typically intended for access by Avalon processors only. This includes Avalon-MM interrupt enable controls, I/O and configuration request control registers, write access to the Avalon-to-PCI mailbox registers, and read access to PCI-to-Avalon mailbox registers.

The data returned to a read issued to any undefined address in this range is unpredictable.

Table 7–17 shows the complete map of registers.

Table 7–17. PCI-Avalon Bridge Register Map	
Address Range	Register
0x0040	PCI interrupt status register
0x0050	PCI interrupt enable register
0x0800–0x081F	PCI-to-Avalon mailbox registers – read/write
0x0900–0x091F	Avalon-to-PCI mailbox registers – read only
0x1000–0x1FFF	Avalon-to-PCI address translation table
0x2C00	General configuration parameters – read only
0x2C04	Performance parameters – read only
0x2C08	Avalon-to-PCI address translation parameters – read only
0x3060	Avalon interrupt status register
0x306C	Current PCI status register – read only
0x3070	Avalon interrupt enable register
0x3A00–0x3A1F	Avalon-to-PCI mailbox registers – read/write
0x3B00–0x3B1F	PCI-to-Avalon mailbox registers – read only

The following sections describe the control and status registers in detail. In describing the register's bits, the following nomenclature is used:

- RO: Read only bit. The value of RO bits cannot be modified, but can be read.
- RW: Read and write. The value of RW bits can be read and written.
- RW1C: Read and write "1" to clear. The RW1C bits can be read, but can only be cleared by writing a 1 to the bit location.

PCI Interrupt Status Register

The PCI interrupt status register contains the status of various events in the PCI-Avalon bridge logic and allows PCI interrupts to be signaled if the indicated status bit is set while the corresponding bit in the PCI interrupt enable register is also set. This register is intended to be accessed only by other PCI masters; however, there is nothing in the hardware that prevents other Avalon-MM masters from accessing it.

Table 7–18 describes the PCI interrupt status register, which shows the status of all conditions that can cause the assertion of a PCI interrupt.

Table 7–18. PCI Interrupt Status Register – Address: 0x0040 (Part 1 of 2)

Bit	Name	Access Mode	Description
0	ERR_PCI_WRITE_FAILURE	RW1C	When set to 1 indicates a write to PCI failure (abort or retry threshold exceeded). This bit can also be cleared by writing a '1' to the same bit in the Avalon-MM interrupt status register. This bit will only be implemented if the bridge is either operating in the PCI Master/Target Peripheral or PCI Host-Bridge Device mode.
1	ERR_PCI_READ_FAILURE	RW1C	When set to 1 indicates a read from PCI failure (abort or retry threshold exceeded). This bit can also be cleared by writing a '1' to the same bit in the Avalon-MM interrupt status register. This bit will only be implemented if the bridge is either operating in the PCI Master/Target Peripheral or PCI Host-Bridge Device mode.
2	ERR_NONP_DATA_DISCARD	RW1C	When set to 1 indicates non-prefetchable data read from Avalon-MM was discarded because the PCI read request was not retried before the discard timer expired. Note that this bit can also be cleared by a write of a '1' to the same bit in the Avalon-MM interrupt status register. This bit will only be implemented when the non-prefetchable Avalon-MM master port is implemented.
6:3	Reserved	N/A	
7	AV_IRQ_ASSERTED	RO	Current value of the Avalon-MM interrupt (IRQ) input port to the non-prefetchable Avalon-MM master port (or prefetchable Avalon-MM master port if the non-prefetchable port is not used). 0 – Avalon IRQ is not being signaled. 1 – Avalon IRQ is being signaled.
8	PCI_PERR_REP	RO	Reflects the current value of PCI status register bit 8, PERR reported. This bit can only be cleared through a direct access to the PCI configuration status register.
9	PCI_TABORT_SIG	RO	Reflects the current value of PCI status register bit 11, target abort signaled. This bit can only be cleared through a direct access to the PCI configuration status register. Because the PCI-Avalon bridge does not signal target abort, this bit is never set.

Table 7–18. PCI Interrupt Status Register – Address: 0x0040 (Part 2 of 2)

Bit	Name	Access Mode	Description
10	PCI_TABORT_RCVD	RO	Reflects the current value of PCI status register bit 12, target abort received. This bit can only be cleared through a direct access to the PCI configuration status register.
11	PCI_MABORT_RCVD	RO	Reflects the current value of PCI configuration status register bit 13, master abort received. This bit can only be cleared through a direct access to the PCI configuration status register.
12	PCI_SERR_SIG	RO	Reflects the current value of PCI configuration status register bit 14, system error signaled. This bit can only be cleared through a direct access to the PCI configuration status register.
13	PCI_PERR_DET	RO	Reflects the current value of PCI configuration status register bit 15, PERR detected.
15:14	Reserved	N/A	
16	A2P_MAILBOX_INT0	RW1C	Set to 1 when the A2P_MAILBOX0 is written to.
17	A2P_MAILBOX_INT1	RW1C	Set to 1 when the A2P_MAILBOX1 is written to.
18	A2P_MAILBOX_INT2	RW1C	Set to 1 when the A2P_MAILBOX2 is written to.
19	A2P_MAILBOX_INT3	RW1C	Set to 1 when the A2P_MAILBOX3 is written to.
20	A2P_MAILBOX_INT4	RW1C	Set to 1 when the A2P_MAILBOX4 is written to.
21	A2P_MAILBOX_INT5	RW1C	Set to 1 when the A2P_MAILBOX5 is written to.
22	A2P_MAILBOX_INT6	RW1C	Set to 1 when the A2P_MAILBOX6 is written to.
23	A2P_MAILBOX_INT7	RW1C	Set to 1 when the A2P_MAILBOX7 is written to.
31:24	Reserved	N/A	

PCI Interrupt Enable Register

By setting the corresponding bits in the PCI interrupt enable register, a PCI interrupt can be signaled for any of the conditions registered in the PCI interrupt status register (Table 7–19). The PCI interrupt enable register has one-to-one mapping to the PCI interrupt status register.

Avalon-MM interrupts can also be enabled for all of the conditions in bits 31:0. However, only one of the Avalon-MM or PCI interrupts (not both) should be enabled for any given bit. There is typically a single process in either the PCI or Avalon-MM domain that is responsible for handling the condition reported by the interrupt.

Table 7–19. PCI Interrupt Enable Register – Address: 0x0050

Bit	Name	Access Mode	Description
31:0	One-to-one enable mapping to the PCI interrupt status register bits	RW	When set to 1, indicates that the associated bit in the PCI interrupt status register will cause the PCI interrupt line (<i>intan</i>) to be asserted if not disabled by the PCI command register. Only bits implemented in the PCI interrupt status register are implemented in the enable register. Unimplemented bits cannot be set to 1.

PCI Mailbox Register Access

The PCI bus typically needs write access to a set of PCI-to-Avalon mailbox registers and read-only access to a set of Avalon-to-PCI mailbox registers. [Table 7–1 on page 7–5](#) lists the specific number (1 or 8) of available mailbox registers.

The PCI-to-Avalon mailbox registers are writable at the addresses shown in [Table 7–20](#). Writing to one of these registers causes the corresponding bit in the Avalon-MM interrupt status register to be set to 1.

Table 7–20. PCI-to-Avalon Mailbox Registers – Address Range: 0x0800-0x081F

Address	Name	Access	Description
0x0800	P2A_MAILBOX0	RW	PCI-to-Avalon mailbox 0.
0x0804	P2A_MAILBOX1	RW	PCI-to-Avalon mailbox 1.
0x0808	P2A_MAILBOX2	RW	PCI-to-Avalon mailbox 2.
0x080C	P2A_MAILBOX3	RW	PCI-to-Avalon mailbox 3.
0x0810	P2A_MAILBOX4	RW	PCI-to-Avalon mailbox 4.
0x0814	P2A_MAILBOX5	RW	PCI-to-Avalon mailbox 5.
0x0818	P2A_MAILBOX6	RW	PCI-to-Avalon mailbox 6.
0x081C	P2A_MAILBOX7	RW	PCI-to-Avalon mailbox 7.

The Avalon-to-PCI mailbox registers are readable at the addresses shown in [Table 7-21](#). PCI Hosts use these addresses to read the mailbox information after being signaled by the corresponding bits in the PCI interrupt status register.

Table 7-21. Avalon-to-PCI Mailbox Registers – Address Range: 0x0900-0x091F

Address	Name	Access	Description
0x0900	A2P_MAILBOX0	RO	Avalon-to-PCI mailbox 0.
0x0904	A2P_MAILBOX1	RO	Avalon-to-PCI mailbox 1.
0x0908	A2P_MAILBOX2	RO	Avalon-to-PCI mailbox 2.
0x090C	A2P_MAILBOX3	RO	Avalon-to-PCI mailbox 3.
0x0910	A2P_MAILBOX4	RO	Avalon-to-PCI mailbox 4.
0x0914	A2P_MAILBOX5	RO	Avalon-to-PCI mailbox 5.
0x0918	A2P_MAILBOX6	RO	Avalon-to-PCI mailbox 6.
0x091C	A2P_MAILBOX7	RO	Avalon-to-PCI mailbox 7.

Avalon-to-PCI Address Translation Table

Unless fixed mapping is used, the Avalon-to-PCI address translation table is writable via the Control Register Access Avalon Slave port. In effect, the translation table is writable if the Dynamic Translation Table is selected (refer to [“Avalon Configuration”](#) on page 6-16). The translation table is always readable at the same addresses. ([Table 7-22.](#))

Each entry in the PCI address translation table is always 64 bits (8 bytes) wide, regardless of whether the system supports 64-bit PCI addressing. 64-bit addressing is supported only when the PCI bus width is 64 bits. This ensures that the table address always has the same register addressing regardless of PCI addressing width.

The lower order address bits that are treated as a pass through between Avalon-MM and PCI, and the number of pass-through bits, are defined by the size of page in the address translation table and are always forced to 0 in the hardware table. For example, if the page size is 4 KBytes, the number of pass-through bits is $\log_2(\text{page size}) = \log_2(4 \text{ KBytes}) = 12$.

Table 7–22. Avalon-to-PCI Address Translation Table – Address Range: 0x1000-0x1FFF *Note (1)*

Address	Bit	Name	Access Mode	Description
0x1000	1:0	A2P_ADDR_SPACE0	RW	Address space indication for entry 0. Refer to Table 7–12 on page 7–37 for the definition of these bits.
	31:2	A2P_ADDR_MAP_LO0	RW	Lower bits of Avalon-to-PCI address map entry 0. The pass through bits are not writable and are forced to 0.
0x1004	31:0	A2P_ADDR_MAP_HI0	RW	Upper bits of Avalon-to-PCI address map entry 0. When the PCI bus width is 32 bits, these bits are not writable and are forced to 0.
0x1008	1:0	A2P_ADDR_SPACE1	RW	Address Space indication for entry 1. Refer to Table 7–12 on page 7–37 for the definition of these bits.
	31:2	A2P_ADDR_MAP_LO1	RW	Lower bits of Avalon-to-PCI address map entry 1. Pass through bits are not writable and are forced to 0. This entry is only implemented if the number of pages in the address translation table is greater than 1.
0x100C	31:0	A2P_ADDR_MAP_HI1	RW	Upper bits of Avalon-to-PCI address map entry 1. When the PCI bus width is 32 bits, these bits are not writable and are forced to 0. This entry is only implemented if the number of pages in the address translation table is greater than 1.

Note to Table 7–22:

- (1) The above table entries are repeated for the number of pages you selected in the Avalon configuration tab. If the Number of Address Pages field is set to the maximum of 512, then 0x1FF8 will contain A2P_ADDR_MAP_LO511 and 0x1FFC will contain A2P_ADDR_MAP_HI511. Refer to “[Avalon Configuration](#)” on page 6–16.

Read-Only Configuration Registers

These registers reflect some of the configuration parameters that enable the software to understand the configuration of the PCI-Avalon bridge. Providing this information in these registers allows the software to adapt to the bridge configuration at run time without specifying the same parameter settings to the software at compilation time.

Table 7–23 lists some basic configuration parameters of the bridge.

Table 7–23. General Configuration Parameters – Address 0x2C00			
Bit	Name	Access Mode	Description
6:0	PCI_ADDRESS_WIDTH	RO	Indicates whether 32- or 64-bits of PCI addressing are kept on the Avalon-to-PCI path. Unimplemented upper bits are always forced to 0 in the address phase of a PCI bus transaction.
7	Reserved	RO	Reserved
8	TARGET_ONLY	RO	Indicates that you have selected the PCI Target-Only Peripheral mode (refer to “System Options-1” on page 6–1).
9	HOST_BRIDGE_MODE	RO	Indicates that you have selected the PCI Host-Bridge Device mode (refer to “System Options-1” on page 6–1).
10	PCI_BUS_64	RO	Indicates that you have selected the 64-Bit PCI Bus option (refer to “Value of Multiple Pending Reads” on page 6–6).
11	COMMON_CLOCK_MODE	RO	Indicates that you have selected the Shared PCI and Avalon Clocks option (refer to “Value of Multiple Pending Reads” on page 6–6).
12	IMPL_PREF_PORT	RO	Indicates that the prefetchable Avalon-MM master port is implemented.
13	IMPL_NONP_PORT	RO	Indicates that the non-prefetchable Avalon-MM master port is implemented.
15:14	Reserved	RO	Reserved
19:16	NUM_A2P_MAILBOX	RO	Reflects the number of implemented Avalon-to-PCI mailbox registers.
23:20	NUM_P2A_MAILBOX	RO	Reflects the number of PCI-to-Avalon mailbox registers.
31:24	Reserved	RO	Reserved

Table 7–24 lists some key performance sizing information of the core.

Table 7–24. Performance Parameters – Address 0x2C04			
Bit	Name	Access Mode	Description
15:0	A2P_WRITE_CD_DEPTH	RO	Reflects the depth of the Avalon-to-PCI command and data buffer. The software may not want to issue burst writes (via Avalon DMA or similar) to the bridge that exceed half this value in length. While larger bursts are supported, if the PCI bus is slow or very busy, larger bursts may take a very long time to complete on Avalon, preventing smaller requests from other Avalon-MM masters from being recognized. Backing up those other requests could slow overall performance.
31:16	Reserved	RO	Reserved

Table 7–25 lists the configuration of the Avalon-to-PCI address translation table.

Table 7–25. Avalon-to-PCI Address Translation Parameters – Address 0x2C08			
Bit	Name	Access Mode	Description
0	A2P_ADDR_MAP_IS_FIXED	RO	Indicates that the Fixed Translation Table (refer to “Avalon Configuration” on page 6–16) is selected.
1	A2P_ADDR_MAP_IS_READABLE	RO	Indicates if the Avalon-to-PCI translation table is readable. This bit is always set to 1.
7:2	Reserved	RO	Reserved
13:8	A2P_ADDR_MAP_PASS_THRU_BITS	RO	Indicates the number of pass-through bits (binary encoded).
15:14	Reserved	RO	Reserved
31:16	A2P_ADDR_MAP_NUM_ENTRIES	RO	Indicates the number of pages in the Address Translation Table Size field.

Avalon-MM Interrupt Status Register

The Avalon-MM interrupt status register contains the status of various signals in the PCI-Avalon bridge logic, and it allows Avalon-MM interrupts to be signaled when enabled via the Avalon-MM interrupt enable register. These registers are not intended to be accessed by the PCI-Avalon bridge master ports. However, there is nothing in the hardware that prevents this.

Table 7–26 describes the Avalon-MM interrupt status register bits.

Table 7–26. Avalon Interrupt Status Register – Address 0x3060 (Part 1 of 2)

Bit	Name	Access Mode	Description
0	ERR_PCI_WRITE_FAILURE	RW1C	When set to 1 indicates a write to PCI failure either due to a master or target abort, or because the retry threshold has been exceeded. This bit can also be cleared by writing '1' to the same bit in the PCI interrupt status register. This bit will only be implemented when the bridge is operating in the PCI Master/Target Peripheral modes or the PCI Host-Bridge Device mode.
1	ERR_PCI_READ_FAILURE	RW1C	When set to 1 indicates a read from PCI failure either due to a master or target abort, or because the retry threshold has been exceeded. This bit can also be cleared by writing '1' to the same bit in the PCI interrupt status register. This bit will only be implemented when the bridge is operating in the PCI Master/Target Peripheral modes or the PCI Host-Bridge Device mode.
2	ERR_NONP_DATA_DISCARD	RW1C	When set to 1 indicates that non-prefetchable data read from the interconnect is discarded because the PCI read request was not retried before the parameterized discard timer expired. This bit can also be cleared by writing '1' to the same bit in the PCI interrupt status register. This bit will only be implemented when the non-prefetchable Avalon-MM master port is implemented.
3	MASTER_ENABLE_FALL	RW1C	This bit is set to 1 when the PCI command register master enable bit (command register bit 2) falls from 1 to 0. This bit is set to 0 when '1' is written to it and master enable does not transition in the same cycle as the write. This bit is only implemented when the bridge is operating in the PCI Master/Target Peripheral modes or the PCI Host-Bridge Device mode.
4	MASTER_ENABLE_RISE	RW1C	This bit is set to 1 when the PCI command register master enable bit (command register bit 2) rises from 0 to 1. This bit is set to 0 when '1' is written to it and master enable does not transition in the same cycle as the write. This bit is only implemented when the bridge is operating in the PCI Master/Target Peripheral modes or the PCI Host-Bridge Device mode.
5	Reserved	N/A	
6	INTAN_FALL	RW1C	This bit is set to 1 when the PCI <code>intan</code> signal changes from 1 to 0. This bit is set to 0 when '1' is written to it and <code>intan</code> does not transition in the same cycle as the write. This bit is only implemented when the bridge is operating in the PCI Host-Bridge Device mode.

Table 7–26. Avalon Interrupt Status Register – Address 0x3060 (Part 2 of 2)

Bit	Name	Access Mode	Description
7	INTAN_RISE	RW1C	This bit is set to 1 when the PCI <code>intan</code> signal changes from 0 to 1. This bit is set to 0 when a '1' is written to it and <code>intan</code> does not transition in the same cycle as the write. This bit is only implemented when the bridge is operating in the PCI Host-Bridge Device mode.
8	PCI_PERR_REP	RO	Reflects the current value of PCI status register bit 8, <code>PERR</code> reported. This bit can only be cleared through a direct access to the PCI configuration status register.
9	PCI_TABORT_SIG	RO	Reflects the current value of PCI configuration status register bit 11, target abort signaled. This bit can only be cleared through a direct access to the PCI configuration status register.
10	PCI_TABORT_RCVD	RO	Reflects the current value of PCI configuration status register bit 12, target abort received. This bit can only be cleared through a direct access to the PCI configuration status register.
11	PCI_MABORT_RCVD	RO	Reflects the current value of the PCI configuration status register bit 13, master abort received. This bit can only be cleared through a direct access to the PCI configuration status register.
12	PCI_SERR_SIG	RO	Reflects the current value of PCI configuration status register bit 14, system error signaled. This bit can only be cleared through a direct access to the PCI configuration status register.
13	PCI_PERR_DET	RO	Reflects the current value of PCI configuration status register bit 15, <code>PERR</code> detected.
14:15	Reserved	N/A	
16	P2A_MAILBOX_INT0	RW1C	Set to 1 when the P2A_MAILBOX0 register is written to.
17	P2A_MAILBOX_INT1	RW1C	Set to 1 when the P2A_MAILBOX1 register is written to.
18	P2A_MAILBOX_INT2	RW1C	Set to 1 when the P2A_MAILBOX2 register is written to.
19	P2A_MAILBOX_INT3	RW1C	Set to 1 when the P2A_MAILBOX3 register is written to.
20	P2A_MAILBOX_INT4	RW1C	Set to 1 when the P2A_MAILBOX4 register is written to.
21	P2A_MAILBOX_INT5	RW1C	Set to 1 when the P2A_MAILBOX5 register is written to.
22	P2A_MAILBOX_INT6	RW1C	Set to 1 when the P2A_MAILBOX6 register is written to.
23	P2A_MAILBOX_INT7	RW1C	Set to 1 when the P2A_MAILBOX7 register is written to.
31:24	Reserved	N/A	

Table 7–27 describes the current PCI status register. This register shows the current status of the PCI `rstn` and `int [a:d]n` lines.

Table 7–27. Current PCI Status Register – Address 0x306C

Bit	Name	Access Mode	Description
2:0	Reserved	N/A	
3	MASTER_ENABLE_CURRENT_VALUE	RO	Current value of the PCI command register master enable bit (command register bit 2). 0 – Not enabled to master transactions on the PCI bus. 1 – Enabled to master transactions on the PCI bus. This bit will always be set to 0 when the bridge is operating in the PCI target mode.
4	Reserved	N/A	
5	A2P_WRITE_IN_PROGRESS	RO	0 – There are no Avalon-to-PCI writes pending in the PCI-Avalon bridge module 1 – There is at least one Avalon-to-PCI write pending in the PCI-Avalon bridge module Due to clock synchronization delays, there will be a slight delay between an Avalon-to-PCI write entering the bridge module and this bit being set. The delay could be up to five of the slowest clock cycles. If an application is concerned about the completion of configuration writes on the target bus, the configuration write can be issued by itself and then this bit can be read to confirm when the write is no longer pending. The <code>ERR_PCI_WRITE_FAILURE</code> bit should be checked to determine if there was an error on the write.
6	INTAN_CURRENT_VALUE	RO	Current value of the PCI <code>intan</code> signal. 0 – PCI <code>int A</code> is being signaled. 1 – PCI <code>int A</code> is not being signaled. This bit is only implemented when the bridge is operating in the PCI Host-Bridge Device mode.
31:7	Reserved	N/A	

Avalon-MM Interrupt Enable Register

An Avalon-MM interrupt can be signaled for any of the conditions noted in the Avalon Interrupt Status Register by setting the corresponding bits in the Avalon Interrupt Enable Register (Table 7–28).

PCI interrupts can also be enabled for all of the error conditions in bits 13:8 and 2:0. However, only one of the Avalon-MM or PCI interrupts (not both) should be enabled for any given bit. There is typically a single process in either the PCI or Avalon-MM domain that is responsible for handling the condition reported by the interrupt.

Table 7–28. Avalon Interrupt Enable Register

Avalon Interrupt Enable Register			Address 0x3070
Bit	Name	Access Mode	Description
31:0	One-to-one enable mapping for the bits in the Avalon-MM interrupt status register	RW	When set to 1 indicates the setting of the associated bit in the Avalon-MM interrupt status register will cause the Avalon-MM interrupt line (CraIrq_o) to be asserted. Only bits implemented in the Avalon-MM interrupt status register are implemented in the enable register. Unimplemented bits cannot be set to 1.

Avalon Mailbox Register Access

A processor local to the interconnect (or any processor not on the PCI bus attached to the bridge) typically needs write access to a set of Avalon-to-PCI mailbox registers and read-only access to a set of PCI-to-Avalon mailbox registers. The specific number (1 or 8) of each of these types of mailbox registers available is shown in Table 7–1 on page 7–5.

The Avalon-to-PCI mailbox registers are writable at the addresses shown in [Table 7–29](#). When the Avalon processor writes to one of these registers, the corresponding bit in the PCI interrupt status register is set to 1.

Table 7–29. Avalon-to-PCI Mailbox Registers – Address Range 0x3A00–0x3A1F

Address	Name	Access	Description
0x3A00	A2P_MAILBOX0	RW	Avalon-to-PCI mailbox 0.
0x3A04	A2P_MAILBOX1	RW	Avalon-to-PCI mailbox 1.
0x3A08	A2P_MAILBOX2	RW	Avalon-to-PCI mailbox 2.
0x3A0C	A2P_MAILBOX3	RW	Avalon-to-PCI mailbox 3.
0x3A10	A2P_MAILBOX4	RW	Avalon-to-PCI mailbox 4.
0x3A14	A2P_MAILBOX5	RW	Avalon-to-PCI mailbox 5.
0x3A18	A2P_MAILBOX6	RW	Avalon-to-PCI mailbox 6.
0x3A1C	A2P_MAILBOX7	RW	Avalon-to-PCI mailbox 7.

The PCI-to-Avalon mailbox registers are read only at the addresses shown in [Table 7–30](#). The Avalon processor reads these registers when the corresponding bit in the Avalon-MM interrupt status register is set to 1.

Table 7–30. PCI-to-Avalon Mailbox Registers – Address Range 0x3B00–0x3B1F

Address	Name	Access	Description
0x3B00	P2A_MAILBOX0	RO	PCI-to-Avalon mailbox 0.
0x3B04	P2A_MAILBOX1	RO	PCI-to-Avalon mailbox 1.
0x3B08	P2A_MAILBOX2	RO	PCI-to-Avalon mailbox 2.
0x3B0C	P2A_MAILBOX3	RO	PCI-to-Avalon mailbox 3.
0x3B10	P2A_MAILBOX4	RO	PCI-to-Avalon mailbox 4.
0x3B14	P2A_MAILBOX5	RO	PCI-to-Avalon mailbox 5.
0x3B18	P2A_MAILBOX6	RO	PCI-to-Avalon mailbox 6.
0x3B1C	P2A_MAILBOX7	RO	PCI-to-Avalon mailbox 7.

General Description

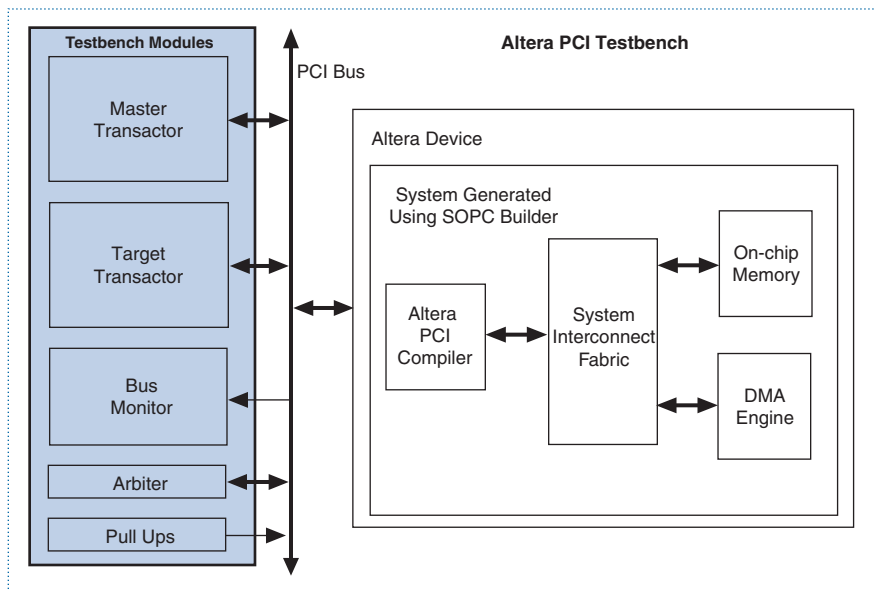
The Altera PCI testbench facilitates the design and verification of systems that implement the Altera PCI-Avalon bridge. The testbench is provided in both VHDL and Verilog HDL. When you build your system with the PCI-Avalon bridge, SOPC Builder automatically integrates the PCI testbench with your system testbench files.

SOPC Builder creates the **pci_sim** directory in your project directory and copies all the PCI testbench files from `<path>/pci_compiler/sopc_flow/testbench/<language>/<core>` into `<project directory>/pci_sim`.



The testbench files must be edited to add the PCI transactions that will be performed on the system. If you regenerate your system, SOPC Builder will not overwrite the testbench files in the **pci_sim** directory. If you want the default testbench files, first delete the **pci_sim** directory and then regenerate your system.

Figure 8–1 shows the block diagram of the PCI testbench. The shaded blocks are provided with the PCI testbench.

Figure 8–1. Altera PCI Testbench Block Diagram

To use the PCI testbench, be sure you have a basic understanding of PCI bus architecture and operations. This document describes the features and applications of the PCI testbench to help you successfully design and verify your design.

Features

The PCI testbench includes the following features:

- Easy to use simulation environment for any standard VHDL or Verilog HDL simulator
- Open source VHDL and Verilog HDL files
- Flexible PCI bus functional model to verify your application that uses any Altera PCI MegaCore function
- Simulates all basic PCI transactions including memory read/write operations, I/O read/write transactions, and configuration read/write transactions
- Simulates all abnormal PCI transaction terminations including target retry, target disconnect, target abort, and master abort
- Simulates PCI bus parking

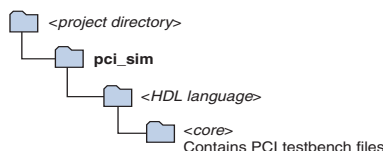
PCI Testbench Files

The Altera PCI testbench is included and installed with the PCI Compiler. [Figure 8–2](#) shows the directory structure of the PCI testbench subdirectory in the project directory.



You will probably modify the PCI testbench directory to simulate your design, so SOPC Builder will not overwrite the `<core>` directory when you regenerate the SOPC Builder system. To revert back to the default PCI testbench settings at regeneration time, just delete the `pci_sim` directory.

Figure 8–2. PCI Testbench Directory Structure



[Table 8–1](#) gives a description of the PCI testbench files provided in the `pci_sim/<HDL language>/<core>` directory. For more information on these files, refer to [“Testbench Specifications” on page 8–4](#).

Table 8–1. Files Contained in the `pci_sim/<HDL language>/<core>` Directory (Part 1 of 2)

File(1)	Description
<code>mstr_tranx</code>	The master transactor defines the procedures (VHDL) or tasks (Verilog HDL) that initiate PCI transactions in the testbench.
<code>mstr_pkg</code>	The master package consists of descriptions of procedures (VHDL) or tasks (Verilog HDL) for master transactor (<code>mstr_tranx</code>) commands.
<code>trgt_tranx</code>	The target transactor simulates the target behavior in the testbench and responds to PCI transactions.
<code>trgt_tranx_mem_init.dat</code>	This file is the memory initialization file for the target transactor.
<code>monitor</code>	This module monitors the PCI transactions on the bus and reports the results.
<code>arbiter</code>	This module contains the PCI bus arbiter.
<code>pull_up</code>	This module provides weak pull-up on the tri-stated signals.

Table 8–1. Files Contained in the pci_sim/<HDL language>/<core> Directory (Part 2 of 2)

File(1)	Description
pci_tb	This top-level file instantiates all the testbench modules.

Note to Table 8–1:

(1) All files are provided in both VHDL and Verilog HDL.

Refer to “Simulation Flow” on page 8–15 for more information on the modified testbench files.

Testbench Specifications

This section describes the modules used by the PCI testbench including master commands, setting and controlling target termination responses, bus parking, and PCI bus speed settings. Refer to Figure 8–1 for a block diagram of the PCI testbench. The Altera PCI testbench has the following modules:

- Master transactor (mstr_tranx)
- Target transactor (trgt_tranx)
- Bus monitor (monitor)
- Clock generator (clk_gen)
- Arbiter (arbiter)
- Pull ups (pull_ups)
- A local reference design

The PCI testbench consists of VHDL and Verilog HDL. If your application requires a feature that is not supported by the PCI testbench, you can modify the source code to add the feature. You can also modify the existing behavior to fit your application needs.

Table 8–2 shows the PCI bus transactions supported by the PCI testbench.

Table 8–2. PCI Testbench PCI Bus Transaction Support (Part 1 of 2)

Transactions	Master Transactor	Target Transactor	Local Master	Local Target
Interrupt acknowledge cycle				
I/O read	✓	✓	✓	✓
I/O write	✓	✓	✓	✓
Memory read	✓	✓	✓	✓
Memory write	✓	✓	✓	✓
Configuration read	✓	✓		

Table 8–2. PCI Testbench PCI Bus Transaction Support (Part 2 of 2)

Transactions	Master Transactor	Target Transactor	Local Master	Local Target
Configuration write	✓	✓		
Memory read multiple		✓		
Dual address cycle				
Memory read line		✓		
Memory write and invalidate		✓		

Table 8–3 shows the testbench's target termination support. The master transactor and the local master respond to the target terminations by terminating the transaction gracefully and releasing the PCI bus.

Table 8–3. PCI Testbench Target Termination Support

Features	Master Transactor	Target Transactor
Target abort	✓	
Target retry	✓	✓
Target disconnect	✓	✓

Master Transactor (mstr_tranx)

The master transactor simulates the master behavior on the PCI bus. It serves as an initiator of PCI transactions for Altera PCI testbench. The master transactor has three main sections:

- PROCEDURES (VHDL) or TASKS (Verilog HDL)
- INITIALIZATION
- USER COMMANDS

PROCEDURES and TASKS Sections

The PROCEDURES (VHDL) and TASKS (Verilog HDL) sections define the events that are executed for the user commands supported by the master transactor. The events written in the PROCEDURES and TASKS sections follow the phases of a standard PCI transaction as defined by the *PCI Local Bus Specification, Revision 3.0*, including:

- Address phase
- Turn-around phase (read transactions)
- Data phases
- Turn-around phase

The master transactor terminates the PCI transactions in the following cases:

- The PCI transaction has successfully transferred all the intended data.
- The PCI target terminates the transaction prematurely with a target retry, disconnect, or abort as defined in the *PCI Local Bus Specification, Revision 3.0*.
- A target does not claim the transaction resulting in a master abort.

The bus monitor informs the master transactor of a successful data transaction or a target termination. Refer to the source code, which shows you how the master transactor uses these termination signals from the bus monitor.

The PCI testbench master transactor PROCEDURES and TASKS sections implement basic PCI transaction functionality. If your application requires different functionality, modify the events to change the behavior of the master transactor. Additionally, you can create new procedures or tasks in the master transactor by using the existing events as an example.

INITIALIZATION Section

This user-defined section defines the parameters and reset length of your PCI bus on power-up. Specifically, the system should reset the bus and write the configuration space of the PCI agents. You can modify the master transactor INITIALIZATION section to match your system requirements by changing the time that the system reset is asserted and by modifying the data written in the configuration space of the PCI agents.

USER COMMANDS Section

The master transactor USER COMMANDS section contains the commands that initiate the PCI transactions you want to run for your tests. The list of events that are executed by these commands is defined in the PROCEDURES and TASKS sections. Customize the USER COMMANDS section to execute the sequence of commands needed to test your design.

Table 8–4 shows the commands that the master transactor supports.

Table 8–4. Supported Master Transactor Commands	
Command Name	Action
<code>cfg_rd</code>	Performs a configuration read
<code>cfg_wr</code>	Performs a configuration write
<code>mem_wr_32</code>	Performs a 32-bit memory write
<code>mem_rd_32</code>	Performs a 32-bit memory read
<code>mem_wr_64</code>	Performs a 64-bit memory write
<code>mem_rd_64</code>	Performs a 64-bit memory read
<code>io_rd</code>	Performs an I/O read
<code>io_wr</code>	Performs an I/O write

cfg_rd

The `cfg_rd` command performs single-cycle PCI configuration read transactions with the address provided in the command argument.

Syntax:	<code>cfg_rd(address)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.

cfg_wr

The `cfg_wr` command performs single-cycle PCI configuration write transactions with the address, data, and byte enable provided in the command arguments.

Syntax:	<code>cfg_wr(address, data, byte_enable)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Transaction data. The data must be in hexadecimal radix.
	<code>byte_enable</code>	Transaction byte enable. The byte enable value must be in hexadecimal radix

mem_wr_32

The `mem_wr_32` command performs a memory write with the address and data provided in the command arguments. This command can perform a single-cycle or burst 32-bit memory write depending on the number of DWORDs provided in the command argument.

- The `mem_wr_32` command performs a single-cycle 32-bit memory write if the DWORD value is 1.
- The `mem_wr_32` command performs a burst-cycle 32-bit memory write if the DWORD value is greater than 1. In a burst transaction, the first data phase uses the data value provided in the command. The subsequent data phases use values incremented sequentially by 1 from the data provided in the command argument.

Syntax:	<code>mem_wr_32(address, data, dword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Data used for the first data phase. Subsequent data phases use a value incremented sequentially by 1. This value must be in hexadecimal radix.
	<code>dword</code>	The number of DWORDs written during the transaction. A value of 1 indicates a single-cycle memory write transaction. A value greater than one indicates a burst transaction. This value must be an integer.

mem_rd_32

The `mem_rd_32` command performs a memory read with the address provided in the command argument. This command can perform single-cycle or burst 32-bit memory read depending on the value of the `dword` argument.

- If the `dword` value is 1, the command performs a single-cycle transaction.
- If the `dword` value is greater than 1, the command performs a burst transaction.

Syntax:	<code>mem_rd_32(address, dword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>dword</code>	The number of DWORDs read during the transaction. A value of one indicates a single-cycle memory read transaction. A value greater than one indicates a burst transaction. This value must be an integer.

mem_wr_64

The `mem_wr_64` command performs a memory write of the data to the address provided in the command. This command can perform single-cycle or burst 64-bit memory write depending on the value of the `qword` argument.

- This command performs a single-cycle 64-bit memory write if the `qword` value is one.
- This command performs a burst-cycle 64-bit memory write if the `qword` value is greater than one. In a burst transaction, the first data phase uses the data value provided in the command. The subsequent data phases use values incremented sequentially by one from the data provided in the command argument.

Syntax:	<code>mem_wr_64 (address, data, qword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Data used for first data phase. Subsequent data phases use a value sequentially incremented by one from this data. This value must be in hexadecimal radix.
	<code>qword</code>	The number <code>QWORDS</code> written in a transaction. A value of one indicates a single-cycle memory write transaction. A value greater than one indicates a burst transaction. This value must be an integer.

mem_rd_64

The `mem_rd_64` command performs memory read transactions with the address provided in the command argument. This command can perform single-cycle or burst 64-bit memory read depending on the value of the `qword` argument.

- If the `qword` value is one, the command performs a single-cycle transaction.
- If the `qword` value is greater than one, the command performs a burst transaction.

Syntax:	<code>mem_rd_32(address, qword)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>qword</code>	The number <code>QWORDS</code> read in the transaction. A one indicates a single-cycle memory read transaction. A value greater than one indicates a burst transaction. This value must be an integer.

io_wr

The `io_wr` command performs a single-cycle memory write transaction with the address and data provided in the command arguments.

Syntax:	<code>io_wr(address, data)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.
	<code>data</code>	Data written during the transaction. This value must be in hexadecimal radix.

io_rd

The `io_rd` command performs single-cycle I/O read transactions with the address provided in the command argument.

Syntax:	<code>io_rd(address)</code>	
Arguments:	<code>address</code>	Transaction address. This value must be in hexadecimal radix.

Target Transactor (trgt_tranx)

The testbench target transactor simulates the behavior of a target agent on the PCI bus. The master transactions initiated by the Altera PCI MegaCore function under test should be addressed to the target transactor. The target transactor operates in 32- or 64-bit mode. The target transactor implements two base address registers BAR0 and BAR1. Refer to [Table 8-5](#).

Table 8-5. Target Transactor Configuration Address Space

Configuration Register	Configuration Address Offset
BAR0	x10
BAR1	x14

The base address registers define the target transactor address space. Refer to [Table 8-6](#).

Table 8-6. Target Transactor Address Space Allocation

Configuration Register	Address Space Type	Block Size	Address Offset
BAR0	Memory Mapped	1 KByte	000-3FF
BAR1	I/O Mapped	16 Bytes	0-F

The memory range reserved by BAR0 is defined by the `address_lines` and `mem_hit_range` settings in the target transactor source code.

The target transactor has a 32-bit register that stores data for I/O transactions. This register is mapped to BAR1 of the configuration address space. Because this is the only register that is mapped to BAR1, any address that is within the BAR1 range results in an `io_hit` action. Refer to the target transactor source code to see how the address is decoded for `io_hit`.



The target transactor ignores byte enables for all memory, I/O, and configuration transactions.

The target transactor `idsel` signal should be connected to one of the PCI address bits in the top-level file of the PCI testbench for configuration transactions to occur on BAR0 and BAR1.

To model different target terminations, use the following three input signals:

- `trgt_tranx_retry`—The target transactor retries the memory transaction if `trgt_tranx_retry` is set to one
- `trgt_tranx_discA`—The target transactor terminates the memory transaction with data if `trgt_tranx_discA` is set to one
- `trgt_tranx_discB`—The target transactor terminates the memory transaction with a disconnect without data if `trgt_tranx_discB` is set to one

The target transactor has two main sections:

- FILE IO
- PROCEDURES (VHDL) and TASKS (Verilog HDL)

FILE IO section

Upon reset, this section initializes the target transactor memory array with the contents of the `trgt_tranx_mem_init.dat` file, which must be in the project's working directory. Each line in the `trgt_tranx_mem_init.dat` file corresponds to a memory location, the first line corresponding to offset "000". The number of lines defined by the `address_lines` parameter in the target transactor source code should be equal to number of lines in the `trgt_tranx_mem_init.dat` file. If the number of lines in `trgt_tranx_mem_init.dat` file is less than the number of lines defined by the `address_lines` parameter, the remaining lines in the memory array are initialized to 0.

PROCEDURES and TASKS sections

The PROCEDURES section (VHDL) and the corresponding TASKS section (Verilog HDL) define the events to be executed for the decoded PCI transaction. These sections are fully documented in the source code. You can modify the procedures or tasks to introduce different variations in the PCI transactions as required by your application. You can also create new procedures or tasks that are not currently implemented in the target transactor by using the existing procedures or tasks as an example.

Bus Monitor (monitor)

The bus monitor displays PCI transactions and information messages to the simulator's console window and in the `log.txt` file when an event occurs on the PCI bus. The bus monitor also sends the PCI transaction status to the master transactor. The bus monitor reports the following messages:

- Target retry
- Target abort
- Target terminated with disconnect-A (target terminated with data)
- Target terminated with disconnect-B (target terminated without data)
- Master abort
- Target not responding

The bus monitor reports the target termination messages depending on the state of the `trdyn`, `devseln`, and `stopn` signals during a transaction. The bus monitor reports a master abort if `devseln` is not asserted within four clock cycles from the start of a PCI transaction. It reports that the target is not responding if `trdyn` is not asserted within 16 clock cycles from the start of the PCI transaction. You can modify the bus monitor to include additional PCI protocol checks as needed by your application.

Arbiter (arbiter)

This module simulates the PCI bus arbiter. The module is a two-port arbiter in which the device connected to port 0 of the arbiter has a higher priority than the device connected to port 1. For example, if device 0 requests the PCI bus while device 1 is performing a PCI transaction, the arbiter removes the grant from device 1 and gives it to device 0. This module allows you to simulate bus parking on devices connected to port 0 by setting the `Park` parameter to true. You change the value of this parameter in the Altera PCI testbench top-level file.

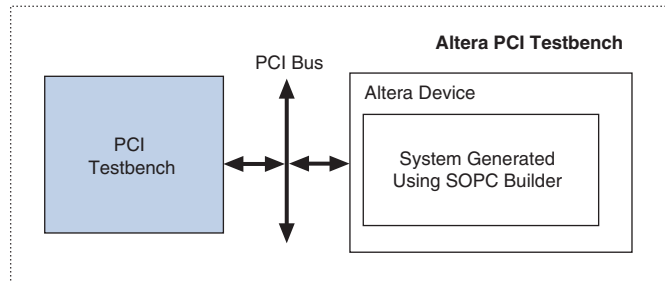
Pull Up (pull_up)

This module simulates the pull up functionality on the PCI signals. The `ad`, `cbe`, `framen`, `irdyn`, `trdyn`, `stopn`, `devseln`, `perrn`, and `serrn` signals of the PCI bus are pulled with a weak high value. This action is necessary to ensure that these signals are never floating or unknown during simulation.

Simulation Flow

This section describes the simulation flow using Altera PCI testbench. [Figure 8–3](#) shows the block diagram of a typical verification environment using the PCI testbench.

Figure 8–3. Typical Verification Environment Using the PCI Testbench



The simulation flow using Altera PCI testbench comprises the following steps.

1. Use SOPC Builder to create your system.



For more information on creating your system using SOPC Builder, refer to [Chapter 5, Getting Started](#).

SOPC Builder creates the **pci_sim** directory in your project directory and copies all the PCI testbench files from

`<path>/pci_compiler/testbench/sopc/<language>/<core>` into `<project directory>/pci_sim`.



The testbench files must be edited to add the PCI transactions that will be performed on the system. If you regenerate your system, SOPC Builder will not overwrite the testbench files in the **pci_sim** directory. If you want the default testbench files, first delete the **pci_sim** directory and then regenerate your system.

2. Set the initialization parameters, which are defined in the master transactor model source code. These parameters control the address space reserved by the target transactor model and other PCI agents on the PCI bus.

Refer to [Figure 8–1](#) for a block diagram of the Master Transactor model instantiated in the PCI testbench.

3. The master transactor defines the procedures (VHDL) or tasks (Verilog HDL) needed to initiate PCI transactions in your testbench. Add the commands that correspond to the transactions you want to implement in your tests to the master transactor model source code. At a minimum, you must add configuration commands to set the BAR for the target transactor model and write the configuration space of the PCI MegaCore function. Additionally, you can add commands to initiate memory or I/O transactions to the PCI MegaCore function.

Refer to [Table 8–4 on page 8–7](#) for more information about the user commands.

4. Compile the files in your simulator, including the testbench modules and the files created by SOPC Builder.
5. Simulate the testbench for the desired time period.

Introduction

Altera provides constraint files, in the form of Tool Command Language (Tcl) scripts, to meet PCI timing requirements in the Quartus II software. The PCI Compiler v11.1 generates a constraint file for your target device family.



For a list of supported device families and the level of support offered for each family, refer to “[About PCI Compiler](#)”.

PCI Constraint Files

PCI constraint files make the following assignments in your Quartus II project:

- PCI pin location assignments
- PCI I/O voltage assignments
- PCI clamp diode assignments
- PCI timing assignments
- Logic option assignments for `clk`, `rstn`, `irdyn`, and `trdyn` pins
- Logic option assignments for the PCI MegaCore function variation
- LogicLock assignments in designs targeting Cyclone devices
- Programmable ground assignments in designs targeting MAX II devices (For more information, refer to “[Simultaneous Switching Noise \(SSN\) Considerations](#)” on page A-2)

When run, the constraint file automatically extracts the PCI MegaCore function, PCI MegaCore function hierarchy, device family, density and package type used in your Quartus II project, and makes PCI assignments for your project based on this information.

To use a PCI constraint file, perform the following steps:

1. Open your project in the Quartus II software.
2. Ensure that you have specified a device family, density and package in your Quartus II project.
3. In the Quartus II software, choose **Tcl Console** (View > Utility Windows menu).

4. Source the constraint file by typing the following in the Quartus II Tcl Console window:

```
source pci_constraints_for_<variation name>.tcl ←
```

5. Add the PCI constraints to your project by typing the following command in the Quartus II Tcl Console window:

```
add_pci_constraints ←
```

Refer to [“Additional Options” on page A-3](#) for the options that can be used with `add_pci_constraints` command.

When you add the PCI constraints file in Step 5 above, the Quartus II software generates a Synopsys Design Constraints (.sdc) file with the file name format, *<variation name>.sdc* file. The Quartus II TimeQuest timing analyzer uses the constraints specified in this file.



For more information on .sdc file or TimeQuest timing analyzer, refer to Quartus II Help.

If you are upgrading from a previous version of PCI compiler, refer to [“Upgrading Assignments from a Previous Version of PCI Compiler” on page A-8](#) for more information.

Simultaneous Switching Noise (SSN) Considerations

PCI MegaCore functions implement a multiplexed high-width address and data (ad) bus. Simultaneous switching noise (SSN) can occur if the bus changes state from 0xFFFF_FFFF to 0x0000_0000, causing ground bounce.



SSN can cause instability to your system. You can avoid instability by assigning the pins for the address and data (ad) bus farther away from the analog power supply pins (VCCA and VCCA_PLL).

To reduce ground bounce, the PCI constraint files for MAX II devices assign several programmable ground pins near the pins used for the PCI ad bus. These pins must be physically connected to ground plane on your board.

These programmable grounds are assigned in the QSF as follows:

```
set_location_assignment <pin location> -to <reserve pin name>
set_instance_assignment -name RESERVE_PIN "AS OUTPUT
DRIVING GROUND" -to <reserve pin name>
```



For more recommendations on reducing SSN in your design, refer to [AN 315: Guidelines for Designing High-Speed FPGA PCBs](#) and [AN 224: High-Speed Board Layout Guidelines](#).

Additional Options

The command syntax description for the `add_pci_constraints` command is the following:

```
add_pci_constraints [-speed "66" | "33"] [-no_compile]
[-no_pinouts] [-pin_prefix <instance name_>]
[-pin_suffix <_instance name>] -help ←
```

These options for the `add_pci_constraints` command are described in the following sections.

-speed

The default value for this option is the maximum speed supported by the targeted device family.

[Table A–1](#) shows the device speed grades required for 33-MHz or 66-MHz operation, and the default speed selected by the PCI constraint file for each supported device family.

Table A–1. Default -speed Value for Supported Device Families			
Device Family	Speed Grade Required for 66 MHz operation	Speed Grade Required for 33 MHz Operation	Default PCI Bus Speed
Arria GX	-6	-6	66 MHz
Arria II GX: (1)			
EP2AGX125DF25 EP2AGX125EF29 EP2AGX125EF35	-3, -4, -5, -6	-3, -4, -5, -6	66 MHz
EP2AGX190EF29 EP2AGX190FF35	-3, -4, -5, -6	-3, -4, -5, -6	66 MHz
EP2AGX260EF29 EP2AGX260FF35	-3, -4, -5, -6	-3, -4, -5, -6	66 MHz
EP2AGX45DF25 EP2AGX45DF29	-3, -4, -5, -6	-3, -4, -5, -6	66 MHz
EP2AGX65DF25 EP2AGX65DF29	-3, -4, -5, -6	-3, -4, -5, -6	66 MHz
EP2AGX95DF25 EP2AGX95EF29 EP2AGX95EF35	-3, -4, -5, -6	-3, -4, -5, -6	66 MHz
Arria II GZ:			

Table A–1. Default -speed Value for Supported Device Families

Device Family	Speed Grade Required for 66 MHz operation	Speed Grade Required for 33 MHz Operation	Default PCI Bus Speed
EP2AGZ225	—	-3, -4	66 MHz
EP2AGZ300	-3	-3, -4	66 MHz
EP2AGZ350FF35 EP2AGZ350FH29 EP2AGZ350HF40	-3	-3, -4	66 MHz
Cyclone	-6, -7	-6, -7, -8	66 MHz
Cyclone II	-7	-7, -8	66 MHz
Cyclone III	-6, -7	-6, -7, -8	66 MHz
Cyclone IV:			
EP4CE6E22 EP4CE6F17 EP4CE6U14	-6, -7, -8	-6, -7, -8, -9	66 MHz
EP4CE10E22 EP4CE10F17 EP4CE10U14	-6, -7, -8	-6, -7, -8, -9	66 MHz
EP4CE15E22 EP4CE15F17 EP4CE15F23 EP4CE15M8 EP4CE15U14	-6, -7	-6, -7, -8, -9	66 MHz
EP4CE22E22 EP4CE22F17 EP4CE22U14	-6, -7	-6, -7, -8, -9	66 MHz
EP4CE30F23 EP4CE30F29	-6, -7	-6, -7, -8, -9	66 MHz
EP4CE40F23 EP4CE40F29 EP4CE40U19	-6, -7	-6, -7, -8, -9	66 MHz
EP4CE55F23 EP4CE55F29 EP4CE55U19	-6, -7	-6, -7, -8, -9	66 MHz
EP4CE75F23 EP4CE75F29 EP4CE75U19	-6, -7	-6, -7, -8, -9	66 MHz
EP4CE115F23 EP4CE115F29	—	-7, -8, -9	66 MHz
EP4CGX15BF14 EP4CGX15BN11	-6, -7	-6, -7, -8	66 MHz
EP4CGX22BF14 EP4CGX22CF19	-6	-6, -7, -8	66 MHz

Table A–1. Default -speed Value for Supported Device Families

Device Family	Speed Grade Required for 66 MHz operation	Speed Grade Required for 33 MHz Operation	Default PCI Bus Speed
EP4CGX30BF14 EP4CGX30CF19 EP4CGX30CF23	-6	-6, -7, -8	66 MHz
EP4CGX50CF23 EP4CGX50DF27	—	-6, -7, -8	66 MHz
EP4CGX75CF23 EP4CGX75DF27	—	-6, -7, -8	66 MHz
EP4CGX110CF23 EP4CGX110DF27 EP4CGX110DF31	—	-7, -8	66 MHz
EP4CGX150CF23 EP4CGX150DF27 EP4CGX150DF31	—	-7, -8	66 MHz
MAX II	-3	-3, -4, -5	66 MHz
Stratix	-5, -6	-5, -6, -7, -8	66 MHz
Stratix GX	-5, -6	-5, -6, -7	66 MHz
Stratix II	-3, -4, -5	-3, -4, -5	66 MHz
Stratix II GX	-3, -4, -5	-3, -4, -5	66 MHz
Stratix III (1)	-2, -3	-2, -3, -4	66 MHz
Stratix IV: (1)			
EP4SE230	-2, -3	-2, -3, -4	66 MHz
EP4SE360F35 EP4SE360H29	-2, -3	-2, -3, -4	66 MHz
EP4SE530	-2	-2, -3, -4	66 MHz
EP4SE820	—	-3, -4	66 MHz
EP4SGX70DF29	-2x, -3	-2x, -3, -4	66 MHz
EP4SGX70HF35	-2, -3	-2, -3, -4	66 MHz
EP4SGX110DF29 EP4SGX110FF35	-2x, -3	-2x, -3, -4	66 MHz
EP4SGX110HF35	-2, -3	-2, -3, -4	
EP4SGX180DF29 EP4SGX180FF35	-2x, -3	-2x, -3, -4	66 MHz
EP4SGX180HF35	-2, -3	-2, -3, -4	66 MHz
EP4SGX180KF40	-2	-2, -3, -4	66 MHz
EP4SGX230DF29 EP4SGX230FF35	-2x, -3	-2x, -3, -4	66 MHz

Table A–1. Default -speed Value for Supported Device Families

Device Family	Speed Grade Required for 66 MHz operation	Speed Grade Required for 33 MHz Operation	Default PCI Bus Speed
EP4SGX230HF35 EP4SGX230KF40	-2, -3	-2, -3, -4	66 MHz
EP4SGX290FF35 EP4SGX290FH29	-2x, -3	-2x, -3, -4	66 MHz
EP4SGX290HF35 EP4SGX290KF40 EP4SGX290KF43	-2, -3	-2, -3, -4	66 MHz
EP4SGX290KF45 EP4SGX290NF45	-2	-2, -3, -4	66 MHz
EP4SGX360FF35 EP4SGX360FH29	-2x, -3	-2x, -3, -4	66 MHz
EP4SGX360HF35 EP4SGX360KF40 EP4SGX360KF43	-2, -3	-2, -3, -4	66 MHz
EP4SGX360NF45	-2	-2, -3, -4	66 MHz
EP4SGX530HH35 EP4SGX530KH40 EP4SGX530NF45 EP4SGX530KF43	-2	-2, -3, -4	66 MHz
EP4S100G2 EP4S100G3 EP4S100G4 EP4S100G5 EP4S40G2 EP4S40G5	-1, -2	-1, -2, -3	66 MHz

Note for Table A–1:

- (1) Pinouts are not provided for this device via the PCI constraint files. As no pinouts are available, the Pin Planner (.ppf) files are not generated. The pinouts are provided during full compilation.

For constraint files that have a default value of 66 MHz, you can override the default value and change it to 33 MHz by typing the following command:

```
add_pci_constraints -speed "33" ↵
```

You cannot override a default speed value of 33 MHz.

-no_compile

By default, the `add_pci_constraints` command performs analysis and synthesis in the Quartus II software and parses the report file to find the hierarchy of the PCI MegaCore function in your design. Use the `-no_compile` option to override analysis and synthesis. This option should only be used if you have performed analysis and synthesis or fully compiled your project prior to using this script.

-no_pinouts

By default, the `add_pci_constraints` command makes Altera-recommended PCI pin location assignments. Use the `-no_pinouts` option if you do not intend to use the recommended PCI pin locations. This option is not recommended and should be used with caution.



The Remote Update core's default pins assignment is not compatible with the pins assigned by PCI Compiler. If you are using a PCI Compiler design with Remote Update in Active Serial Mode, use this option to avoid pins incompatibility.



Remember to assign the pins for address and data (ad) bus farther away from the analog power supply pins (VCCA and VCCA_PLL) to avoid any probable instability caused by SSN.

-pin_prefix

When you specify this option with an instance name, this option appends a prefix consisting of the PCI MegaCore instance name and the leading underscore (`_`) character to the default names of all pins in the MegaCore. You can use this option in your design to uniquely identify and differentiate pins that belong to different MegaCore functions that have common pin names. This option cannot be used with the `-pin_suffix` option. The syntax for this option is as follows:

```
add_pci_constraints -pin_prefix <instance name_>
```

The `<instance name_>` is the prefix that will be attached to all default pins.

-pin_suffix

Appends a suffix consisting of the PCI MegaCore instance name and the leading underscore character (`_`) that you specify to the default names of all pins in the MegaCore. You can use this option in your design to uniquely identify and differentiate pins that belong to different

MegaCore functions that have common pin names. This option cannot be used with the `-pin_prefix` option. The syntax for this option is as follows:

```
add_pci_constraints -pin_suffix <_instance name>
```

The `<_instance name>` is the suffix that will be attached to all default pins.

-help

Use the `-help` option for information on the options used with the `add_pci_constraints` command.

Upgrading Assignments from a Previous Version of PCI Compiler

If your design contains PCI assignments from a previous version of PCI Compiler, attempt to compile your project using your existing PCI assignments. After compilation, check to see if timing requirements are met. If PCI timing requirements are not met, source the constraint file to make new PCI assignments. Refer to [“PCI Constraint Files” on page A-1](#) for more information on using the PCI constraint files.

When the PCI Constraint file is used to upgrade from previous PCI assignments, it performs the following steps:

1. Checks for the default PCI signal names in your QSF.

The PCI Constraint file gives errors if your project uses nondefault PCI pin names. Refer to [“Upgrading PCI Assignments Containing Nondefault PCI Pin Names” on page A-8](#) for more information.

2. Archives the Quartus II project, before making any changes, in **PCI_Archive_<date>_<time>.qar**, where *<date>* is the year, month, and day the script was run; and *<time>* is the hour, minute, and second that the script was run.
3. Removes the previous PCI assignments from your project’s QSF file
4. Makes new PCI assignments for your project.

Upgrading PCI Assignments Containing Nondefault PCI Pin Names

The PCI constraint file uses the default PCI pin names to make PCI assignments. When removing the existing PCI assignments the PCI constraint file checks the PCI pin names against the default PCI pin names. If there is a mismatch between your PCI pin names and the default PCI pin names, do one of the following:

- Manually delete all the existing PCI assignments from your QSF, then use the PCI constraint file as shown in “PCI Constraint Files” on page A-1.
- Update the pin list in the PCI constraint file. The PCI constraint file has a mapping of default PCI pin names to user PCI pin names. Edit the `get_user_pin` name procedure in the PCI constraint file to match the default PCI pin names to your PCI pin names, then use the PCI constraint file as shown in “PCI Constraint Files” on page A-1.

In both of the above methods the PCI constraint files use the default PCI pin names to make new PCI pin assignments in your project. You must manually edit your QSF to change the default PCI pin names to your project-specific pin names.

To edit the PCI constraint file, follow these steps:

1. Open the PCI constraint file in a text editor of your choice. Make sure that any automatic line-wrapping functionality is disabled.
2. Go to the `get_user_pin_name` procedure. This procedure maps the default PCI pin names to user PCI pin names. The first few lines of the procedure are shown below.

```
proc get_user_pin_name { internal_pin_name } {

    #----- Do NOT change -----          Change -----
    array set map_user_pin_name_to_internal_pin_name { ad          ad          }
```

3. Edit the pin names under the "Change" header in the file to match the PCI pin names used in your Quartus II project. In the example line below, the name `ad` is changed to `pci_ad`:

```
#----- Do NOT change -----          Change -----
array set map_user_pin_name_to_internal_pin_name { ad          pci_ad          }
```




Additional Information

Revision History The following table displays the revision history for chapters in this User Guide.

Date	Version	Changes Made
October 2011	11.1	• Updated the User Guide for version 11.1 of PCI Compiler. • Added final support for Cyclone IV E device family. • Updated support level for Cyclone III LS, Cyclone IV GX, HardCopy III and HardCopy IV (E, GX) device family. • Updated speed grade information for Arria II GX, Arria II GZ, Cyclone IV, and Stratix IV device family.
January 2011	10.1	Added speed grade information for Stratix IV device family.
July 2010	10.0	Updated the User Guide for version 10.0 of PCI Compiler.
November 2009	9.1	• Updated the User Guide for version 9.1 of PCI Compiler. • Added support for Cyclone III LS, Cyclone IV GX, HardCopy III and HardCopy IV (E, GX) device family.
March 2009	9.0	Updated the User Guide for version 9.0 of PCI Compiler.
November 2008	8.1	Updated the User Guide for version 8.1 of PCI Compiler.
May 2008	8.0	Updated the User Guide for version 8.0 of PCI Compiler.
October 2007	4.5	• Added information for I/O BAR (SOPC Builder Flow). • Updated the PCI bus command support summary.
May 2007	4.4	Updated the User Guide for version 7.1 of PCI Compiler.
December 2006	4.3	Added preliminary support for Cyclone III device family.
December 2006	4.2	Updated the User Guide for version 6.1 of PCI Compiler.
April 2006	4.1.1	• Updated the User Guide for version 4.1.1 of PCI Compiler. • Updated timing diagrams. • Documented <code>-pin_prefix</code> and <code>-pin_suffix</code> add_pci_constraints command options.
October 2005	4.1.0	Updated the User Guide for version 4.1.0 of the MegaCore functions and PCI compiler.
April 2005	4.0.0	Updated the User Guide for version 4.0.0 of the PCI Compiler. Divided the User Guide into two sections. The first section contains the previous version of the User Guide, which describes the PCI Compiler with the MegaWizard flow. The second section describes the PCI Compiler with the SOPC Builder flow, which is a new feature added in v4.0.0 of the PCI Compiler.
June 2004	3.2.0	Updated the User Guide for version 3.2.0 of the MegaCore functions and PCI compiler.

Date	Version	Changes Made
April 2004	3.1.0	Updated the User Guide for version 3.1.0 of the MegaCore functions and PCI compiler.
February 2004	3.0.0	Updated the user guide for version 3.0.0 of the PCI MegaCore functions and the PCI Compiler. Retitled the user guide <i>PCI Compiler User Guide</i> , and included content from the <i>PCI Compiler Data Sheet</i> and the <i>PCI Testbench User Guide</i> , which are now obsolete.
September 2003	2.4.0	Updated the user guide for version 2.4.0 of the cores and compiler.
February 2003	2.3.0	Updated the user guide for version 2.3.0 of the cores and compiler.
September 2002	2.2.0	Updated the user guide for version 2.2.0 of the cores and compiler.
August 2001	2.0.0	Updated the user guide for version 2.0.0 of the cores and compiler.
February 2001	1.3.0	Updated documentation for version 1.3 of the cores. As of this version, the cores were distributed as part of the PCI compiler.
December 1999	1.0.0	First release of user guide, which described the individual PCI MegaCore functions, including the <code>pci_mt64</code> , <code>pci_mt32</code> , <code>pci_t64</code> , and <code>pci_t32</code> functions.

How to Contact Altera

For the most up-to-date information about Altera products, see the following table.

Contact (1)	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Product literature	Website	www.altera.com/literature
Non-technical support (General) (Software Licensing)	Email	nacomp@altera.com
	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, Save As dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, qdesigns directory, d: drive, and chiptrip.gdf file.
<i>Italic Type with Initial Capital Letters</i>	Indicates document titles. For example, <i>AN 519: Stratix IV Design Guidelines</i> .
<i>Italic type</i>	Indicates variables. For example, $n + 1$. Variable names are enclosed in angle brackets (< >). For example, <file name> and <project name>.pdf file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key, and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
Courier type	Indicates signal, port, register, bit, block, and primitive names. For example, data1, tdi, and input . Active-low signals are denoted by suffix n. For example, resetn. Indicates command line commands and anything that must be typed exactly as it appears. For example, c:\qdesigns\tutorial\chiptrip.gdf. Also, indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword SUBDESIGN), and logic function names (for example, TRI).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● ●	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press Enter.
	The feet direct you to more information about a particular topic.

