

# PCI

---

## MegaCore Function User Guide



101 Innovation Drive  
San Jose, CA 95134  
(408) 544-7000  
<http://www.altera.com>

pci_mt64 version:	2.2.0
pci_mt32 version:	2.2.0
pci_t64 version:	2.2.0
pci_t32 version:	2.2.0
Document Version:	v. 2.1
Document Date:	September 2002

Copyright © 2002 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, mask work 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.



This user guide provides comprehensive information about the Altera® PCI MegaCore® functions included with the PCI compiler.

Table 1 shows the user guide revision history.



See the following sources for more information:

- See “Features...” on page 7 for a complete list of the core features, including new features in this release.
- Refer to the PCI compiler readme files for late-breaking information that is not available in this user guide.

<i>Table 1. User Guide Revision History</i>	
Date	Description
September 2002	Updated the user guide for version 2.2.0 of the cores and compiler.
August 2001	Updated the user guide for version 2.0.0 of the cores and compiler.
February 2001	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	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 Find Information

- The Adobe Acrobat Find feature allows you to search the contents of a PDF file. Click on the binoculars icon in the top toolbar to open the Find dialog box.
- Bookmarks serve as an additional table of contents.
- Thumbnail icons, which provide miniature previews of each page, provide a link to the pages.
- Numerous links, shown in green text, allow you to jump to related information.

## How to Contact Altera

For the most up-to-date information about Altera products, go to the Altera world-wide web site at <http://www.altera.com>.

For additional information about Altera products, consult the sources shown in [Table 2](#).






<i>Table 2. How to Contact Altera</i>		
Information Type	USA & Canada	All Other Locations
Technical support	<a href="http://www.altera.com/mysupport/">http://www.altera.com/mysupport/</a>	<a href="http://www.altera.com/mysupport/">http://www.altera.com/mysupport/</a>
	(800) 800-EPLD (3753) (7:00 a.m. to 5:00 p.m. Pacific Time)	(408) 544-7000 (1) (7:00 a.m. to 5:00 p.m. Pacific Time)
Product literature	<a href="http://www.altera.com">http://www.altera.com</a>	<a href="http://www.altera.com">http://www.altera.com</a>
Altera literature services	<a href="mailto:lit_req@altera.com">lit_req@altera.com</a> (1)	<a href="mailto:lit_req@altera.com">lit_req@altera.com</a> (1)
Non-technical customer service	(800) 767-3753	(408) 544-7000 (7:30 a.m. to 5:30 p.m. Pacific Time)
FTP site	<a href="ftp.altera.com">ftp.altera.com</a>	<a href="ftp.altera.com">ftp.altera.com</a>

**Note:**

(1) You can also contact your local Altera sales office or sales representative.

## Typographic Conventions

The *PCI Compiler MegaCore Function User Guide* uses the typographic conventions shown in [Table 3](#).

Table 3. Conventions	
Visual Cue	Meaning
<b>Bold Type with Initial Capital Letters</b>	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: <b>Save As</b> dialog box.
<b>bold type</b>	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: <b>f<sub>MAX</sub></b> , <b>\quartus</b> directory, <b>d:</b> drive, <b>chiptrip.gdf</b> file.
<b><i>Bold italic type</i></b>	Book titles are shown in bold italic type with initial capital letters. Example: <b><i>1999 Device Data Book</i></b> .
<i>Italic Type with Initial Capital Letters</i>	Document titles are shown in italic type with initial capital letters. Example: <i>AN 75 (High-Speed Board Design)</i> .
<i>Italic type</i>	Internal timing parameters and variables are shown in italic type. Examples: <i>t<sub>PIA</sub></i> , <i>n + 1</i> . Variable names are enclosed in angle brackets (< >) and shown in italic type. Example: <i>&lt;file name&gt;</i> , <i>&lt;project name&gt;.pof</i> file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of Quartus® II Help topics are shown in quotation marks. Example: "Configuring a FLEX 10K or FLEX 8000 Device with the BitBlaster™ Download Cable."
Courier type	Signal and port names are shown in lowercase Courier type. Examples: <code>data1</code> , <code>tdi</code> , <code>input</code> . Active-low signals are denoted by suffix <code>n</code> , e.g., <code>resetn</code> .  Anything that must be typed exactly as it appears is shown in Courier type. For example: <code>c:\max2work\tutorial\chiptrip.gdf</code> . Also, sections of an actual file, such as a Report File, references to parts of files (e.g., the AHDL keyword <code>SUBDESIGN</code> ), as well as logic function names (e.g., <code>TRI</code> ) are shown in Courier.
1., 2., 3., and a., b., c.,...	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
	Bullets are used in a list of items when the sequence of the items is not important.
	The checkmark indicates a procedure that consists of one step only.
	The hand points to information that requires special attention.
	The angled arrow indicates you should press the Enter key.
	The feet direct you to more information on a particular topic.



<b>About this User Guide .....</b>	<b>iii</b>
How to Find Information .....	iii
How to Contact Altera .....	iv
Typographic Conventions .....	v
<b>About this Core .....</b>	<b>1</b>
Introduction .....	1
General Description .....	1
Features... ..	7
...and More Features .....	8
<b>Getting Started .....</b>	<b>9</b>
PCI MegaCore Directory Structure .....	10
Altera PCI MegaCore Function Design Flow .....	11
Obtain the PCI MegaCore Functions .....	11
Instantiate a PCI MegaCore Function in Your Design .....	11
Synthesize .....	11
Simulate .....	12
Obtain PCI Constraint File and Analyze Timing .....	12
License MegaCore Function .....	13
Configure a Device .....	13
Design Walk-Through .....	13
Generating a Project-Specific Instance of the pci_mt64, pci_t64, pci_mt32, or pci_t32 Function .....	13
Generating Project-Specific Constraint Files to Achieve PCI Timing Requirements ..	16
Compilation, Functional Simulation & Timing Analysis in the Quartus II Software .	19
Compilation .....	19
Timing Analysis .....	20
Functional Simulation .....	20
<b>MegaCore Overview .....</b>	<b>21</b>
Compliance Summary .....	21
PCI Bus Signals .....	22
Parameterized Configuration Register Signals .....	26
Local Address, Data, Command and Byte Enable Signals .....	27
Target Local-Side Signals .....	30
Master Local-Side Signals .....	33
MegaWizard Plug-In .....	37

Parameters .....	37
Application Speed Capability .....	37
Read-Only PCI Configuration Registers .....	38
PCI Base Address Registers (BARs) .....	39
Advanced Features in the pci_mt64, pci_mt32, pci_t64, and pci_t32 MegaCore Functions .....	41
Optional Registers .....	41
Optional Interrupt Capabilities .....	42
Optional Master Features .....	42
64-Bit PCI Options .....	43
Functional Description .....	44
Target Device Signals & Signal Assertion .....	44
Master Device Signals & Signal Assertion .....	47
<b>Specifications .....</b>	<b>49</b>
PCI Bus Commands .....	49
Configuration Registers .....	50
Vendor ID Register .....	53
Device ID Register .....	53
Command Register .....	53
Status Register .....	54
Revision ID Register .....	56
Class Code Register .....	56
Cache Line Size Register .....	56
Latency Timer Register .....	57
Header Type Register .....	57
Base Address Registers .....	58
CardBus CIS Pointer Register .....	61
Subsystem Vendor ID Register .....	61
Subsystem ID Register .....	62
Expansion ROM Base Address Register .....	62
Capabilities Pointer .....	63
Interrupt Line Register .....	63
Interrupt Pin Register .....	64
Minimum Grant Register .....	64
Maximum Latency Register .....	65
Target Mode Operation .....	65
64-Bit Target Read Transactions .....	68
64-Bit Single-Cycle Target Read Transaction .....	69
64-Bit Memory Burst Read Transaction .....	72
32-Bit Target Read Transactions .....	76
32-Bit Memory Read Transactions .....	77
I/O Read Transaction .....	80
Configuration Read Transaction .....	82



64-Bit Target Write Transactions .....	83
64-Bit Single-Cycle Target Write Transaction .....	83
64-Bit Target Burst Write Transaction .....	86
32-Bit Target Write Transactions .....	90
32-Bit Memory Write Transaction .....	90
I/O Write Transaction .....	93
Configuration Write Transaction .....	95
Target Transaction Terminations .....	96
Retry .....	96
Disconnect .....	98
Target Abort .....	103
Master Mode Operation .....	105
PCI Bus Parking .....	108
Design Consideration .....	108
64-Bit Master Read Transactions .....	108
64-Bit Master Burst Memory Read Transaction with Local-Side Wait State .....	113
64-Bit Master Burst Memory Read Transaction with PCI Wait State .....	115
64-Bit Master Single-Cycle Memory Read Transaction .....	117
32-Bit Master Read Transactions .....	119
32-Bit PCI & 64-Bit Local-Side Master Burst Memory Read Transaction .....	119
32-Bit PCI & 32-Bit Local-Side Master Burst Memory Read Transaction .....	121
32-Bit PCI & 32-Bit Local Side Single-Cycle Memory Read Transaction .....	123
64-Bit Master Write Transactions .....	125
64-Bit Master Zero Wait State Burst Memory Write Transaction .....	126
64-Bit Master Burst Memory Write Transaction with Local Wait State .....	131
64-Bit Master Burst Memory Write Transaction with PCI Wait State .....	133
32-Bit Master Write Transactions .....	135
32-Bit PCI & 64-Bit Local-Side Master Burst Memory Write Transaction .....	135
32-Bit PCI & 32-Bit Local-Side Master Burst Memory Write Transaction .....	137
32-Bit PCI & 32-Bit Local-Side Single-Cycle Memory Write Transaction .....	139
Abnormal Master Transaction Termination .....	141
Latency Timer Expires .....	141
Retry .....	141
Disconnect Without Data .....	142
Disconnect with Data .....	142
Target Abort .....	142
Master Abort .....	142
Host Bridge Operation .....	142
Using the PCI MegaCore Function as a Host Bridge .....	143
PCI Configuration Read Transaction from the pci_mt64 Local Master Device to the Internal Configuration Space .....	144
PCI Configuration Write Transaction from the pci_mt64 Local Master Device to the Internal Configuration Space .....	146
Implementing Internal Bus Arbitration Logic .....	148
64-Bit Addressing, Dual Address Cycle (DAC) .....	148
Target Mode Operation .....	148

64-Bit Address, 64-Bit Data Single-Cycle Target Read Transaction .....	149
Master Mode Operation .....	151
64-Bit Address, 64-Bit Data Master Burst Memory Read Transaction .....	151
<b>Appendix A: Tips for 66-MHz PCI Designs .....</b>	<b>153</b>
Pipelining the Local-Side Design .....	153
Designing to the PCI Function Local Side .....	153
Design Examples .....	153
<b>Appendix B: Using PCI Constraint Files .....</b>	<b>155</b>
PCI Constraint File Contents .....	155
Generate a Constraint File for Your Project .....	155
PCI System Speed .....	156
Input Constraint File .....	157
Output Constraint File .....	157
Project Name .....	157
How to Use PCI Tcl Scripts in the Quartus II Software .....	158
<b>Appendix C: 64-Bit Options for the pci_mt64 and pci_t64 MegaCore Functions .....</b>	<b>159</b>
Introduction .....	159
64-Bit Only Devices Option .....	159
Add Internal Data Steering Logic for 32/64-Bit Systems Option .....	162
<b>Appendix D: PCI MegaCore Function Parameters .....</b>	<b>167</b>

## Introduction

Altera® peripheral component interconnect (PCI) MegaCore® functions provide solutions for interfacing your system to a 32-bit or 64-bit PCI bus. A PCI bus can be used to implement peripheral devices such as network adapters, graphic accelerator boards, and embedded control modules.

The Altera PCI functions—`pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32`—enhance your productivity by allowing you to focus your efforts on the custom logic surrounding the PCI interface. The functions are optimized for Altera APEX™, ACEX™, FLEX®, Excalibur®, Stratix®, and Cyclone® FPGA devices and are fully tested to meet the requirements of the PCI Special Interest Group (SIG) **PCI Local Bus Specification, Revision 2.2** and **Compliance Checklist, Revision 2.2**. You can test-drive Altera PCI MegaCore functions using the OpenCore™ feature to compile and simulate the functions within your custom logic. When you are ready to license a function, contact your local Altera sales representative.

The PCI compiler contains everything you need to use Altera PCI solutions including the MegaCore functions—which can be instantiated with a wizard-driven interface—behavioral models, testbench, and reference designs. You can download the PCI compiler from the IP MegaStore area on the Altera web site at <http://www.altera.com/IPmegastore>. Refer to the *PCI Compiler Data Sheet* for more information on the PCI compiler contents and how to obtain and install the compiler. The **PCI MegaCore Function User Guide** provides information on how to get started using the PCI MegaCore functions and describes the technical specifications of the functions.



You can use the PCI compiler to test-drive several PCI MegaCore functions using the OpenCore feature; however, each PCI MegaCore function must be licensed separately.

## General Description

The PCI MegaCore functions covered in this document are hardware-tested, high-performance, flexible implementations of PCI interfaces. These functions handle the complex PCI protocol and stringent timing requirements internally, and their back-end interface is designed for easy integration. Therefore, you can focus their engineering efforts on value-added custom development, significantly reducing time-to-market.

Optimized for Altera® devices, the PCI functions support configuration, I/O, and memory transactions. With the high density of Altera's devices, you have ample resources for custom local logic after implementing 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 run at either 33-MHz or 66-MHz PCI bus clock speeds; they thus achieve 132-MBps throughput in a 32-bit, 33-MHz PCI bus system and up to 528-MBps throughput in a 64-bit, 66-MHz PCI bus system.

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

To ensure timing and protocol compliance, PCI MegaCore functions have been vigorously hardware tested. See [“Compliance Summary” on page 21](#) for more information on the hardware tests performed.

As parameterized functions, `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` have configuration registers that can be modified upon instantiation. These features provide scalability, adaptability, and efficient silicon implementation. As a result, the same MegaCore functions can be used in multiple PCI projects with different requirements. For example, these functions offer up to six BARs for multiple local-side devices. However, some applications require only one contiguous memory range. PCI designers can choose to instantiate only one BAR, which reduces logic cell consumption. After you define the parameter values, the Quartus II software automatically and efficiently modifies the design and implements the logic.

This user guide should be used in conjunction with the latest PCI specification, published by the PCI Special Interest Group (SIG). Users should be fairly familiar with the PCI standard before using these functions. [Figures 1](#) through [4](#) show the block diagrams for `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32`, respectively. Refer to these figures for signal names and directions for the individual functions.

The functions consist of several blocks:

- *PCI bus configuration register space.* This block implements all of the configuration registers required by the **PCI Local Bus Specification, Revision 2.2**. You can set these registers to your system requirements by setting the parameters provided.

- *Parity checking and generation.* This block is responsible for parity checking and generation. It also asserts parity error signals and required status register bits.
- *Target interface control logic.* This block controls the operation of the corresponding MegaCore function on the PCI bus in target mode.
- *Master interface control logic.* This block controls the PCI bus operation of the corresponding PCI MegaCore function in master mode. This block is only implemented in the `pci_mt64` and `pci_mt32` functions.
- *Local target control.* This block controls the local side interface operation in target mode.
- *Local master control.* This block controls the local side interface operation in master mode. This block is implemented only in the `pci_mt64` and `pci_mt32` functions.
- *Local address/data/command/byte enables.* This block multiplexes and registers all the address, data, command, and byte enable signals to the local side interface.

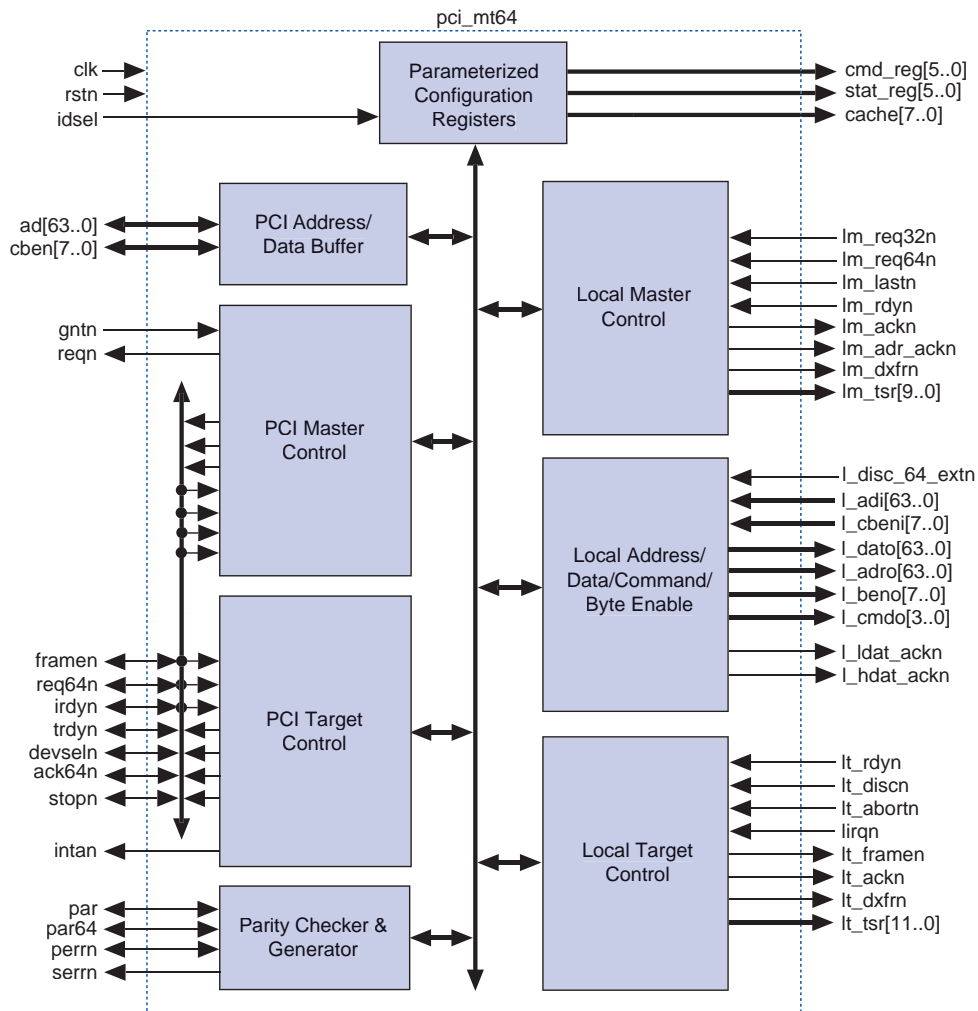
Figure 1. *pci\_mt64* Functional Block Diagram

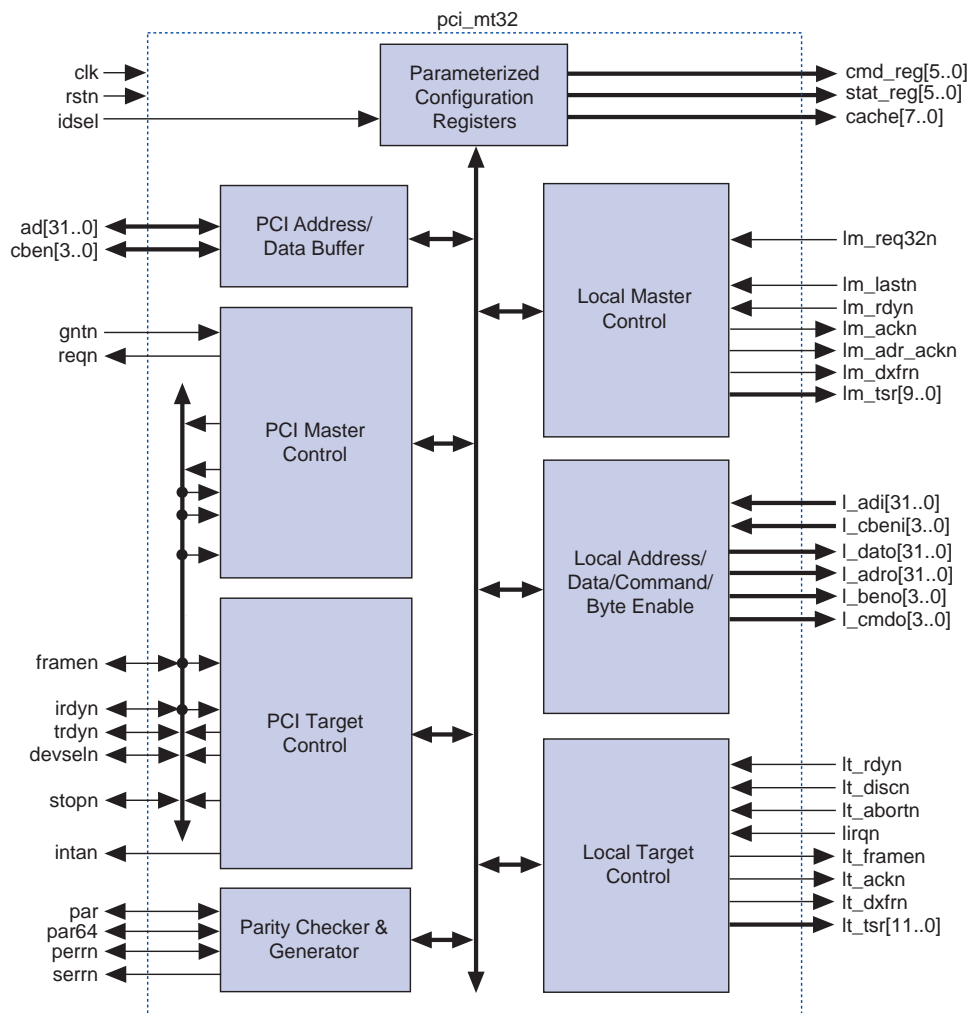
Figure 2. *pci\_mt32* Functional Block Diagram

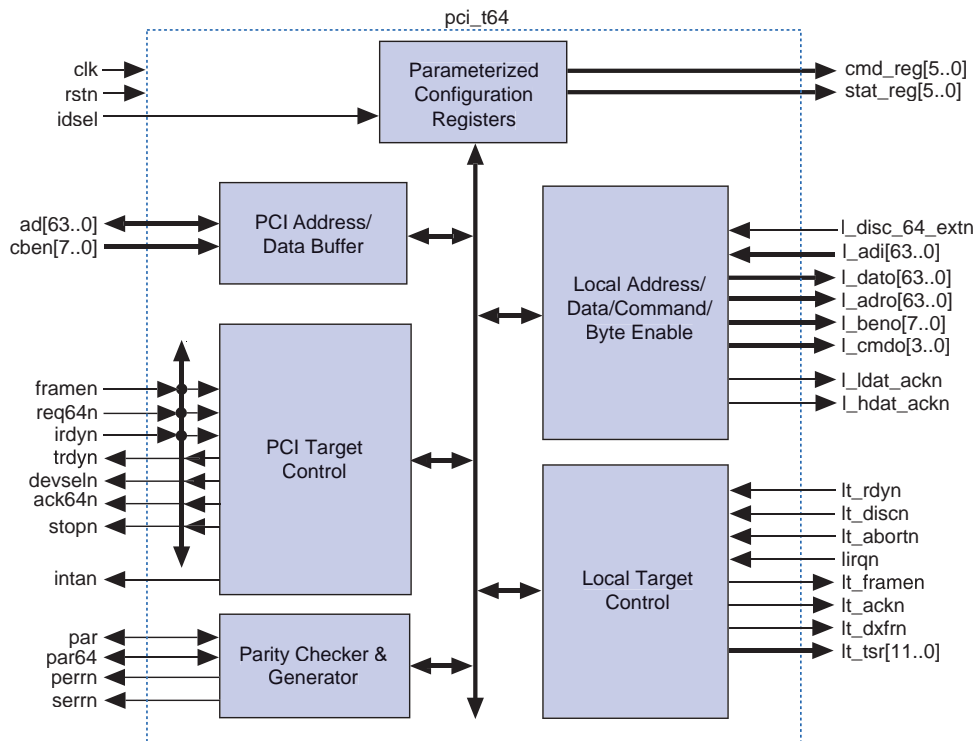
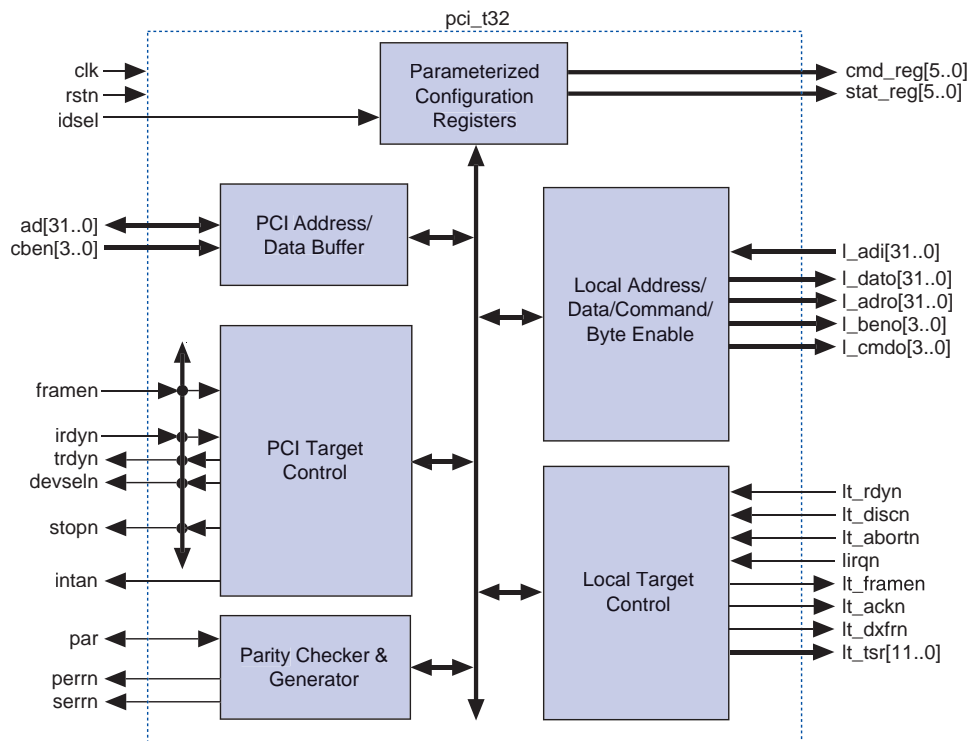
Figure 3. *pci\_t64* Functional Block Diagram



Figure 4. *pci\_t32* Functional Block Diagram

## Features...

This section describes the features of the following PCI MegaCore™ functions: *pci\_mt64*, *pci\_mt32*, *pci\_t64*, and *pci\_t32*. These functions are parameterized MegaCore functions implementing peripheral component interconnect (PCI) interfaces.

- Flexible general-purpose interfaces that can be customized for specific peripheral requirements
- Dramatically shortens design cycles
- Fully compliant with the PCI Special Interest Group (PCI SIG) **PCI Local Bus Specification, Revision 2.2** timing and functional requirements
- Extensively verified using industry-proven Phoenix Technology test bench
- Extensively hardware tested using the following hardware and software (see “[Compliance Summary](#)” on page 21 for details)
  - Agilent E2928A PCI Bus Analyzer and Exerciser
  - Agilent E2920 Computer Verification Tools, PCI series
  - Altera FLEX 10KE PCI Development Board
  - Altera APEX 20KE PCI Development Board

## ...and More Features

- Optimized for the APEX, ACEX, FLEX, Excalibur, Stratix, and Cyclone devices.
- 66-MHz compliant when used with 66-MHz PCI-compliant Altera devices
- No-risk OpenCore™ feature allows you to instantiate and simulate designs prior to purchase
- Supports most PCI commands, including: configuration read/write, memory read/write, I/O read/write, memory read multiple (MRM), memory read line (MRL), and memory write and invalidate (MWI)
- PCI target features (applies to `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32`):
  - Capabilities list pointer support
  - Parity error detection
  - Up to six base address registers (BARs) with adjustable memory size and type
  - Expansion ROM BAR support
  - Local side can request a target abort, retry, or disconnect
  - Local-side interrupt request
- PCI master features (applies to `pci_mt64` and `pci_mt32`):
  - Host bridge application support
- 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, latency timer, cache line size, interrupt pin, interrupt line
- 64-bit PCI master only features (applies to `pci_mt64`):
  - 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
- 64-bit PCI target only features (applies to `pci_t64` and `pci_mt64`):
  - 64-bit addressing capable
  - Automatically responds to 32- or 64-bit transactions

Altera® peripheral component interconnect (PCI) MegaCore® functions provide solutions for interfacing your system to a 32-bit or 64-bit PCI bus. A PCI bus can be used to implement peripheral devices such as network adapters, graphic accelerator boards, and embedded control modules.

The Altera PCI functions—`pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32`—enhance your productivity by allowing you to focus your efforts on the custom logic surrounding the PCI interface. The functions are optimized for Altera APEX, ACEX, FLEX, Excalibur, Stratix, and Cyclone devices and are fully tested to meet the requirements of the PCI Special Interest Group (SIG) **PCI Local Bus Specification, Revision 2.2** and **Compliance Checklist, Revision 2.2**. You can test-drive Altera PCI MegaCore functions using the OpenCore™ feature to compile and simulate the functions within your custom logic. When you are ready to license a function, contact your local Altera sales representative.

The PCI compiler contains everything you need to use Altera PCI solutions including the MegaCore functions—which can be instantiated with a wizard-driven interface—behavioral models, testbench, and reference designs. You can download the PCI compiler from the IP MegaStore area on the Altera web site at <http://www.altera.com/IPmegastore>. Refer to the **PCI Compiler Data Sheet** for more information on the PCI compiler contents and how to obtain and install the compiler. The **PCI MegaCore Function User Guide** provides information on how to get started using the PCI MegaCore functions and describes the technical specifications of the functions.



You can use the PCI compiler to test-drive several PCI MegaCore functions using the OpenCore feature; however, each PCI MegaCore function must be licensed separately.

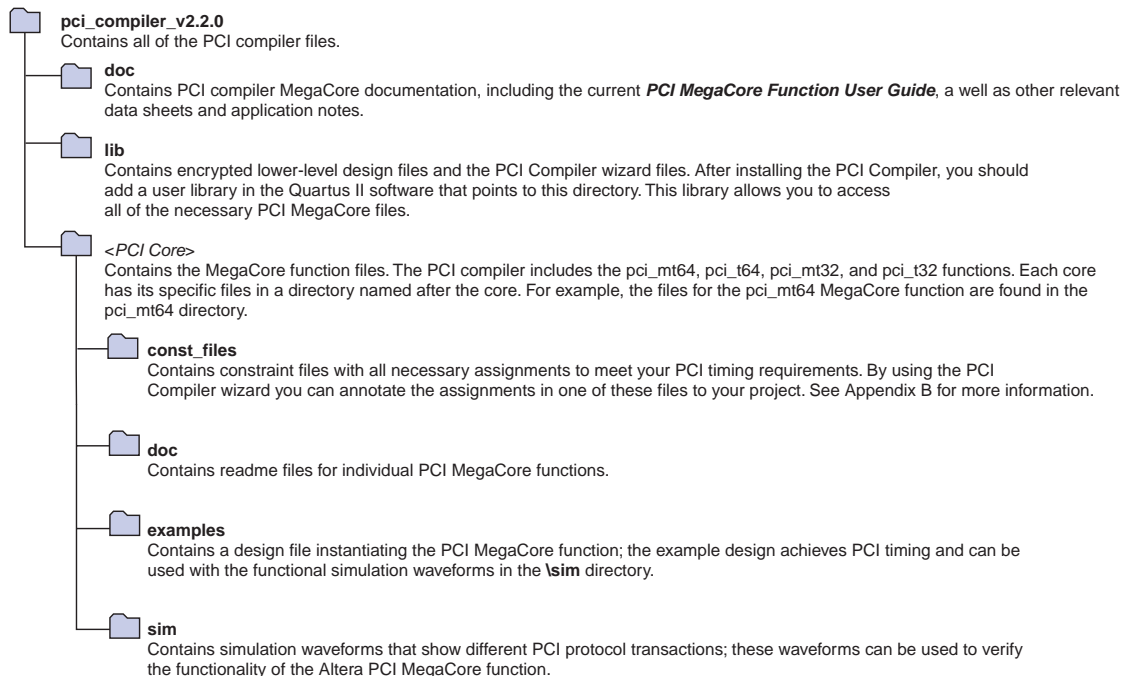
This section discusses the following areas:

- PCI MegaCore directory structure
- Design flow using PCI MegaCore functions
- Design walkthrough
  - Generating a project-specific PCI instance of the `pci_mt64`, `pci_mt32`, `pci_t64`, or `pci_t32` functions
  - Generating project-specific constraint files to achieve PCI timing requirements
  - Compilation, timing analysis, and functional simulation

## PCI MegaCore Directory Structure

The PCI compiler installs files—including the PCI MegaCore function files—into several directories; the top-level directory is `\pci_compiler_v2.2.0`. [Figure 1](#) describes the directory structure for the MegaCore functions only; refer to the *PCI Compiler Data Sheet* for the PCI compiler directory structure.

Figure 1. PCI MegaCore Function Directory Structure



# Altera PCI MegaCore Function Design Flow

Altera PCI MegaCore functions are flexible intellectual property (IP) functions that can be integrated into any design flow supported by Altera tools, including third-party EDA tools supported for synthesis and simulation.

The following steps describe the design flow when using Altera PCI MegaCore functions:

1. Obtain the PCI MegaCore Functions
2. Instantiate a PCI MegaCore Function in Your Design
3. Synthesize
4. Simulate
5. Analyze Timing
6. License MegaCore Function
7. Configure a Device

## Obtain the PCI MegaCore Functions

The `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` functions are included in the PCI compiler, which can be downloaded from the Altera web site at <http://www.altera.com/IPmegastore>. Refer to the **PCI Compiler Data Sheet** for more information on the PCI compiler, including a full list of contents, how to obtain the compiler, and installation instructions.

## Instantiate a PCI MegaCore Function in Your Design

You can use the PCI compiler wizard to choose the desired Altera PCI MegaCore function, set the parameters, and generate an instance of the function for your design. After you create the instance, you can add it as a module to your overall design. The PCI module provides the interface between the PCI bus and your local design.

## Synthesize

After integrating the instance into your overall design, synthesize the full design. The Altera PCI functions are provided as encrypted design files that must be synthesized with the Quartus II software. If you use a third-party EDA tool for synthesis, instantiate the PCI MegaCore function as a black box in your design. Then synthesize the design to generate a netlist. The PCI MegaCore logic is synthesized when the netlist is compiled using the Quartus II software.

## Simulate

Altera provides behavioral models for functional simulation in third-party EDA tools, as well as VHDL and Verilog HDL test benches. See **AN 169: Simulating the PCI Behavioral Models** and the **PCI Testbench User Guide** for information on using the models and test benches.

After synthesis, you can perform pre- or post- place-and-route simulation (functional and timing, respectively) using the Quartus II software. The MegaCore functions include functional waveform simulation files, which you can use to simulate your design using the Quartus II software.

Alternatively, after you have licensed a PCI MegaCore function, you can use a VHDL Output File (**.vho**) or Verilog Output File (**.vo**)— generated by the Altera Quartus II software—to simulate your design in third-party EDA tools. To perform functional simulation, generate a **.vho** or **.vo** with a Standard Delay Format (SDF) Output File (**.sdo**) but do not compile the **.sdo** in your simulation environment. To perform timing simulation, compile the **.sdo** in your simulation environment. Refer to Quartus II Help for details on generating **.vho** and **.vo** netlist files.



Altera has developed the Quartus II Nativelink Guidelines for use with the Quartus II software, which describe how to create, compile, and simulate your design with tools from leading EDA vendors. These guidelines are available on the software installation CD-ROMs and on the Altera web site at <http://www.altera.com>.

## Obtain PCI Constraint File and Analyze Timing

The Quartus II software provides static timing analysis results, allowing you to verify that your design timing requirements are achieved. Altera provides constraint files, which add timing requirements, logic option settings, and logic location assignments to your project to ensure that the PCI MegaCore function achieves PCI timing requirements. Refer to “[Generating Project-Specific Constraint Files to Achieve PCI Timing Requirements](#)” on page 16 for detailed information on integrating PCI constraint files into your project.



Constraint files are specific to the core being used, (i.e., **pci\_mt32**, **pci\_mt64**, etc.) as well as the device and package being used. In many cases the Quartus II revision must match. To obtain a PCI constraint file, visit [http://www.altera.com/pci\\_cf](http://www.altera.com/pci_cf)

## License MegaCore Function

Once you have determined that an Altera PCI MegaCore function meets your design needs, contact your local Altera sales office or distributor sales representative to license the MegaCore function. Contact information for all regional offices is available on the Altera web site at <http://www.altera.com>.

## Configure a Device

After you have compiled and analyzed your design and licensed the desired Altera MegaCore function, you are ready to configure your targeted Altera device. For more information on configuring Altera devices, please refer to the Quartus II software help system.

This section describes the design flow using the Altera PCI compiler wizard to instantiate the `pci_mt64`, `pci_t64`, `pci_mt32`, or `pci_t32` MegaCore function. The wizard streamlines the design entry process, making designing with Altera PCI MegaCore functions easier and less time-consuming. Because the PCI MegaCore functions are parameterized, you can set the parameters to meet the needs of your specific application.

These instructions assume that:

- You are using a PC.
- You are familiar with the Quartus II software.
- Quartus II software version 2.1 Service Pack 1 (or higher) is installed in the default location (`c:\quartus`).
- The Altera PCI MegaCore files are located in the default directory, `c:\megacore`. If the files are installed in a different directory on your system, substitute the appropriate path name.
- You are using the OpenCore feature to test-drive the PCI MegaCore function or you have licensed the function.



You can use Altera's OpenCore feature to compile and simulate PCI MegaCore functions, allowing you to evaluate the functions before deciding to license them.

## Generating a Project-Specific Instance of the `pci_mt64`, `pci_t64`, `pci_mt32`, or `pci_t32` Function

Altera provides a MegaWizard Plug-In Manager, which you can use within the Quartus II software or as a stand-alone application. Using a wizard allows you to create or modify design files that instantiate a MegaCore function. You can then instantiate the design file generated by the wizard in your project.

## Design Walk-Through

You can use the Altera OpenCore feature to compile and simulate the MegaCore functions, allowing you to evaluate the functions before deciding to license them. However, you must obtain a license from Altera before you can generate programming files or EDIF, VHDL, or Verilog HDL gate-level netlist files.



Because the PCI compiler wizard relies on library files that only exist in version 2.1 Service Pack 1 of the Quartus II software or higher, you should not use earlier versions of the software when working with the wizard.

To create a project-specific instance of the `pci_mt64`, `pci_t64`, `pci_mt32`, or `pci_t32` function, follow these steps:

1. Create a new project in the Quartus II software by choosing **New project** (File menu); add `c:\megacore\PCICompiler_2.2.0\lib` as a user library.
2. Start the MegaWizard Plug-In Manager by choosing **MegaWizard Plug-In Manager** (Tools menu).
3. The MegaWizard Plug-In Manager dialog box is displayed.

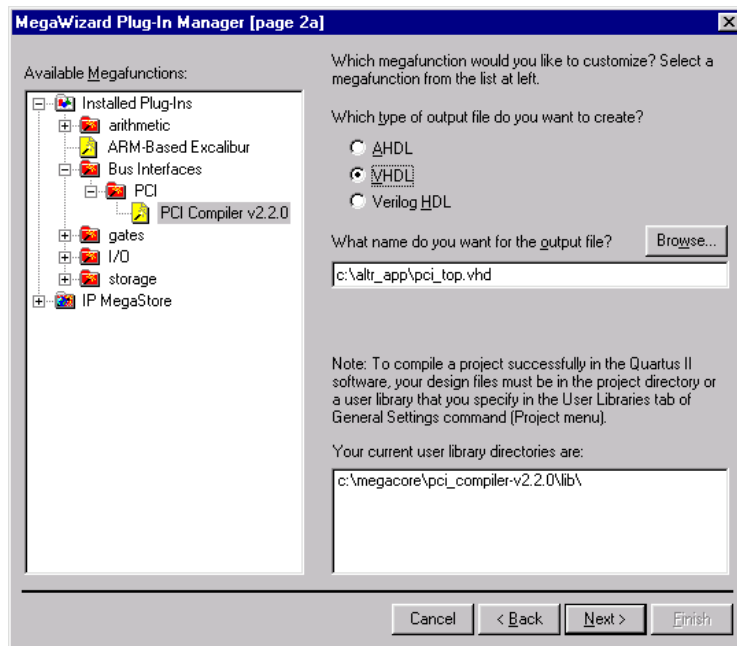


Refer to Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

4. Specify that you want to create a new custom megafunction and click **Next**.
5. Select **PCI Compiler** from the **Bus Interfaces folder** (see [Figure 2](#)).

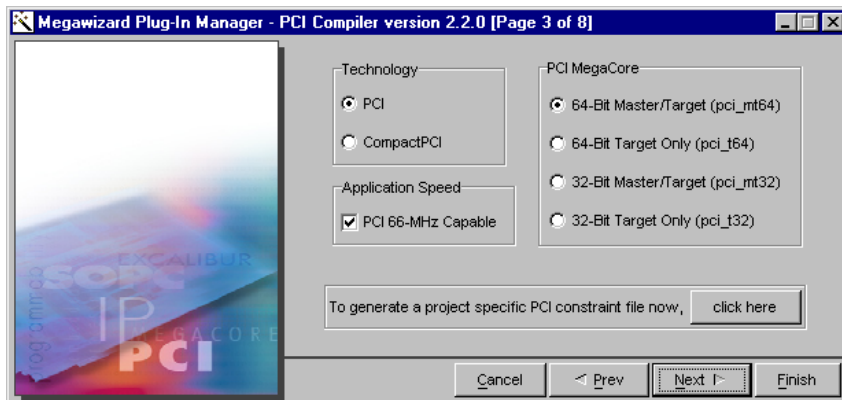


Figure 2. Selecting the PCI Compiler Function



6. Select the type of output file that you want to create (i.e., AHDL, VHDL, or Verilog HDL), and specify a filename. Click **Next**.
7. Select the PCI system technology, **PCI** or **CompactPCI**, the application speed (turn the PCI 66 MHz Capable option on or off), and the PCI MegaCore function desired, **pci\_mt64**, **pci\_t64**, **pci\_mt32**, or **pci\_t32** (see Figure 3). See “Parameters” on page 37 for more information on the application speed parameter. See “General Description” on page 1 to determine which function your design should target.

Figure 3. Selecting the PCI Technology & PCI MegaCore Function



The next several wizard screens allow you to set the MegaCore function parameters to customize the PCI instance for your application. For a detailed description of the PCI parameters and configuration registers available in the PCI functions, see [“Configuration Registers” on page 50](#) and [“Parameters” on page 37](#).

### Generating Project-Specific Constraint Files to Achieve PCI Timing Requirements

After setting the PCI MegaCore function parameters, you have the option to generate a project-specific constraint file that will ensure the PCI MegaCore function achieves PCI timing requirements in your design. To download a device-specific constraint file please visit [http://www.altera.com/pci\\_cf](http://www.altera.com/pci_cf). If you choose not to generate a constraint file at this time, click **Next** and skip to step 7 on page 18.

Altera provides software constraint files to ensure that the PCI MegaCore function achieves PCI specification timing requirements in Altera devices. The constraint files contain logic option settings, as well as device, PCI pin, timing, and location assignments for PCI MegaCore function logic.

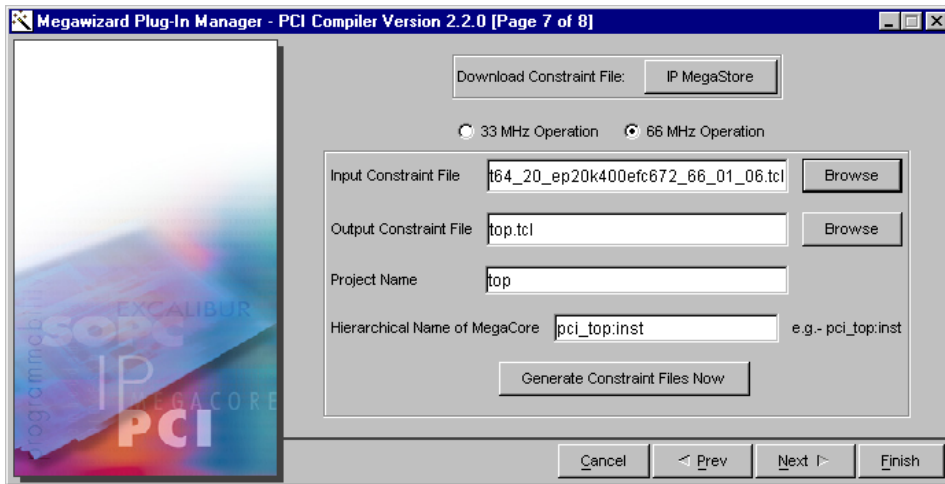
When you have chosen a specific Altera device and are ready to assign pin locations, the constraint files should be incorporated into your design. Quartus II constraint files are provided as a Tool Command Language (.tcl) file that can be used to generate a CSF and ESF in the Quartus II software.



For more information on supported devices and using constraint files in your design, See [“Appendix B: Using PCI Constraint Files” on page 155](#).

Refer to [Figure 4](#) for steps 1 through 6.

Figure 4. Generating Project Specific Constraint Files



1. Use the wizard web link to download a constraint file from the Altera IP MegaStore site on the Internet. Save the constraint file to your hard disk. You may also select one of the constraint files provided in the PCI compiler (<PCI function>\const\_files\ for Quartus II projects, where <PCI function> is pci\_mt64, pci\_mt32, pci\_t64, or pci\_t32).
2. Select either the 33-MHz operation or the 66-MHz operation option.
3. Select an input constraint file by browsing to the local directory where your input constraint file is saved. (Valid input files have the extension .tcl.)
4. Browse to the directory where you would like to save your project constraint file(s) and enter the name of the output file.
5. Enter the name of your project.
6. Enter the project hierarchy for the PCI MegaCore function instance. For example, you may have created a PCI function instance named **pci\_top** through the PCI compiler wizard. You may have instantiated **pci\_top:inst** in your project **top**. Thus, the hierarchical name of the MegaCore function is **pci\_top:inst**. If you do not apply the correct project hierarchy the project constraints will not be applied correctly and you may not meet PCI timing requirements.

### Choose **Generate Constraint Files Now**.

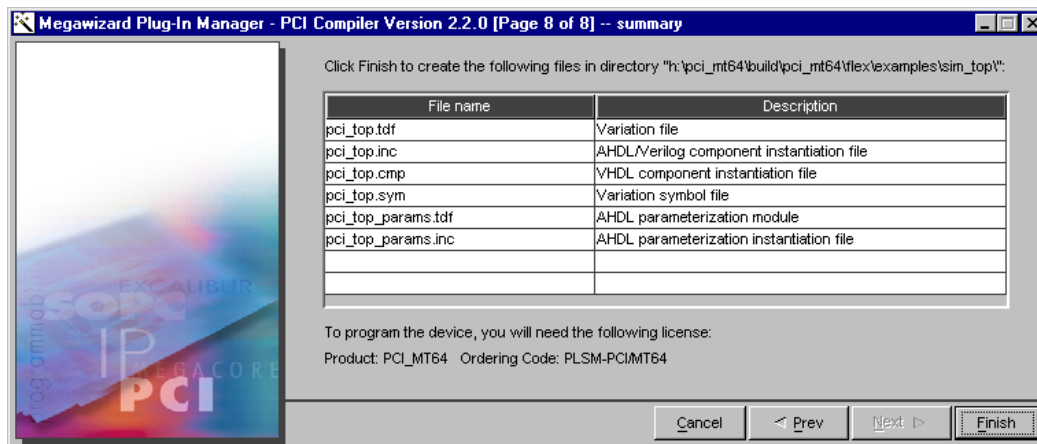


If you have a Tcl file as your input and output for the PCI compiler wizard, you can save these files into any directory. The Tcl file contains all necessary information to create a CSF and ESF. To generate a project-specific CSF and ESF—after generating a project-specific Tcl file with the wizard—run the Tcl script through the Quartus II Tcl console while the PCI project is active. Refer to Quartus II Help for more information on using Tcl in the Quartus II software.

The Tcl file assumes a fresh project, erasing all preexisting project settings. You must reapply all your project specific settings after running the PCI Tcl file.

7. Click **Next** to view the summary screen.
8. The summary screen lists the design files that the PCI compiler wizard creates as well as the product ordering code for the MegaCore function targeted in your design (see [Figure 5](#)). You will need the product ordering code to license the MegaCore function. Click **Finish**.

Figure 5. Summary Screen



## Compilation, Functional Simulation & Timing Analysis in the Quartus II Software

This section explains the steps required to compile your design, to ensure PCI timing is achieved using static timing analysis, and to simulate your design functionally in the Quartus II software.

These steps assume that you use the **pci\_top.tdf** file in the *<PCI function>\examples\quartus2* directory. This example file instantiates six 32-bit BARs and uses all other parameter settings used in the functional simulation waveform files available with the PCI MegaCore functions.

### Compilation

The following steps explain how to compile your design.

1. Create a project with the **pci\_top.tdf** file in the Quartus II software.
2. Choose the **Compile Mode** command (Processing menu).
3. Choose **Start Compile** (Processing menu) to compile your design.



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

### Timing Analysis

The following steps explain how to verify the timing results for your design.

1. Open the **Compilation Report** (Processing menu) and expand the Timing Analysis section.
2. The Quartus II software lets you perform the following five types of timing analysis:
  - **f<sub>MAX</sub>**—The **fmax** section reports the maximum clock frequency and identifies the longest delay paths between registers.
  - **t<sub>SU</sub>**—The **tsu** section reports the setup times of the registers.
  - **t<sub>H</sub>**—The **th** section reports the hold times of the registers.
  - **t<sub>CO</sub>**—The **tco** section reports the clock-to-output delays of the registers.
  - **t<sub>PD</sub>**—The **tpd** section reports the combinatorial pin-to-pin delays.

### Functional Simulation

To perform functional simulation, perform the following steps:

1. Change to **Simulate Mode** (Processing menu) to functionally simulate your design.
2. In the Quartus II **Simulator Settings** dialog box, choose the **Mode** tab and select **Functional**. Click **Apply**.
3. Choose the **Time/Vectors** tab and specify a Vector Waveform File (**.vwf**) from the **c:\megacore\PCICompiler\_<version>\<PCI function>\sim\quartus2\<target or master>** directory as the source of vector stimuli and click **Apply**.
4. Choose **Run Simulation** (Processing menu) to simulate your design and view the simulation results. The different simulation files show the behavior of the PCI and local-side signals for different types of transactions.



If you are simulating the MegaCore function using the PCI behavioral models and the PCI testbench in third-party simulation tools, refer to **AN 169: Simulating the PCI MegaCore Function Behavioral Models** and the **PCI Testbench User Guide** for more information.

## Compliance Summary

The `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` functions are compliant with the requirements specified in the PCI SIG **PCI Local Bus Specification, Revision 2.2** and **Compliance Checklist, Revision 2.2**. The function is shipped with sample Quartus II Vector Waveform Files (.vwf), PCI Testbench, and behavioral model which can be used to validate the functions.

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 PCI specification Appendix C, 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 included the Agilent E2928A PCI Bus Exerciser and Analyzer, an Altera PCI development board with a device configured with the MegaCore function and a reference design, and PCI bus agents such as the host bridge, Ethernet network adapter, and video card. (The Altera PCI MegaCore functions are tested on the following Altera devices: EPF10K100EFC484-1, EPF10K200SFC672-1, EP20K400EFC672-1, EP20K1000EFC672-1, EP20K1000CF672C7.) The 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 are performed to guarantee robustness and strict compliance. These tests include:

- 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 simulates random behavior on the PCI bus by randomizing transactions with variable parameters such as:

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

The local side also emulates a variety of test conditions in which the PCI MegaCore function experiences random wait states and terminations. During the tests, the Agilent E2928A PCI Bus Exerciser and Analyzer also acts as a PCI protocol and data integrity checker as well as a logic analyzer to aid in debugging. This testing ensures that the functions operate under the most stringent conditions in your system. For more information on the Agilent E2928A PCI Bus Exerciser and Analyzer, see the Agilent web site at <http://www.agilent.com>.

## 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 wire-ORed with other agents. The signaling agent asserts the open-drain signal, and a weak pull-up resistor deasserts the open-drain signal. The pull-up resistor may require two or three PCI bus clock cycles to restore the open-drain signal to its inactive state.



**Table 1** summarizes the PCI bus signals that provide the interface between the PCI MegaCore functions and the PCI bus.

<i>Table 1. PCI Interface Signals (Part 1 of 3)</i>			
Name	Type	Polarity	Description
clk	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> .
rstn	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> .
gntn	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.
reqn	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.
ad[63..0]	Tri-State	–	Address/data bus. The <code>ad[63..0]</code> 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 <code>irdyn</code> and <code>trdyn</code> are both asserted. In the case of a 32-bit data phase, only the <code>ad[31..0]</code> bus holds valid data. For <code>pci_mt32</code> and <code>pci_t32</code> , only <code>ad[31..0]</code> is implemented.
cben[7..0]	Tri-State	–	Command/byte enable. The <code>cben[7..0]</code> 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 <code>pci_mt32</code> and <code>pci_t32</code> , only <code>cben[3..0]</code> is implemented.
par	Tri-State	–	Parity. The <code>par</code> signal is even parity across the 32 least significant address/data bits and four least significant command/byte enable bits. In other words, the number of 1s on <code>ad[31..0]</code> , <code>cben[3..0]</code> , and <code>par</code> equal an even number. The parity of a data phase is presented on the bus on the clock following the data phase.
par64	Tri-State	–	Parity 64. The <code>par64</code> signal is even parity across the 32 most significant address/data bits and the four most significant command/byte enable bits. In other words, the number of 1s on <code>ad[63..32]</code> , <code>cben[7..4]</code> , and <code>par64</code> equal an even number. The parity of a data phase is presented on the bus on the clock following the data phase. This signal is not implemented in the <code>pci_mt32</code> and <code>pci_t32</code> functions.
idsel	Input	High	Initialization device select. The <code>idsel</code> input is a chip select for configuration transactions.

*Table 1. PCI Interface Signals (Part 2 of 3)*

Name	Type	Polarity	Description
<code>framen</code> (1)	STS	Low	Frame. The <code>framen</code> signal is an output from the current bus master that indicates the beginning and duration of a bus operation. When <code>framen</code> is initially asserted, the address and command signals are present on the <code>ad[63..0]</code> and <code>cben[7..0]</code> buses ( <code>ad[31..0]</code> and <code>cben[3..0]</code> only for 32-bit functions). The <code>framen</code> signal remains asserted during the data operation and is deasserted to identify the end of a transaction.
<code>req64n</code> (1)	STS	Low	Request 64-bit transfer. The <code>req64n</code> signal is an output from the current bus master and indicates that the master is requesting a 64-bit transaction. <code>req64n</code> has the same timing as <code>framen</code> . This signal is not implemented in <code>pci_mt32</code> and <code>pci_t32</code> .
<code>irdyn</code> (1)	STS	Low	Initiator ready. The <code>irdyn</code> 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, <code>irdyn</code> indicates that the address bus has valid data. In a read transaction, <code>irdyn</code> indicates that the master is ready to accept data.
<code>devseln</code> (1)	STS	Low	Device select. Target asserts <code>devseln</code> to indicate that the target has decoded its own address and accepts the transaction.
<code>ack64n</code> (1)	STS	Low	Acknowledge 64-bit transfer. The target asserts <code>ack64n</code> to indicate that the target can transfer data using 64 bits. The <code>ack64n</code> has the same timing as <code>devseln</code> . This signal is not implemented in <code>pci_mt32</code> and <code>pci_t32</code> .
<code>trdyn</code> (1)	STS	Low	Target ready. The <code>trdyn</code> signal is a target output, indicating that the target can complete the current data transaction. In a read operation, <code>trdyn</code> indicates that the target is providing valid data on the address bus. In a write operation, <code>trdyn</code> indicates that the target is ready to accept data.
<code>stopn</code> (1)	STS	Low	Stop. The <code>stopn</code> signal is a target device request that indicates to the bus master to terminate the current transaction. The <code>stopn</code> signal is used in conjunction with <code>trdyn</code> and <code>devseln</code> to indicate the type of termination initiated by the target.

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

Name	Type	Polarity	Description
<code>perrn</code>	STS	Low	Parity error. The <code>perrn</code> signal indicates a data parity error. The <code>perrn</code> signal is asserted one clock following the <code>par</code> and <code>par64</code> signals or two clocks following a data phase with a parity error. The PCI functions assert the <code>perrn</code> signal if a parity error is detected on the <code>par</code> or <code>par64</code> signals and the <code>perrn</code> bit (bit 6) in the command register is set. The <code>par64</code> signal is only evaluated during 64-bit transactions in <code>pci_mt64</code> and <code>pci_t64</code> functions. In <code>pci_mt32</code> and <code>pci_t32</code> , only <code>par</code> is evaluated.
<code>serrn</code>	Open-Drain	Low	System error. The <code>serrn</code> signal indicates system error and address parity error. The PCI functions assert <code>serrn</code> if a parity error is detected during an address phase and the <code>serrn</code> enable bit (bit 8) in the command register is set.
<code>intan</code>	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.

**Note:**

- (1) In the MegaCore function symbols, the signals are separated into two components: input and output. For example, `framen` has the input `framen_in` and the output `framen_out`. This separation of signals allows the use of devices that do not meet set-up times to implement a PCI interface. Driving the input part of one or more of these signals to a dedicated input pin and the output part to a regular I/O pin, allows devices that cannot otherwise meet set-up times to meet them.

## Parameterized Configuration Register Signals

**Table 2** summarizes the PCI local interface signals for the parameterized configuration register signals.

<i>Table 2. Parameterized Configuration Register Signals</i>			
Name	Type	Polarity	Description
cache[7..0]	Output	–	Cache registers output. The cache[7..0] bus is the same as the configuration space cache register. The local-side logic uses this signal to provide support for cache commands.
cmd_reg[5..0]	Output	–	Command register output. The cmd_reg[5..0] bus drives the important signals of the configuration space command register to the local side. See <a href="#">Table 3</a> .
stat_reg[5..0]	Output	–	Status register output. The stat_reg[5..0] bus drives the important signals of the configuration space status register to the local side. See <a href="#">Table 4</a> .

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

<i>Table 3. PCI Command Register Output Bus (cmd_reg[5..0]) Bit Definition</i>		
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	mw_i_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.

**Table 4** shows definitions for the PCI status register bits.

<i>Table 4. PCI Status Register Output Bus (stat_reg[5..0]) Bit Definition</i>		
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.




## Local Address, Data, Command and Byte Enable Signals

**Table 5** summarizes the PCI local interface signals for the address, data, command, and byte enable signals.

*Table 5. PCI Local Address, Data, Command and Byte Enable Signals (Part 1 of 4)*

Name	Type	Polarity	Description
<code>l_adi[63..0]</code>	Input	—	<p>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 considered.</p> <ul style="list-style-type: none"> <li>■ 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.</li> <li>■ The <code>l_adi[63..0]</code> bus is driven active by the local-side device 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.</li> <li>■ 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 and 32-bit target read and 64-bit master write transactions.</li> </ul>
<code>l_cbeni[7..0]</code>	Input	—	<p>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 dual address cycle (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_adr_ackn</code> is asserted. The local side must provide the command with the same encoding as specified in the <b>PCI Local Bus Specification, Revision 2.2</b>.</p> <p>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 after <code>lm_adr_ackn</code> is asserted. The PCI MegaCore functions drive the byte-enable value from the local side to the PCI side and maintain the same byte-enable value for the entire transaction. 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.</p>

Table 5. PCI Local Address, Data, Command and Byte Enable Signals (Part 2 of 4)

Name	Type	Polarity	Description
<code>l_disc_64_extn</code>	Input	Low	<p>Local disable 64-bit PCI extension signals. If left unconnected, the default value is high. When driven high, the <code>ad[63..0]</code> and <code>cben[7..0]</code> signals operate normally and the <code>pci_mt64</code> and <code>pci_t64</code> functions will initiate and respond to 32- and 64-bit PCI transactions. When driven low, the <code>ad[63..32]</code> and <code>cben[7..4]</code> signals are disabled—by being driven at all times—and the <code>pci_mt64</code> and <code>pci_t64</code> functions will only initiate and respond to 32-bit PCI transactions.</p> <p> This signal is provided for 64-bit PCI cards that may be inserted into a 32-bit PCI slot. The PCI card should be designed to detect whether it's in a 32- or 64-bit PCI slot and drive the <code>l_disc_64_extn</code> signal accordingly. To avoid damaging the Altera device, do not allow <code>ad[63..0]</code> and <code>cben[7..4]</code> to float.</p> <p> This signal is not implemented in the <code>pci_mt32</code> and <code>pci_t32</code> functions.</p>
<code>l_adro[63..0]</code>	Output	—	<p>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> functions, the <code>l_adro[63..32]</code> bus is driven with a valid address. DAC is indicated by sampling the <code>lt_tsr[11]</code> status signal set. For more information on the local target status signals, refer to <a href="#">Table 7</a>.</p> <p> 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.</p>

*Table 5. PCI Local Address, Data, Command and Byte Enable Signals (Part 3 of 4)*

Name	Type	Polarity	Description
<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 <code>pci_mt64</code> and <code>pci_t64</code> 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> function will either assert <code>l_ldat_ackn</code> or <code>l_hdat_ackn</code> to indicate whether the address for the current byte enables 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 <code>pci_mt64</code> and <code>pci_t64</code> 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> function will either assert <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.
<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.

*Table 5. PCI Local Address, Data, Command and Byte Enable Signals (Part 4 of 4)*

Name	Type	Polarity	Description
<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. In other words, 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. 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. This signal is not implemented in the <code>pci_mt32</code> and <code>pci_t32</code> functions.

## Target Local-Side Signals

**Table 6** summarizes the target interface signals that provide the interface between the MegaCore function to 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 6. Target Signals Connecting to the Local Side (Part 1 of 3)*

Name	Type	Polarity	Description
<code>lt_abortn</code>	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.



*Table 6. Target Signals Connecting to the Local Side (Part 2 of 3)*

Name	Type	Polarity	Description
lt_discn	Input	Low	<p>Local target disconnect request. The <code>lt_discn</code> input requests the PCI MegaCore function to issue a retry or a disconnect. The PCI MegaCore function issue a retry or disconnect depending on when the signal is asserted during a transaction.</p> <p> 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 <code>lt_discn</code> if the transaction crosses its memory space.</p>
lt_rdyn	Input	Low	<p>Local target ready. The local side asserts <code>lt_rdyn</code> to indicate a valid data input during target read, or ready to accept data input during a target write. During a target read, <code>lt_rdyn</code> deassertion suspends the current transfer (i.e., a wait state is inserted by the local side). During a target write, an inactive <code>lt_rdyn</code> 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 <code>lt_rdyn</code> inserts wait states on the local side.</p> <p> <code>lt_rdyn</code> is sampled one clock before actual data is transferred on the local side.</p>
lt_framen	Output	Low	Local target frame request. The <code>lt_framen</code> output is asserted while the PCI MegaCore function is requesting access to the local side. It is asserted one clock before the function asserts <code>devseln</code> , 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 <code>lt_ackn</code> to indicate valid data output during a target write, or ready to accept data during a target read. During a target read, an inactive <code>lt_ackn</code> indicates that the function is not ready to accept data and local logic should hold off the bursting operation. During a target write, <code>lt_ackn</code> de-assertion suspends the current transfer (i.e., a wait state is inserted by the PCI master). The <code>lt_ackn</code> 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 <code>lt_dxfrn</code> signal when a data transfer on the local side is successful during a target transaction.
lt_tsr[11..0]	Output	–	Local target transaction status register. The <code>lt_tsr[11..0]</code> bus carries several signals which can be monitored for the transaction status. See <a href="#">Table 7</a> .

*Table 6. Target Signals Connecting to the Local Side (Part 3 of 3)*

Name	Type	Polarity	Description
<code>lirqn</code>	Input	Low	Local interrupt request. The local-side peripheral device asserts <code>lirqn</code> to signal a PCI bus interrupt. Asserting this signal forces the PCI MegaCore function to assert the <code>intan</code> signal for as long as the <code>lirqn</code> signal is asserted.

**Table 7** shows definitions for the local target transaction status register outputs.

*Table 7. Local Target Transaction Status Register (`lt_tsr[11..0]`) Bit Definition*

Bit Number	Bit Name	Description
5..0	<code>bar_hit[5..0]</code>	Base address register hit. Asserting <code>bar_hit[5..0]</code> 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 <code>bar_hit[5..0]</code> bus is used for the corresponding base address register (e.g., <code>bar_hit[0]</code> is used for BAR0). The <code>bar_hit[5..0]</code> bus has the same timing as the <code>lt_framen</code> signal. When a 64-bit base address register is used, both <code>bar_hit[0]</code> and <code>bar_hit[1]</code> are asserted to indicate that <code>pci_mt64</code> and <code>pci_t64</code> have claimed the transaction.
6	<code>exp_rom_hit</code>	Expansion ROM register hit. The PCI MegaCore function asserts this signal when the transaction address matches the address in the expansion ROM BAR.
7	<code>trans64bit</code>	64-bit target transaction. The <code>pci_mt64</code> and <code>pci_t64</code> 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 <code>pci_mt32</code> and <code>pci_t32</code> .
8	<code>targ_access</code>	Target access. The PCI MegaCore functions assert this signal when PCI target access is in progress.
9	<code>burst_trans</code>	Burst transaction. When asserted, this signal indicates that the current target transaction is a burst. This signal is asserted if the PCI MegaCore functions detect both <code>framen</code> and <code>irdyn</code> signals asserted at the same time during the first data phase.
10	<code>pci_xfr</code>	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	<code>dac_cyc</code>	Dual address cycle. When asserted, this signal indicates that the current transaction is using a dual address cycle.

## Master Local-Side Signals

**Table 8** 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 driven to the deasserted state.

**Table 8. PCI Master Signals Interfacing to the Local Side (Part 1 of 2)**

Name	Type	Polarity	Description
<code>lm_req32n</code>	Input	Low	<p>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_dato[31..0]</code> for a master read transaction is valid.</p> <p> 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.</p>
<code>lm_req64n</code>	Input	Low	<p>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. 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>.</p> <p> 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.</p>

*Table 8. PCI Master Signals Interfacing to the Local Side (Part 2 of 2)*

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

**Table 9** shows definitions for the local master transaction status register outputs.

**Table 9. *pci\_mt64 & pci\_mt32 Local Master Transaction Status Register (lm\_tsr[9..0]) Bit Definition***  
**Note (1)**

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_xfr	Data transfer. 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 ended 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_phase	Data phase. This signal indicates that a successful data transfer has occurred on the PCI side in the prior clock cycle. This signal can be used by the local side to keep track of how much data was actually transferred on the PCI side.
9	trans64	64-bit transaction. This signal indicates that the target claiming the transaction has asserted its <code>ack64n</code> signal. Because <code>pci_mt32</code> does not request 64-bit transactions, this signal is reserved.

**Note:**

- (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.

## MegaWizard Plug-In

The PCI MegaWizard® Plug-In streamlines the design entry process, making it easier and less time consuming to design with Altera MegaCore functions. You can either launch the PCI compiler MegaWizard Plug-In from within the Quartus II software, or from the command line. The PCI compiler wizard generates an Altera hardware description language (AHDL), VHDL, or Verilog hardware description language (HDL) instance of the Altera PCI MegaCore function that can be instantiated in your design, and the generated files set all nonreserved parameters in the Altera PCI MegaCore function. The PCI MegaCore parameters are described in [Table 9 on page 35](#).

When the wizard compilation is complete, the following files are generated:

- Text Design File (.tdf), VHDL Design File (.vhd), and Verilog Design File (.v)—Used to instantiate an instance of the `pci_mt64`, `pci_t64`, `pci_mt32`, or `pci_t32` MegaCore function.
- Symbol File (.sym)—Used to instantiate the PCI interface into a schematic design.
- Constraint file (optional), (.tcl file which can be used to generate .csf and .esf for Quartus II software)—Used to ensure that the PCI MegaCore function achieves PCI timing requirements.

## Parameters

This section describes the `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore functions' features and options available via parameters, which are easily defined using the PCI compiler wizard.

Parameters allow you to customize the PCI MegaCore functions, thus, you can meet specific application requirements. For example, the parameters define read-only and read or write PCI configuration space, as well as setup optional features specific to the Altera PCI MegaCore functions. When generating a MegaCore instance via the PCI compiler wizard, parameters can be customized from default settings to application-specific settings.

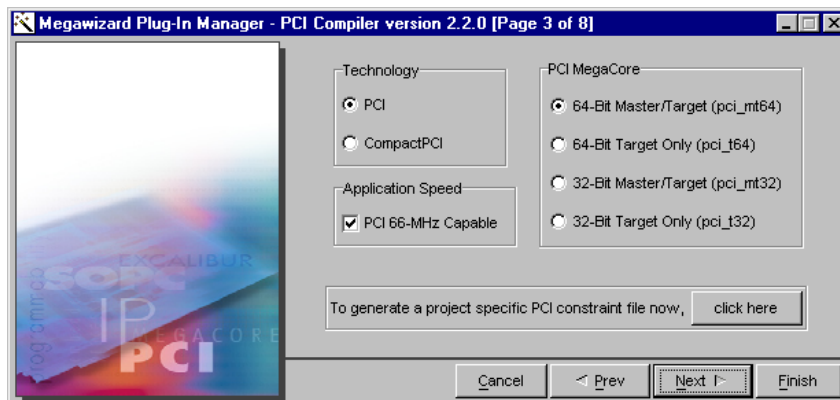


If the wizard is not being utilized, parameters can be set directly in the HDL or graphic design files. For a list of parameter names and descriptions, see [“Appendix D: PCI MegaCore Function Parameters” on page 167](#).

### Application Speed Capability

The `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore functions are capable of operating at PCI clock speeds of up to 66 MHz. Depending on the PCI device speed, the **PCI 66 MHz Capable** option can be enabled or disabled through screen 3 of the PCI compiler wizard. See [Figure 1](#).

Figure 1. Choosing the PCI Application Speed

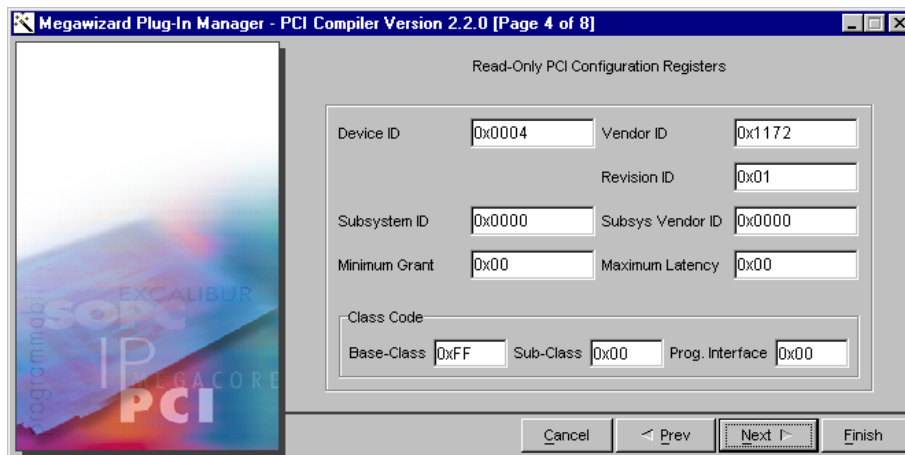


When selected, the **PCI 66 MHz Capable** option enables 66-MHz operation by setting bit 5 of the PCI configuration space status register. For more information on the function of this register, see [“Configuration Registers” on page 50](#).

### Read-Only PCI Configuration Registers

Read-only PCI configuration space registers are defined through the parameters on screen 4 of the PCI compiler wizard. See [Figure 2](#).

Figure 2. Defining Read-Only PCI Configuration Registers





The following read-only PCI configuration space registers are set as parameters through screen 4:

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

The parameters are in hexadecimal format. For information on the functionality of the read-only registers, see “[Configuration Registers](#)” on [page 50](#).

### PCI Base Address Registers (BARs)

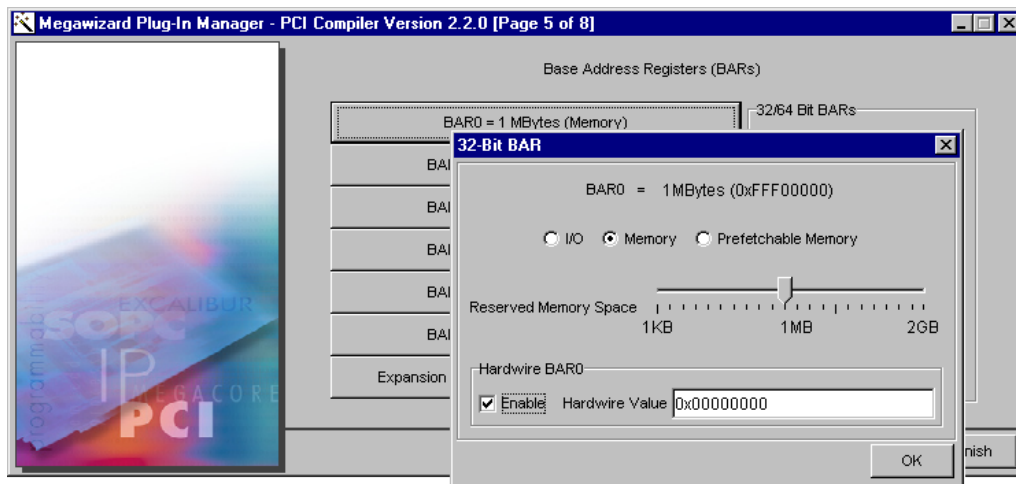
The `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore functions can implement up to six 32-bit BARs, as well as the expansion ROM BAR. The `pci_mt64` and `pci_t64` functions can also implement one 64-bit BAR using either BAR 1 and BAR0, or BAR2 and BAR1 registers.

At least one BAR must be utilized. More than one BAR can be utilized; however, BARs must be used sequentially. 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, Altera 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 is implemented as a read-only value supplied to the MegaCore function through the parameter value. System software cannot overwrite a base address that is hardwired; all other BAR attributes are set normally.

The PCI BAR attributes can be defined through parameters on screen 5 of the PCI compiler wizard. See [Figure 3](#).

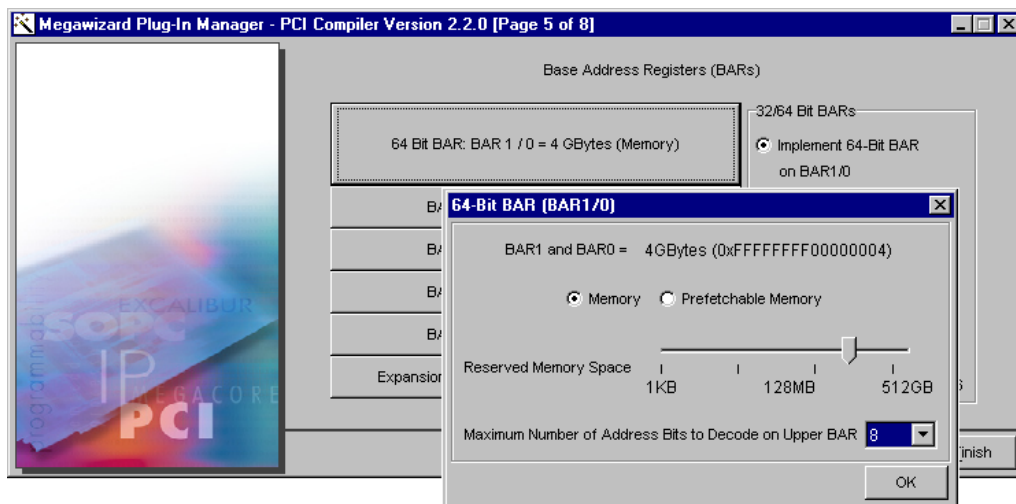
Figure 3. Defining BAR Attributes



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 Altera 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 GBytes, you only need 8 bits of the most significant BAR to be decoded. The maximum number of read/write bits is from 8 to 32. When the maximum number of read/write bits is set to 32, all bits of the most significant BAR will be decoded.

Figure 4 shows some of the steps in setting the attributes of a 64-Bit BAR.

Figure 4. Setting the Number of Address Bits to Decode on the Upper BAR for a 64-Bit BAR



For more information on the function of the BARs, please refer to “[Base Address Registers](#)” on page 58.

## Advanced Features in the pci\_mt64, pci\_mt32, pci\_t64, and pci\_t32 MegaCore Functions

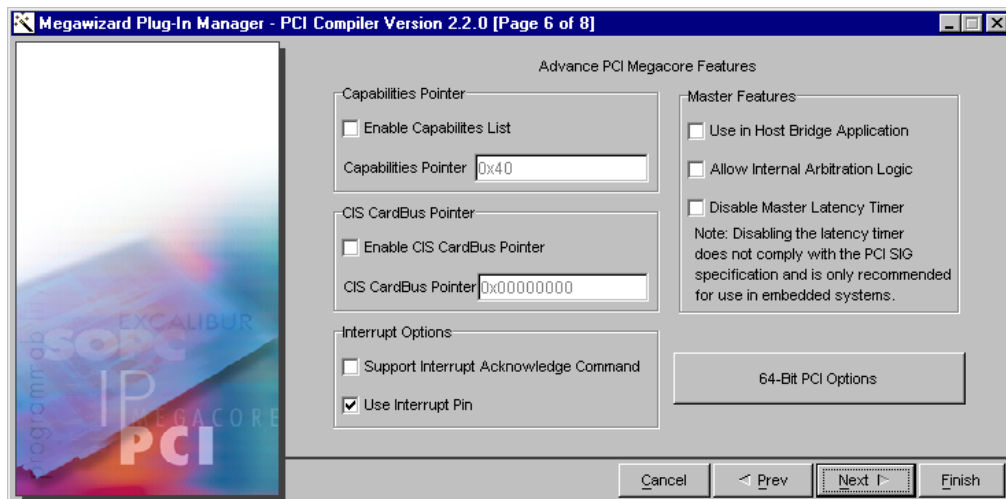
Figure 5 shows the advanced features available with the MegaCore functions (i.e., optional registers, interrupt capabilities, master features, and 64-bit PCI options).

### Optional Registers

The Altera PCI MegaCore functions support two optional read-only registers: the capabilities list pointer register and CIS cardbus pointer register. When used, the value provided via the PCI compiler wizard is stored in these optional registers. When **CompactPCI Technology** is selected in screen 3 of the PCI compiler wizard, the capabilities list pointer register will be enabled with the default value of 40 Hex.

Advanced features in the pci\_mt64, pci\_mt32, pci\_t64, and pci\_t32 MegaCore functions can be enabled in screen 6 of the PCI compiler wizard. See Figure 5.

Figure 5. Using Advanced Features in PCI MegaCore Functions



### Optional Interrupt Capabilities

The Altera 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 (i.e., `intan`) is used by storing a value of 01 Hex in the interrupt pin register. Disabling the interrupt pin results in the interrupt pin register being set to 00. The PCI MegaCore functions also provide the option to respond to the interrupt acknowledge command. When disabled, the MegaCore functions will ignore the interrupt acknowledge command. When enabled, the MegaCore functions respond 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, please refer to the [“Configuration Registers” on page 50](#).

### Optional Master Features

The `pci_mt64` and `pci_mt32` functions also provide some master-specific features. For example, the `pci_mt64` and `pci_mt32` functions can be used as a host bridge application. For more information on using the `pci_mt64` or `pci_mt32` function in a host bridge application, see [“Host Bridge Operation” on page 142](#).

Additionally, the disable master latency timer option allows users 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.



Using the disable master latency timer option violates the PCI specification; and therefore should only be used in embedded applications where the designer can control the entire system configuration. In addition, using the disable master latency timer option can result in increased latency for other master devices on the system. If the increased latency will result in undesirable effects, this option should not be used.

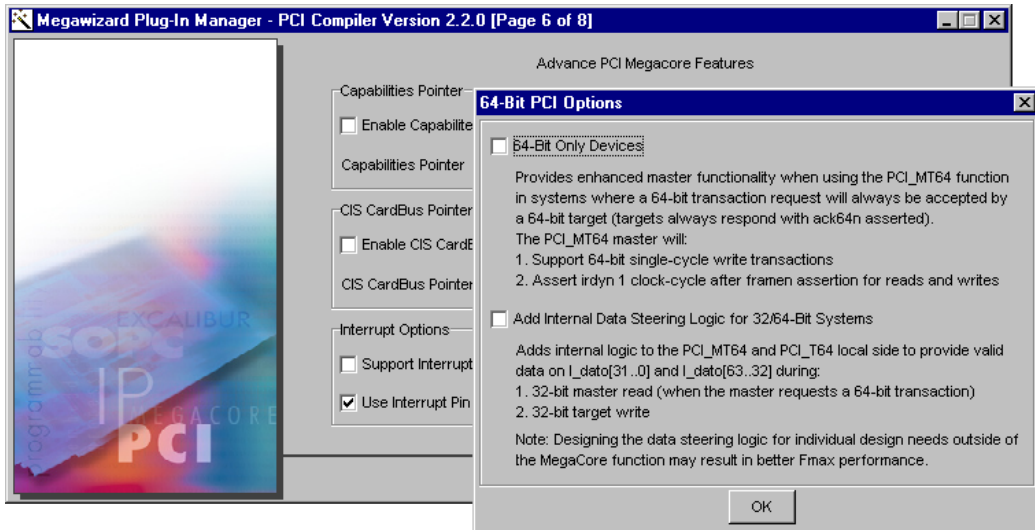
### 64-Bit PCI Options

The `pci_mt64` and `pci_t64` functions provide 64-bit specific features. Choosing the **64-Bit PCI Options** button provides access to two additional 64-bit options. See [Figure 6](#).



See “[Appendix C: 64-Bit Options for the `pci\_mt64` and `pci\_t64` MegaCore Functions](#)” on page 159 for more information on these options.

Figure 6. 64-Bit PCI Options with `pci_mt64` and `pci_t64` Functions



The **64-Bit Only Devices** option provides enhanced master device functionality when using the `pci_mt64` function in systems where a 64-bit transaction request will always be accepted by a 64-bit target device. (Target devices always respond with `ack64n` asserted.) The `pci_mt64` master will:

- Support 64-bit single-cycle write transactions
- Assert `irdyn` one clock-cycle after the assertion of `framen` for read and write transactions



The **64-Bit Only Devices** option should only be used in embedded applications where the designer controls the entire system configuration.

The **Add Internal Data Steering Logic for 32/64-Bit Systems** option adds internal logic to the `pci_mt64` and `pci_t64` local side to provide valid data on `l_dato[31..0]` and `l_dato[63..32]` buses during the following transactions:

- 32-bit master read (when the master requests a 64-bit transaction)
- 32-bit target write

Enabling the **Add Internal Data Steering Logic for 32/64-Bit Systems** option provides full backwards compatibility to the `pci_mt64` and `pci_t64` functions prior to version 2.0.0. If the **Add Internal Data Steering Logic for 32/64-Bit Systems** option is not used, the data steering logic should be added to the local side application. Adding the data steering logic to the local side application will most likely result in less duplicate logic and faster system performance.

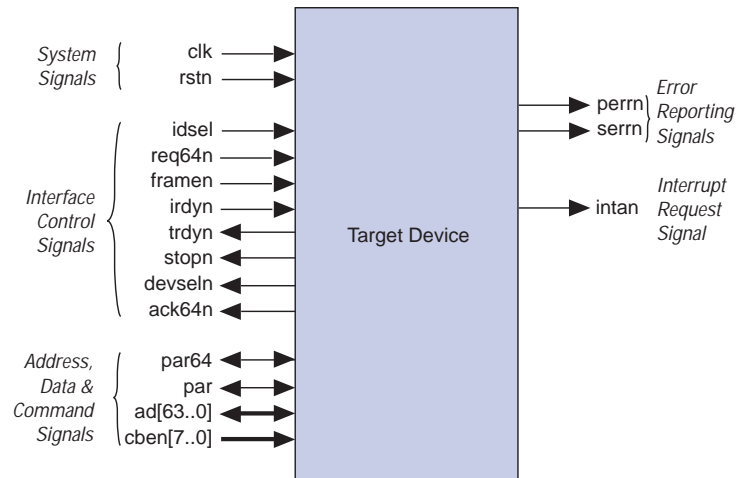
## Functional Description

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.

### Target Device Signals & Signal Assertion

**Figure 7** 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 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 7. Target Device Signals



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

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

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

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

**Table 10** shows the control signal combinations possible on the PCI bus during a PCI transaction. The MegaCore function processes the PCI signal assertion from the local side. Therefore, the MegaCore function only drives the control signals per the **PCI Local Bus Specification, Revision 2.2**. The local-side application can force retry, disconnect, abort, successful data transfer, and target wait state cycles to appear on the PCI bus by driving the `lt_rdyn`, `lt_discn`, and `lt_abortn` signals to certain values. See “**Target Transaction Terminations**” on page 96 for more details.

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.

**Table 10. Control Signal Combination Transfer**

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

**Notes:**

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

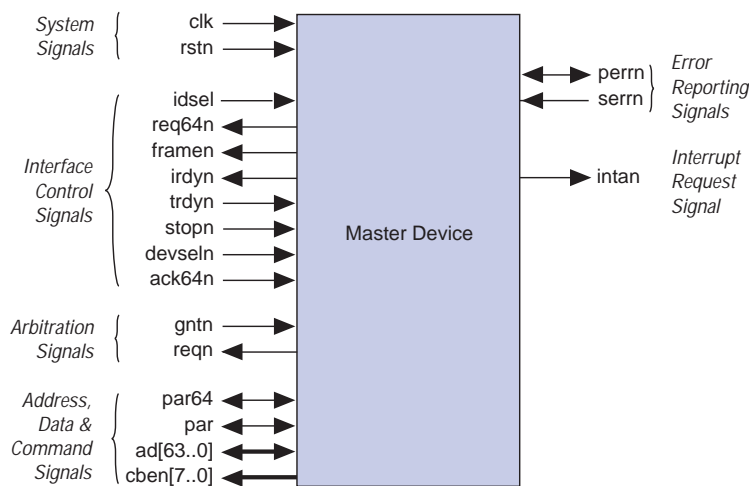


The `pci_mt64`, `pci_t64`, `pci_mt32`, and `pci_t32` functions support unlimited burst access cycles. Therefore, they can achieve a throughput from 132 MBps (for 32-bit, 33-MHz transactions) up to 528 MBps (for 64-bit, 66-MHz transactions). However, the **PCI Local Bus Specification, Revision 2.2** does not recommend bursting beyond 16 data cycles because of the latency of other devices that share the bus. You should be aware of the trade-off between bandwidth and increased latency.

## Master Device Signals & Signal Assertion

**Figure 8** illustrates the PCI-compliant master device signals that connect to the PCI bus. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the PCI MegaCore function operating as a master on the PCI bus. **Figure 8** shows all master signals; the 64-bit extension signals, including `req64n`, `ack64n`, `par64`, `ad[63..32]`, and `cben[7..0]`, are not implemented in the `pci_mt32` function.

**Figure 8. 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 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[63..0]` and the transaction command on `cben[7..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. A target device is determined by the decoding of the address and command signals presented on the PCI bus during the address phase of the transaction. The target device drives the control signals `devseln`, `trdyn`, and `stopn` to indicate one of the following conditions:

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

Table 10 on page 46 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 signaled by the target, master abort, or latency timer expiration. See “**Abnormal Master Transaction Termination**” on page 141 for more details.

In addition to single-cycle and burst 32-bit transactions, the local side master can request 64-bit transactions by asserting the `lm_req64n` signal. In 64-bit transactions, the `pci_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. See “**Master Mode Operation**” on page 151 for more information on 64-bit master transactions.

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

## PCI Bus Commands

**Table 1** shows the PCI bus commands that can be initiated or responded to by Altera's PCI MegaCore functions.

<i>Table 1. PCI Bus Command Support Summary</i>			
cben[3..0] Value	Bus Command Cycle	Master	Target
0000	Interrupt acknowledge	Ignored	Yes (1)
0001	Special cycle	Ignored	Ignored
0010	I/O read	Yes	Yes
0011	I/O write	Yes	Yes
0100	Reserved	Ignored	Ignored
0101	Reserved	Ignored	Ignored
0110	Memory read	Yes	Yes
0111	Memory write	Yes	Yes
1000	Reserved	Ignored	Ignored
1001	Reserved	Ignored	Ignored
1010	Configuration read	Yes	Yes
1011	Configuration write	Yes	Yes
1100	Memory read multiple (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:

- (1) Interrupt acknowledge support can be enabled through the 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 1](#).
- (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. See [Table 1](#).

The PCI functions respond to standard memory read/write, cache memory read/write, I/O read/write, and configuration read/write commands. The bus commands are discussed in greater detail in “[Target Mode Operation](#)” on page 65 and “[Master Mode Operation](#)” on page 105.

In master mode, the `pci_mt64` and the `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 MRL and MWI commands. It is the responsibility of the local-side interface to ensure that this requirement is not violated. Additionally, it is the responsibility of the local-side interface to ensure that proper address and byte enable combinations are used during I/O read/write cycles.

## Configuration Registers

Each logical PCI bus device includes a block of 64 configuration DWORDS reserved for the implementation of its configuration registers. The format of the first 16 DWORDS is defined by the PCI Special Interest Group (PCI SIG) *PCI Local Bus Specification, Revision 2.2* and the *Compliance Checklist, Revision 2.2*. 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 Altera’s PCI functions.

[Table 2](#) shows the defined 64-byte configuration space. The registers within this range are used to identify the device, control PCI bus functions, and provide PCI bus status. The shaded areas indicate registers that are supported by Altera’s PCI functions.

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

**Table 3** summarizes the supported configuration registers address map. Unused registers produce a zero when read, and they ignore a write operation. Read/write refers to the status at runtime, i.e., from the perspective of other PCI bus agents. You can set some of the read-only registers when creating a custom PCI design by setting the MegaCore function parameters through the PCI compiler wizard. For example, you can change the device ID register value from the default value through screen 3 of the PCI compiler wizard. The specified default state is defined as the state of the register when the PCI bus is reset.

*Table 3. Supported Configuration Registers Address Map*

Address Offset (Hex)	Range Reserved (Hex)	Bytes Used/ Reserved	Read/Write	Mnemonic	Register Name
00	00-01	2/2	Read	ven_id	Vendor ID
02	02-03	2/2	Read	dev_id	Device ID
04	04-05	2/2	Read/write	comd	Command
06	06-07	2/2	Read/write	status	Status
08	08-08	1/1	Read	rev_id	Revision ID
09	09-0B	3/3	Read	class	Class code
0C	0C-0C	1/1	Read/write	cache	Cache line size (1)
0D	0D-0D	1/1	Read/write	lat_tmr	Latency timer (1)
0E	0E-0E	1/1	Read	header	Header type
10	10-13	4/4	Read/write	bar0	Base address register zero
14	14-17	4/4	Read/write	bar1	Base address register one
18	18-1B	4/4	Read/write	bar2	Base address register two
1C	1C-1F	4/4	Read/write	bar3	Base address register three
20	20-23	4/4	Read/write	bar4	Base address register four
24	24-27	4/4	Read/write	bar5	Base address register five
28	28-2B	4/4	Read	cardbus_ptr	CardBus CIS pointer
2C	2C-2D	2/2	Read	sub_ven_id	Subsystem vendor ID
2E	2E-2F	2/2	Read	sub_id	Subsystem ID
30H	30-33	4/4	Read/write	exp_rom_bar	Expansion ROM BAR
34H	34-35	1/1	Read	cap_ptr	Capabilities pointer
3C	3C-3C	1/1	Read/write	int_ln	Interrupt line
3D	3D-3D	1/1	Read	int_pin	Interrupt pin
3E	3E-3E	1/1	Read	min_gnt	Minimum grant (1)
3F	3F-3F	1/1	Read	max_lat	Maximum latency (1)

**Note:**

(1) These registers are supported by the `pci_mt64` and `pci_mt32` 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 1172 Hex. However, by setting the VEND\_ID value through the wizard, you can change the value of the vendor ID register to their PCI SIG-assigned vendor ID value. See [Table 4](#).

Table 4. Vendor ID Register Format			
Data Bit	Mnemonic	Read/Write	Definition
15..0	ven_id	Read	PCI vendor ID

### Device ID Register

Device ID is a 16-bit read-only register that identifies the device type. The value of this register is assigned by the manufacturer. The default value of the device ID register is 0 Hex. You can change the value of the device ID register through the wizard. See [Table 5](#).

Table 5. Device ID Register Format			
Data Bit	Mnemonic	Read/Write	Definition
15..0	dev_id	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. See [Table 6](#).

*Table 6. Command Register Format*

Data Bit	Mnemonic	Read/Write	Definition
0	io_ena	Read/write	I/O access enable. When high, <code>io_ena</code> lets the function respond to the PCI bus I/O accesses as a target.
1	mem_ena	Read/write	Memory access enable. When high, <code>mem_ena</code> lets the function respond to the PCI bus memory accesses as a target.
2	mstr_ena	Read/write	Master enable. When high, <code>mstr_ena</code> 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 <code>mwi_ena</code> 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, <code>perr_ena</code> enables the function to report parity errors via the <code>perrn</code> output.
7	Unused	—	—
8	serr_ena	Read/write	System error enable. When high, <code>serr_ena</code> allows the function to report address parity errors via the <code>serrn</code> output. However, to signal a system error, the <code>perr_ena</code> bit must also be high.
15..9	Unused	—	—

## Status Register

Status is a 16-bit register that provides the status of bus-related events. Read transactions from the status register behave normally. However, 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 4000 Hex to the status register clears bit 14 and leaves the rest of the bits unchanged. The default value of the status register is 0400 Hex. See [Table 7](#).



Table 7. Status Register Format

Data Bit	Mnemonic	Read/Write	Definition
3..0	Unused	–	Reserved.
4	cap_list_ena	Read	Capabilities list enable. This bit is read only and is set by the user when enabling the <b>Capabilities List Pointer</b> through the wizard. When set, this bit enables the capabilities list pointer register at offset 34 Hex. See “ <a href="#">Capabilities Pointer</a> ” on page 63 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 MegaCore function can run at either 66 MHz or 33 MHz depending on the device used. You can set this bit to 1 by enabling the PCI 66MHz Capable option in the 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").
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 functions). For Altera PCI MegaCore functions, the default value of the revision ID register is the revision number of the function. See [Table 8](#). You can change the value of the revision ID register through the wizard.

*Table 8. Revision ID Register Format*

Data Bit	Mnemonic	Read/Write	Definition
7..0	rev_id	Read	PCI revision ID

## Class Code Register

Class code is a 24-bit read-only register divided into three sub-registers: base class, sub-class, and programming interface. Refer to the **PCI Local Bus Specification, Revision 2.2** for detailed bit information. The default value of the class code register is FF0000 Hex. You can change the value of the `class_code` register through the PCI compiler wizard. See [Table 9](#).

*Table 9. Class Code Register Format*

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

## Cache Line Size Register

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



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

*Table 10. Cache Line Size Register Format*

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

## Latency Timer Register

The latency timer register is an 8-bit register with bits 2, 1, and 0 tied to ground. The register defines the maximum amount of time, in PCI bus clock cycles, that the PCI 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. The default value of the latency timer register is 00 Hex. See [Table 11](#).



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

*Table 11. Latency Timer Register Format*

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

## Header Type Register

Header type is an 8-bit read-only register that identifies the PCI function as a single-function device. The default value of the header type register is 00 Hex. See [Table 12](#).

*Table 12. Header Type Register Format*

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

## Base Address Registers

The PCI function supports up to six BARs. Each base address register (BAR $n$ ) has identical attributes. You can control the number of BARs that are instantiated in the function by enabling BARs on an individual basis through the wizard. 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). Each BAR 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
- Sets the reserved memory as prefetchable or non-prefetchable
- Size of memory or I/O address space reserved for the BAR



When compiling the PCI 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 2.2**. Bit 0 of each BAR is read only, and is used to indicate whether the reserved address space is memory or I/O. BARs that map to memory space must hardwire bit 0 to 0, and BARs that map to I/O space must hardwire bit 0 to 1. Depending on the value of bit 0, the format of the BAR changes. You can set the type of BAR through the PCI compiler wizard.

In a memory BAR, bits 2 and 1 indicate the location of the address space in the memory map. You can control the location of specific BAR addresses (i.e., whether they are mapped in 32- or 64-bit address space) through options in the PCI compiler 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 PCI compiler wizard, bit 3 of the corresponding BAR $n$  parameter will be updated. See [Table 13](#).

Table 13. Memory BAR Format

Data Bit	Mnemonic	Read/Write	Definition
0	mem_ind	Read	Memory indicator. The <code>mem_ind</code> bit indicates that the register maps into memory address space. This bit must be set to 0 in the <code>BARn</code> parameter.
2..1	mem_type	Read	Memory type. The <code>mem_type</code> 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 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 <code>pre_fetch</code> 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 `BARn` parameter value accordingly. The parameter value `BARn` defines the number of read/write bits instantiated in the corresponding BAR (See Section 6.2.5 in the **PCI Local Bus Specification, Revision 2.2**). The number of read/write bits instantiated in a BAR is indicated by the number of 1s in the corresponding `BARn` value starting from bit 31. The `BARn` parameter should contain 1s from bit 31 down to the required bit without any 0s in between (e.g., "FF000000" Hex is legal, but "FF700000" Hex is not). The PCI compiler wizard does not offer options that set the `BARn` 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 `BARn` 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  $\text{BAR}_n$  parameter accordingly. The PCI Local Bus Specification, Revision 2.2 prevents any single I/O BAR from reserving more than 256 bytes of I/O space. See [Table 14](#).

**Table 14. I/O Base Address Register Format**

Data Bit	Mnemonic	Read/Write	Definition
0	io_ind	Read	I/O indicator. The io_ind bit indicates that the register maps into I/O address space. This bit must be set to 1 in the $\text{BAR}_n$ parameter.
1	Reserved	—	—
31..2	bar	Read/write	Base address registers.

In some applications, one or more BARs must be hardwired. The 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  $\text{BAR}_n$  basis and sets the corresponding parameters accordingly. When using the hardwire BAR feature, the corresponding  $\text{BAR}_n$  attributes must indicate the appropriate BAR settings, such as size and type of address space.



When implementing a hardwire BAR, the corresponding BAR registers become read-only. A configuration write to the hardwired BAR will proceed normally. However, a configuration read of hardwired BAR registers will return the value set in the hardwire  $\text{BAR}_n$  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 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 15](#) shows this register's format. For more information on the CardBus CIS pointer register, refer to the *PCMCIA Specification, Version 2.10*.

Table 15. CIS Pointer Register Format			
Data Bit	Mnemonic	Read/Write	Definition
0..2	adr_space_ind	Read	Address space indicator. The value of these bits indicates that the CIS pointer register is pointing to one of the following spaces: configuration space, memory space, or expansion ROM space.
3..27	adr_offset	Read	Address space offset. This value gives the address space's offset indicated by the address space indicator.
31..28	rom_im	Read	ROM image. These bits are the uppermost bits of the address space offset when the CIS pointer register is pointing to an expansion ROM space.

### Subsystem Vendor ID Register

Subsystem vendor ID is a 16-bit read-only register that identifies add-in cards from different vendors that have the same functionality. The value of this register is assigned by the PCI SIG. See [Table 16](#). The default value of the subsystem vendor ID register is 0000 Hex. However, you can change the value through the wizard.

Table 16. 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. See [Table 17](#). The default value of the subsystem ID register is 0000 Hex. However, you can change the value through the wizard.

*Table 17. Subsystem ID Register Format*

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

## Expansion ROM Base Address Register

The expansion ROM base address register contains a 32-bit hexadecimal number that defines the base address and size information of the expansion ROM. You can instantiate the expansion ROM BAR through the wizard; the PCI function's parameters will be set accordingly. The expansion ROM BAR functions exactly like a 32-bit BAR, except that the encoding of the bottom bits is different. Bit 0 in the register is a read/write and is used to indicate whether or not the device accepts accesses to its expansion ROM. You can disable the expansion ROM address space by setting bit 0 to 0. You can enable the address decoding of the expansion ROM by setting bit 0 to 1. The upper 21 bits correspond to the upper 21 bits of the expansion ROM base address. The amount of address space a device requests must not be greater than 16 Mbytes. The expansion ROM BAR is formatted per the *PCI Local Bus Specification, Revision 2.2*. See [Table 18](#).

*Table 18. 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 MegaCore functions allow you to set a default expansion ROM base address using the hardwire option in the 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 BAR registers 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, see appendix H in the *PCI Local Bus Specification, Revision 2.2*. The address location of the pointer must be 40 Hex or greater, and each capability must be within DWORD boundaries. See [Table 19](#).

Table 19. Interrupt Line Register 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 40 Hex are transferred to the local side of the 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 FF Hex. See [Table 20](#).



The interrupt pin can be enabled or disabled in the PCI compiler wizard. The interrupt pin register will be set to 00 Hex if the interrupt option is disabled in the wizard.

*Table 20. Interrupt Line Register Format*

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

## Interrupt Pin Register

The interrupt pin register is an 8-bit read-only register that defines the PCI function PCI bus interrupt request line to be `intan`. The default value of the interrupt pin register is 01 Hex. See [Table 21](#).

*Table 21. Interrupt Pin Register Format*

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

## Minimum Grant Register

The minimum grant register 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. See [Table 22](#).

*Table 22. Minimum Grant Register Format*

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

## Maximum Latency Register

The maximum latency register is an 8-bit read-only register that defines the frequency in which the function would like to gain access to the PCI bus. See [Table 23](#). You can set this register through the wizard.

*Table 23. Maximum Latency Register Format*

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

## Target Mode Operation

This section describes all supported target transactions for the PCI functions. Although this section includes waveform diagrams showing typical PCI cycles in target mode for the `pci_mt64` function, these waveforms are also applicable for the `pci_mt32`, `pci_t64`, and `pci_t32` functions. The `pci_mt64` and `pci_t64` MegaCore functions support both 32-bit and 64-bit transactions. [Table 24](#) lists the PCI and local side signals that apply for each PCI function.

*Table 24. PCI MegaCore Function Signals (Part 1 of 2)*

PCI Signals	pci_mt64	pci_t64	pci_mt32	pci_t32
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	✓	✓		
trdyn	✓	✓	✓	✓
stopn	✓	✓	✓	✓
perrn	✓	✓	✓	✓
serrn	✓	✓	✓	✓
intan	✓	✓	✓	✓

Table 24. PCI MegaCore Function Signals (Part 2 of 2)

PCI Signals	pci_mt64	pci_t64	pci_mt32	pci_t32
Local side signals				
l_adi[63..0]	✓	✓	l_adi[31..0]	l_adi[31..0]
l_cbeni[3..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_cmndo[3..0]	✓	✓	✓	✓
l_ldat_ackn	✓	✓		
l_hdat_ackn	✓	✓		
Target local side				
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				
lm_req32n	✓		✓	
lm_req64n	✓			
lm_lastn	✓		✓	
lm_rdyn	✓		✓	
lm_adr_ackn	✓		✓	
lm_ackn	✓		✓	
lm_dxfrn	✓		✓	
lm_tsr[9..0]	✓		✓	

The pci\_mt64 and pci\_t64 functions support the following 64-bit target memory transactions:

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

Each PCI function supports the following 32-bit transactions:

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

Memory transactions can be single-cycle or burst. In target mode, the MegaCore function supports an unlimited length of zero-wait-state memory burst read or write. In a read transaction, data is transferred from the local side to the PCI master. In a write transaction, data is transferred from the PCI master to the local-side device. A memory transaction can be terminated by either the PCI master or the local-side device. The local-side device can terminate the memory transaction using one of three types of terminations: retry, disconnect, or target abort. “[Target Transaction Terminations](#)” on page 96 describes how to initiate the different types of termination.



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

I/O transactions are always single-cycle 32-bit transactions. Therefore, the MegaCore function handles them like single-cycle memory commands. Any of the six BARs in the PCI functions can be configured to reserve I/O space. See “[Base Address Registers](#)” on page 58 for more information on how to configure a specific BAR to be an I/O BAR. Like memory transactions, I/O transactions can be terminated normally by the PCI master, or the local-side device can instruct the MegaCore function to terminate the transactions with a retry or target abort. Because all I/O transactions are single-cycle, terminating a transaction with a disconnect does not apply.

## 64-Bit Target Read Transactions

In target mode, the `pci_mt64` and `pci_t64` functions support two types of 64-bit read transactions:

- Memory single-cycle read
- Memory burst read

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

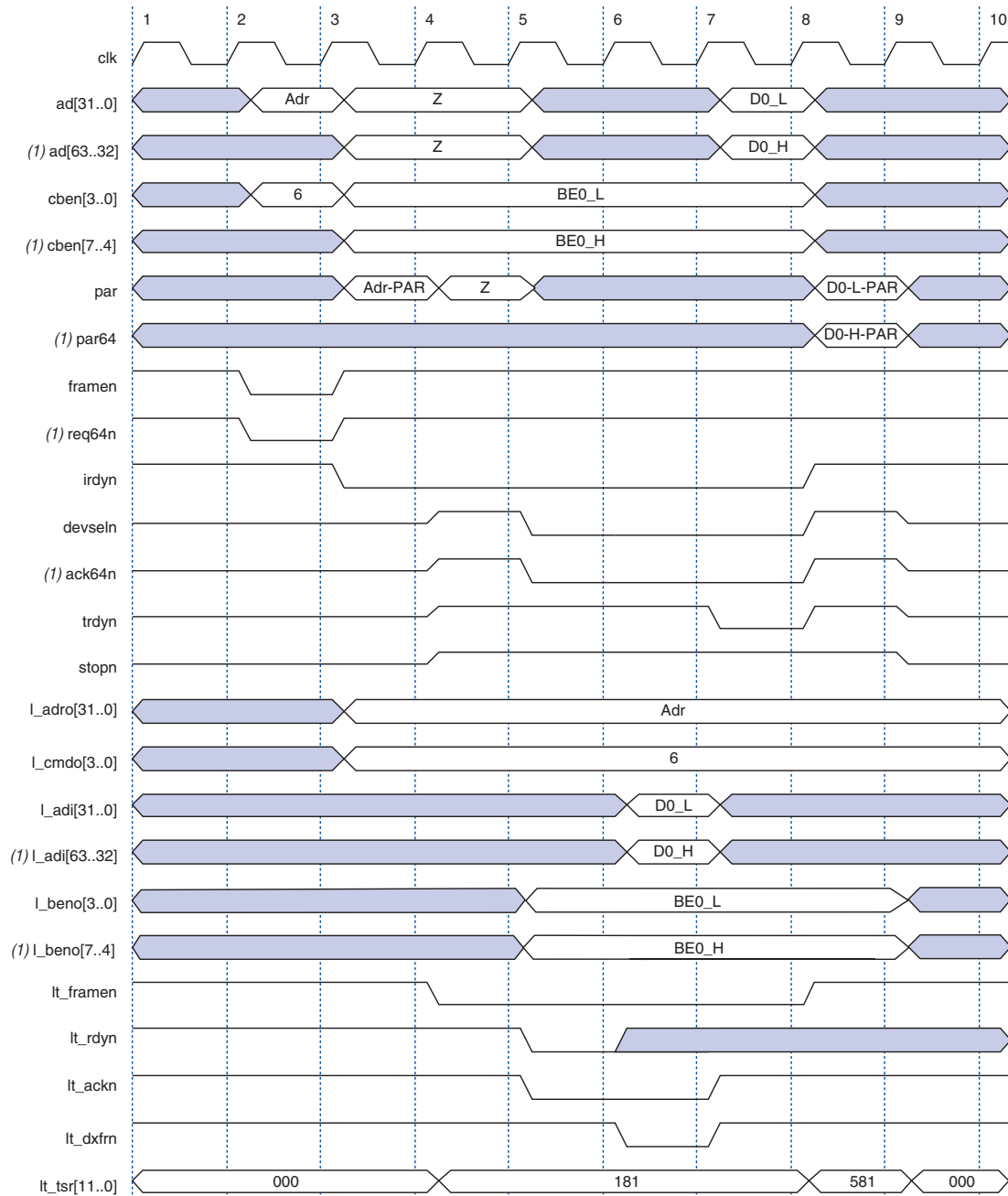
1. The address phase occurs when the PCI master asserts `framen` and `req64n` signals and drives the address and command on `ad[31..0]` and `cben[3..0]`, correspondingly. Asserting the `req64n` signal indicates to the target device that the master device is requesting a 64-bit data transaction.

2. Turn-around cycles on the `ad[63..0]` bus occur during the clock immediately following the address phase. During the turn-around cycles, the PCI master tri-states the `ad[63..0]` bus, but drives correct byte-enables on `cben[7..0]` for the first data phase. This process is necessary because the PCI agent driving the `ad[63..0]` bus changes during read cycles.
3. If the address of the transactions matches one of the base address registers, the `pci_mt64` and `pci_t64` functions turn on the drivers for the `ad[63..0]`, `devseln`, `ack64n`, `trdyn`, and `stopn` signals. The drivers for `par` and `par64` are turned on in the following clock.
4. The `pci_mt64` and `pci_t64` functions drive and assert `devseln` and `ack64n` to indicate to the master device that it is accepting the 64-bit transaction.
5. One or more data phases follow next, depending on the type of read transaction.

#### *64-Bit Single-Cycle Target Read Transaction*

Figure 1 shows the waveform for a 64-bit single-cycle target read transaction. This figure applies to all PCI MegaCore functions, except the 64-bit extension signals as noted for the `pci_mt32` and `pci_t32` functions.

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

**Note:**

- (1) These signals do not apply to `pci_mt32` or `pci_t32` for 32-bit target read transactions. For these transactions, the signals should be ignored.



**Table 25** shows the sequence of events for a single-cycle target read transaction.

**Table 25. Single-Cycle Target Read Transaction (Part 1 of 2)**


Clock Cycle	Event
1	The PCI bus is idle.
2	The address phase occurs.
3	<p>The MegaCore function latches the address and command, and decodes the address to check if it falls within the range of one of its BARs. During clock 3, the master deasserts the <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 target read, this phase is the only data phase in the transaction. The MegaCore function begins to decode the address during clock 3, and if the address falls in the range of one of its BARs, the transaction is claimed.</p> <p>The PCI master tri-states the <code>ad[63..0]</code> bus for the turn-around cycle.</p>
4	<p>If the MegaCore function detects an address hit in clock 3, several events occur during clock 4:</p> <ul style="list-style-type: none"> <li>■ The MegaCore function informs the local-side device that it is going to claim the read transaction by asserting one of the <code>lt_tsr[5..0]</code> signals and <code>lt_framen</code>. In <a href="#">Figure 1</a>, <code>lt_tsr[0]</code> is asserted indicating that a base address register zero hit.</li> <li>■ The MegaCore function drives the transaction command on <code>l_cmdo[3..0]</code> and address on <code>l_adro[31..0]</code>.</li> <li>■ The 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 5.</li> <li>■ <code>lt_tsr[7]</code> is asserted to indicate that the pending transaction is 64-bits.</li> <li>■ <code>lt_tsr[8]</code> is asserted to indicate that the PCI side of the MegaCore function is busy.</li> </ul>
5	The 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 <code>l_adi[63..0]</code> . The MegaCore function also enables the output drivers of the <code>ad[63..0]</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 5, the data transfer does not occur until clock 6.
6	<p><code>lt_rdyn</code> is asserted in clock 5, indicating that valid data is available on <code>l_adi[63..0]</code> in clock 6. The MegaCore function registers the data into its internal pipeline on the rising edge of clock 7. The local side transfer is indicated by the <code>lt_dxfrn</code> signal. The <code>lt_dxfrn</code> signal is low during the clock where a data transfer on the local side occurs.</p> <p> 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.</p>
7	The rising edge of clock 7 registers the valid data from <code>l_adi[63..0]</code> and drives the data on the <code>ad[63..0]</code> bus. At the same time, the MegaCore function asserts the <code>trdyn</code> signal to indicate that there is valid data on the <code>ad[63..0]</code> bus.

Table 25. Single-Cycle Target Read Transaction (Part 2 of 2)

Clock Cycle	Event
8	The 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 MegaCore function drives <code>devseln</code> , <code>ack64n</code> , <code>trdyn</code> , and <code>stopn</code> high during this clock cycle. Additionally, the MegaCore function tri-states the <code>ad[63..0]</code> bus because the cycle is complete. The rising edge of clock 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 8, the 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 MegaCore function informs the local-side device that the transaction is complete by deasserting the <code>lt_tsr[11..0]</code> signals. Additionally, the 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.

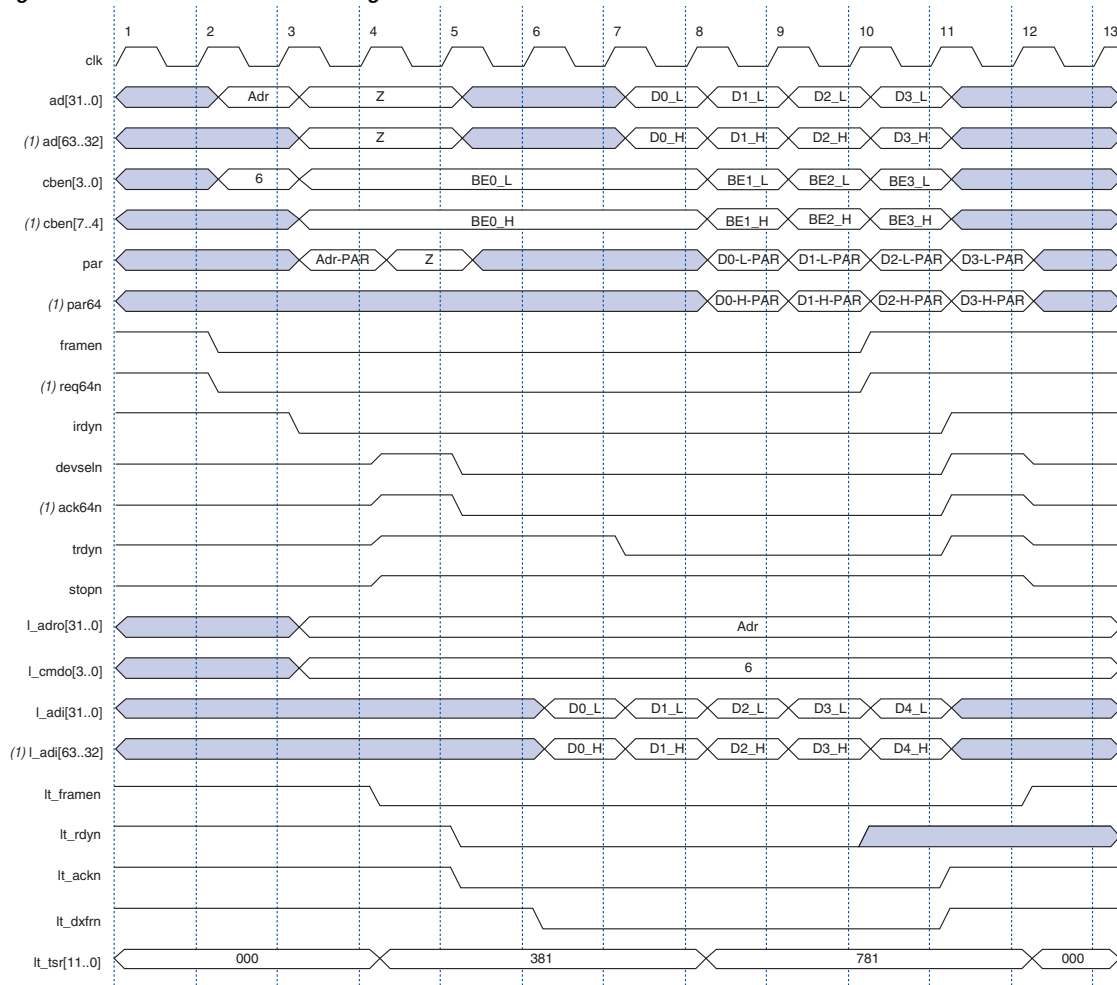


The local-side device must ensure that PCI latency rules are not violated while the MegaCore function waits for data. If the local-side device is unable to meet the latency requirements, it must assert `lt_discn` to request that the MegaCore function terminate the transaction. The PCI target latency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

### 64-Bit Memory Burst Read Transaction

The sequence of events for a burst read transaction is the same as that of a single-cycle read transaction. However, during a burst read transaction, more data is transferred and both the local-side device and the PCI master can insert wait states at any point during the transaction. [Figure 2](#) illustrates a burst read transaction. This figure applies to all PCI MegaCore functions, except the 64-bit extension signals as noted for the `pci_mt32` and `pci_t32` functions.

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

**Note:**

- (1) These signals do not apply to `pci_mt32` or `pci_t32` for 32-bit target read transactions. For these transactions, the signals should be ignored.

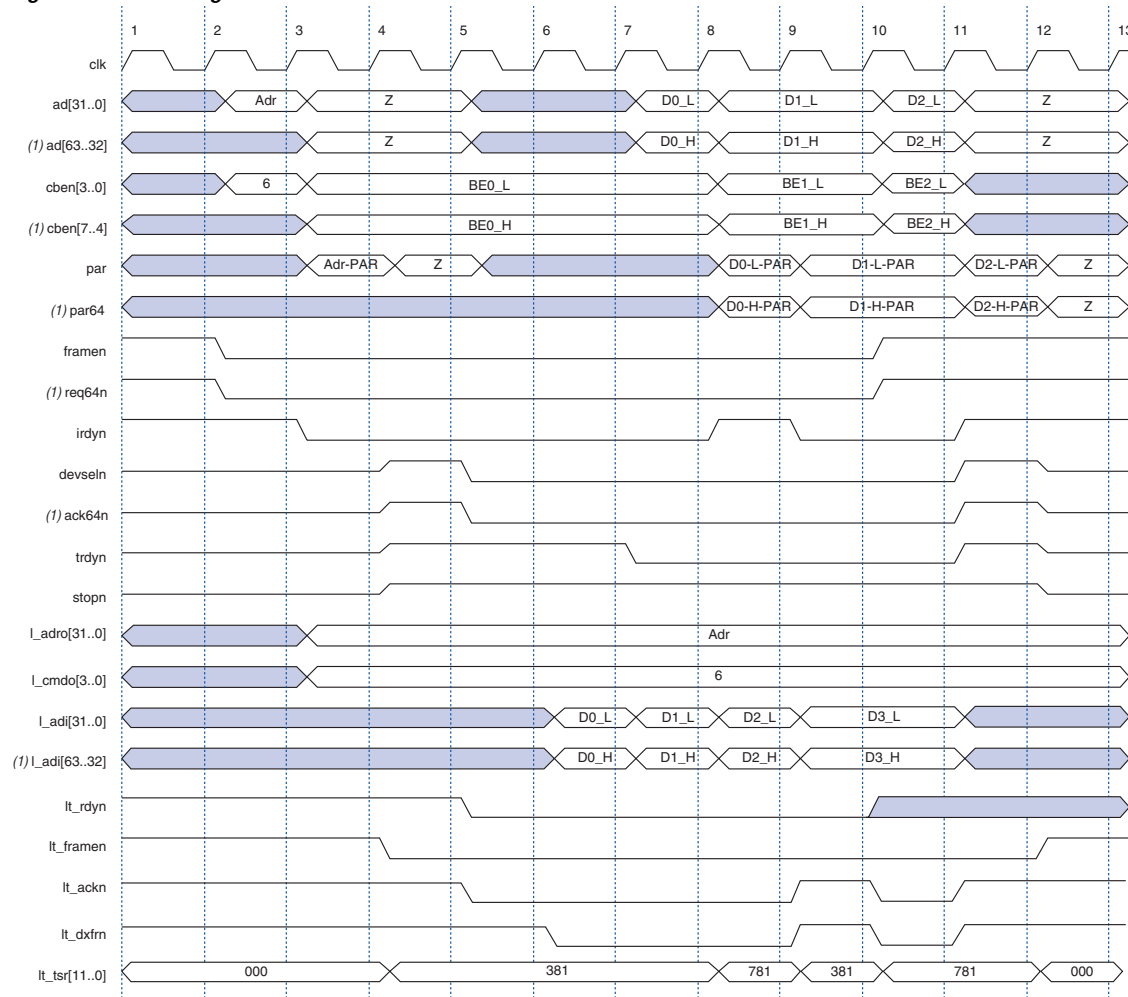
Figure 2 shows a 64-bit zero wait state burst transaction with four data phases. The local side transfers five quad words (QWORDS) in clocks 6 through 10. The PCI side transfers data in clocks 7 through 10. Because of the MegaCore function's zero wait state requirement, the PCI side reads ahead from the local side. Also, because the `l_beno[7..0]` signals are 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 2 shows the `lt_tsr[9]` signal asserted in clock 4 because the master device has `framen` and `irdyn` signals asserted, thus indicating a burst transaction.



A burst transaction can be identified if both the `irdyn` and `framen` signals are asserted at the same time during a 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 clocks. 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.

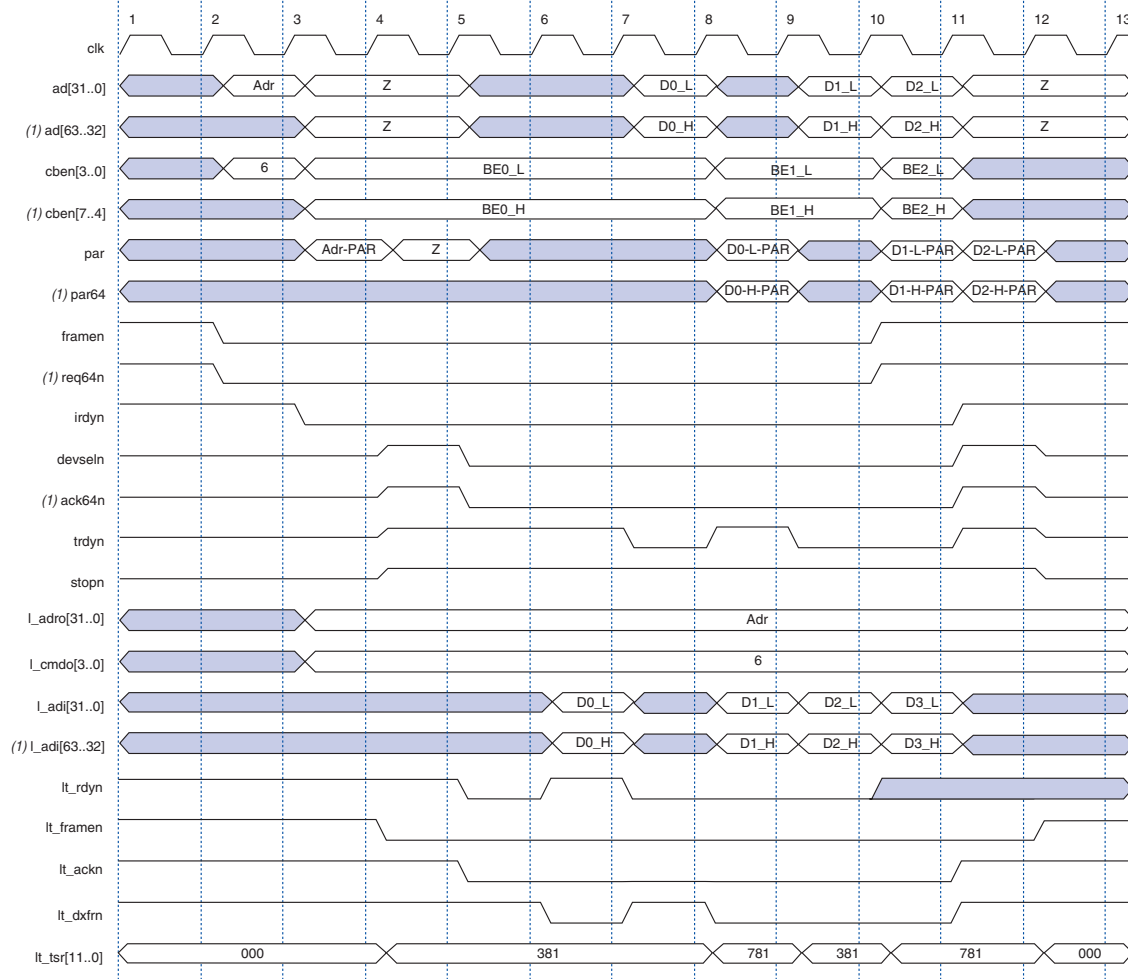
Figure 3 shows the same transaction as in Figure 2 with the PCI bus master inserting a wait state. Figure 3 applies to all PCI MegaCore functions, except the 64-bit extension signals as noted for the `pci_mt32` and `pci_t32` functions. The PCI bus master inserts a wait state by deasserting `irdyn` in clock 8. The effect of this wait state on the local side is shown in clock 9 because `lt_ackn` is deasserted, and as a result `lt_dxfrn` is also deasserted. This situation prevents further data from being transferred on the local side because the internal pipeline of the MegaCore function is full.

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

**Note:**

- (1) These signals do not apply to `pci_mt32` or `pci_t32` for 32-bit target read transactions. For these transactions, the signals should be ignored.

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

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

- (1) These signals do not apply to the pci\_mt32 or pci\_t32 functions for target read transactions. For these transactions, the signals should be ignored.

## 32-Bit Target Read Transactions

The PCI MegaCore functions respond to three types of 32-bit target read transactions:

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

### 32-Bit Memory Read Transactions

For all MegaCore functions, 32-bit memory read transactions are either single-cycle or burst. For the `pci_mt32` and `pci_t32` functions, the waveforms for 32-bit memory read transactions are described in [Figures 1 through 4](#), excluding the 64-bit extension signals as noted. For 32-bit memory read transactions, the `pci_mt64` and `pci_t64` functions always assume a 64-bit local side. The `pci_mt64` and `pci_t64` 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. In a memory read cycle, `pci_mt64` and `pci_t64` assert both `l_ldat_ackn` and `l_hdat_ackn` to indicate that data is transferred 64 bits at a time on the local side. The `pci_mt64` and `pci_t64` functions decode whether the low or high DWORD is addressed by the master, based on the starting address of the transaction:

- If the address of the transaction is a QWORD boundary (`ad[2..0] == B"000"`), the first DWORD transferred to the PCI side is the low DWORD, and `pci_mt64` or `pci_t64` assert both `l_ldat_ackn` and `l_hdat_ackn`.
- 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 of the first 64-bit data phase. The low DWORD of the first 64-bit data phase is not transferred to the PCI side. After the 64-bit data phase, the low DWORD of the following phases is transferred to the PCI side before the high DWORD, followed by the high DWORD.

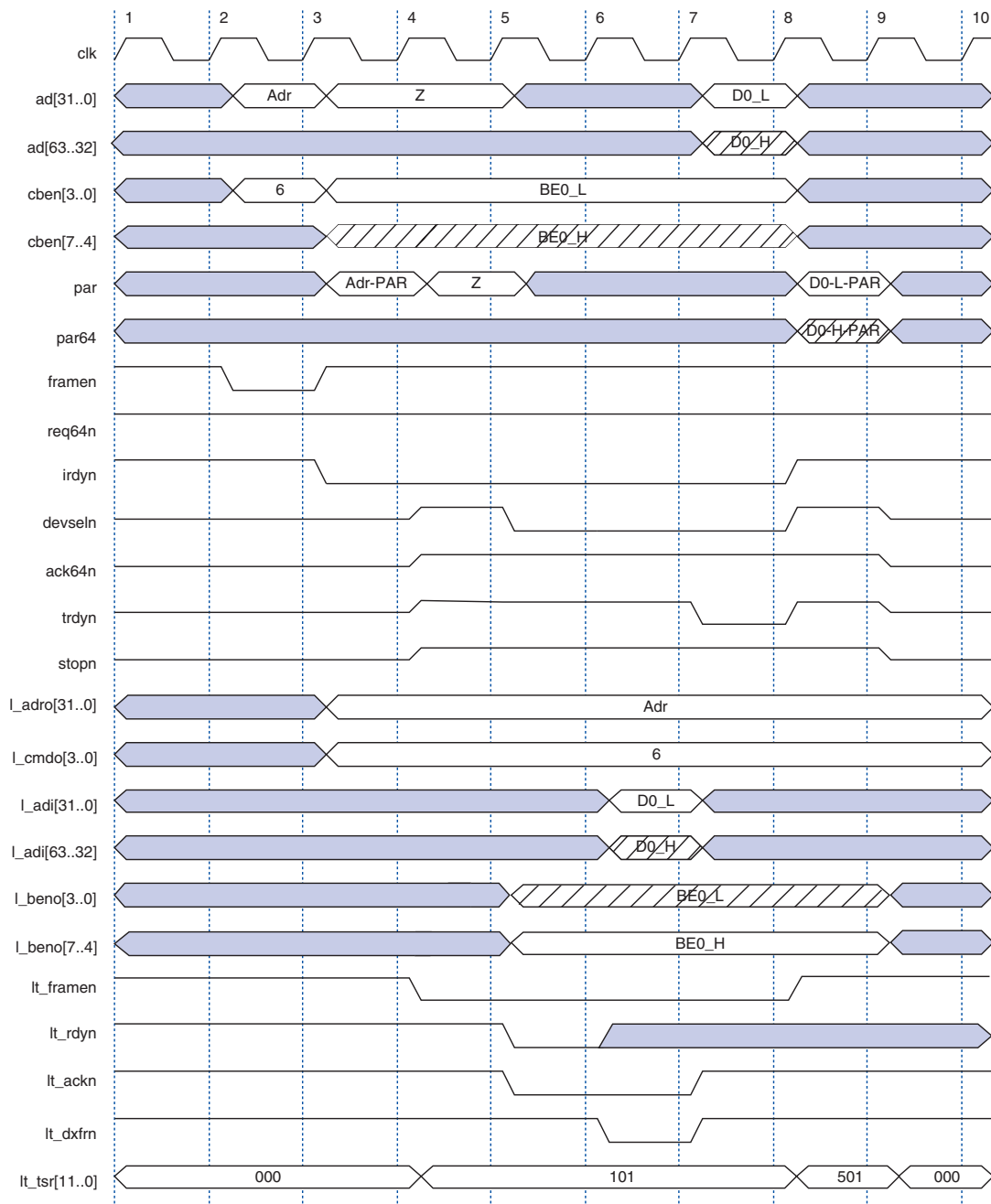
[Figure 5](#) shows a 32-bit single-cycle memory read transaction, which applies to the `pci_mt64` and `pci_t64` functions. Refer to [Figure 1](#) for the description of a 32-bit single-cycle memory read transaction using the `pci_mt32` and `pci_t32` functions.

The sequence of events in [Figure 5](#) is exactly the same as in [Figure 1](#), except for the following cases:

- During the address phase (clock 3), the master does not assert `req64n` because the transaction is 32 bits.
- The `pci_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 5](#) shows that the local side transfers a full QWORD in clock 6. In clock 7, the `pci_mt64` and `pci_t64` functions transfer a full QWORD, however, only the least significant DWORD is accepted by the PCI bus master. The `pci_mt64` and `pci_t64` functions drive the correct parity value on the `par64` signal in clock 8.

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



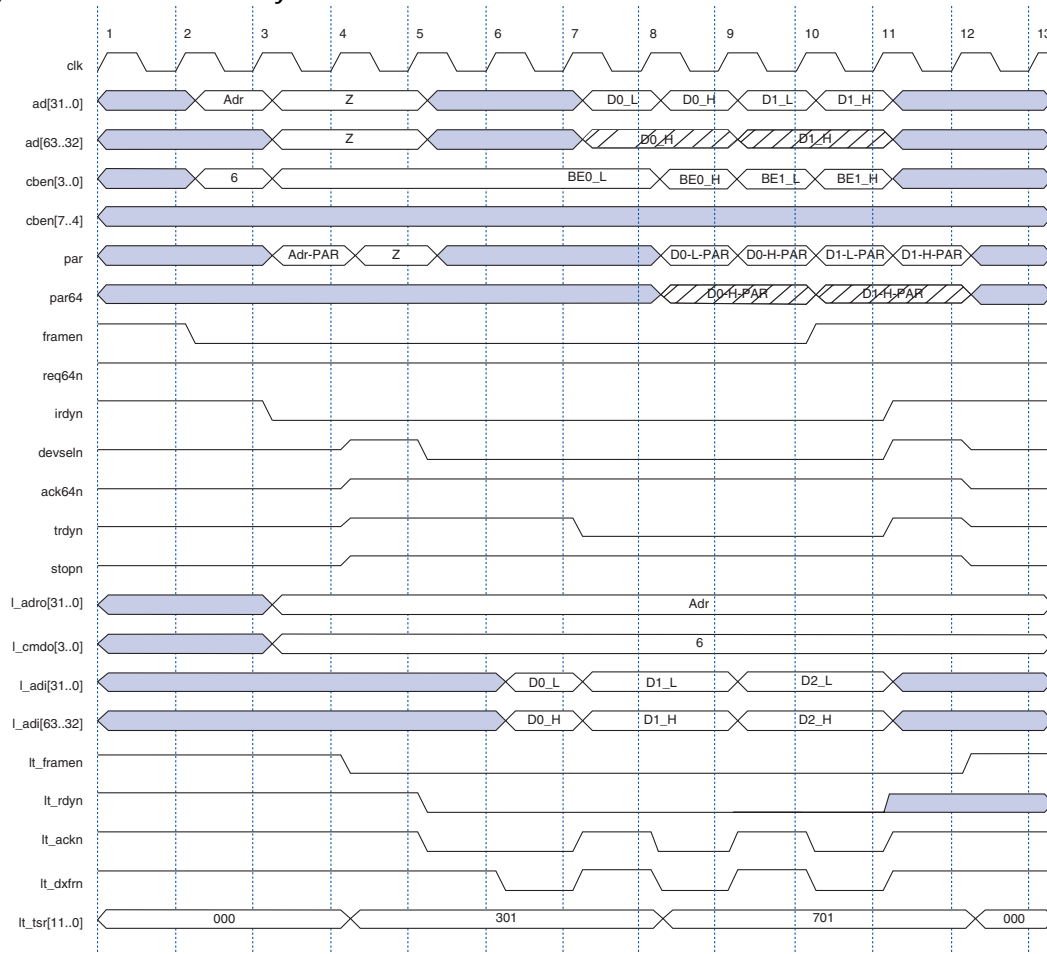




The `pci_mt64` and the `pci_t64` functions always transfer 64-bit data on the local side. In a 32-bit single-cycle memory read transaction, only the least significant DWORD is transferred to the PCI master. Therefore, the local side is only required to transfer the least significant DWORD in a 32-bit single-cycle transaction. See [Figure 5](#).

[Figure 6](#) shows a 32-bit burst memory read transaction. This figure only applies to the `pci_mt64` and `pci_t64` functions. For `pci_mt32` and `pci_t32`, [Figure 2](#) reflects the waveforms for a 32-bit burst read transaction, excluding the 64-bit extension signals as noted. The events in [Figure 6](#) are the same as in [Figure 2](#). The main difference between the two is that a 64-bit transfer takes one clock on the local side, but requires two clocks on the PCI side. Therefore, the function automatically inserts local wait states in clocks 7 and 9 to temporarily suspend the local transfer allowing sufficient time for the data to be transferred on the PCI side. In [Figure 6](#), `lt_tsr[7]` is not asserted and `lt_tsr[9]` is asserted indicating that the transaction is a 32-bit burst. If the local side cannot handle 32-bit burst transactions, it can disconnect after the first local transfer.

Figure 6. 32-Bit Burst Memory Read Transaction



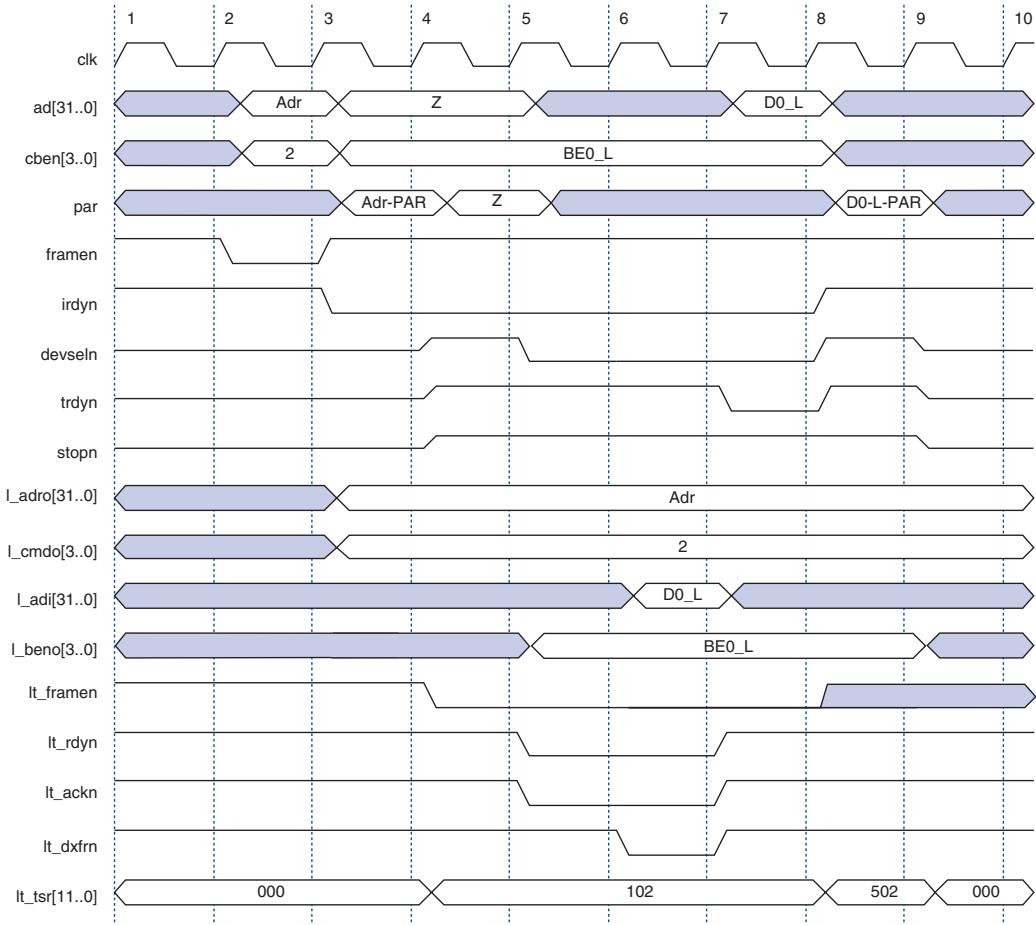
### I/O Read Transaction

I/O read transactions by definition are 32 bits. [Figure 7](#) shows a sample I/O read transaction. This figure applies to all PCI MegaCore functions. 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_cmndo[3..0]` bus. In [Figure 7](#), `lt_tsr[11..0]` indicates that the base address register that detected the address hit is BAR1. Additionally, during an I/O transaction `l_ldat_ackn` and `l_hdat_ackn` are not relevant.



The PCI MegaCore functions do not ensure that the combination of the `ad[1..0]` and `cben[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 2.2** for more information on handling invalid combinations of these signals.

Figure 7. I/O Read Transaction

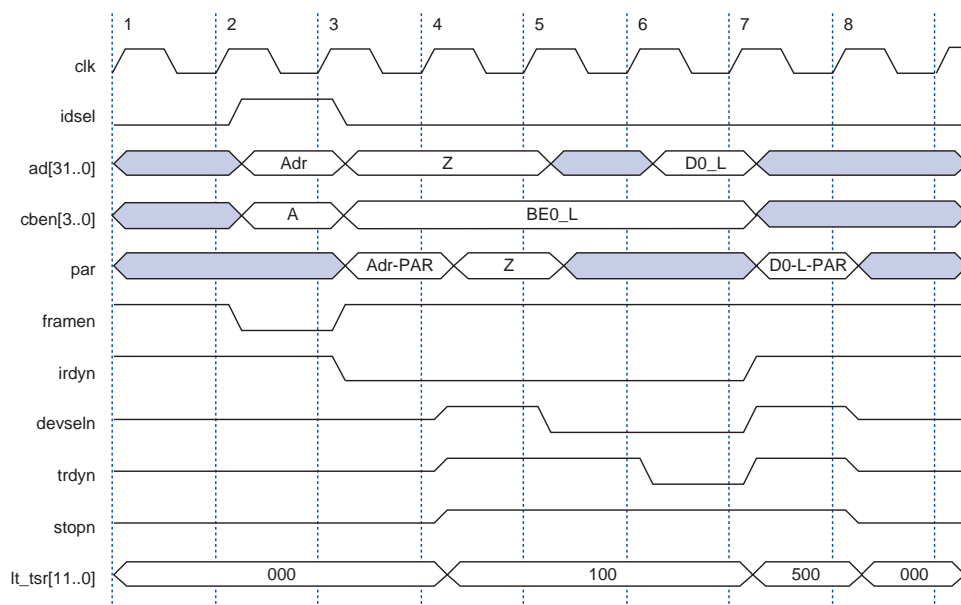


### Configuration Read Transaction

Configuration read transactions are 32 bits. Configuration cycles are automatically handled by the MegaCore functions and do not require local side actions. [Figure 8](#) 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 MegaCore functions assert `trdyn` independent from the `lt_rdyn` signal. This situation results in `trdyn` being asserted in clock 6 instead of clock 7 as shown in [Figure 4](#). The configuration read cycle ends in clock 8.

**Figure 8. Configuration Read Transaction**



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

## 64-Bit Target Write Transactions

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

- Memory single-cycle write
- Memory burst write

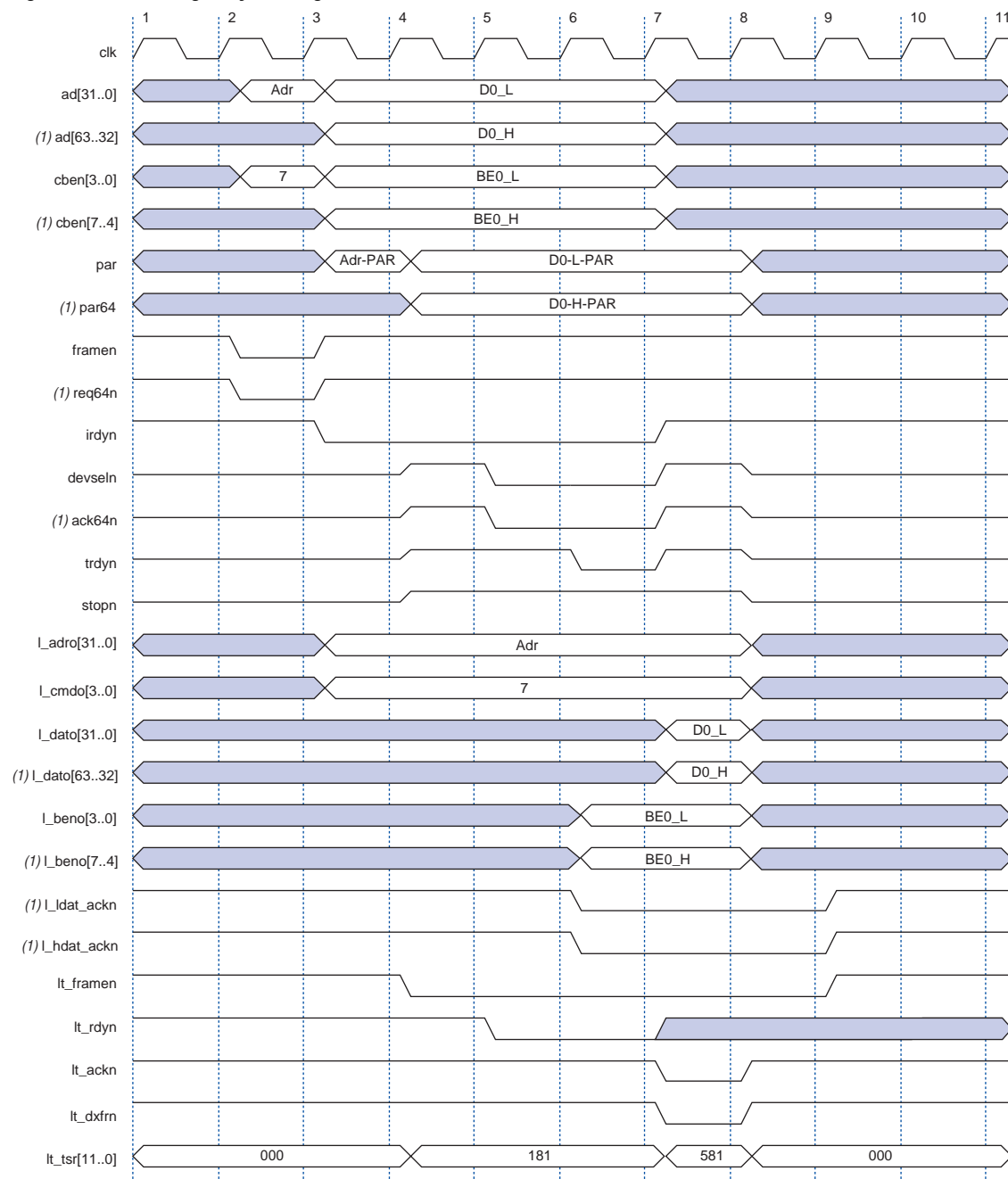
For both types of write transactions, the events follow the sequence described below:

1. The address phase occurs when the PCI master asserts the `framen` and `req64n` signals and drives the address and command on `ad[31..0]` and `cben[3..0]` correspondingly. Asserting `req64n` indicates to the target device that the master device is requesting a 64-bit data transaction.
2. If the address of the transaction matches one of the BARs, the `pci_mt64` or `pci_t64` function turns on the drivers for `ad[63..0]`, `devseln`, `ack64n`, `trdyn`, and `stopn`. The drivers for `par` and `par64` are turned on during the following clock.
3. The `pci_mt64` or `pci_t64` function asserts `devseln` and `ack64n` to indicate to the master device that it is accepting the 64-bit transaction.
4. One or more data phases follow next, depending on the type of write transaction.

### *64-Bit Single-Cycle Target Write Transaction*

Figure 9 shows the waveform for a 64-bit single-cycle target write transaction. This figure applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for the `pci_mt32` and `pci_t32` functions.

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

**Note:**

- (1) These signals do not apply to the `pci_mt32` or `pci_t32` functions for 32-bit target write transactions. For these transactions, the signals should be ignored.

**Table 26** shows the sequence of events for a 64-bit single-cycle target write transaction.

<i>Table 26. 64-Bit Single-Cycle Target Write Transactions (Part 1 of 2)</i>	
Clock Cycle	Event
1	The PCI bus is idle.
2	The address phase occurs.
3	The MegaCore function latches the address and command, and decodes the address to check if it falls within the range of one of its BARs. During clock 3, the master deasserts the <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 target write, this phase is the only data phase in the transaction. The MegaCore function uses clock 3 to decode the address, and if the address falls in the range of one of its BARs, the transaction is claimed.
4	<p>If the MegaCore function detects an address hit in clock 3, several events occur during clock 4:</p> <ul style="list-style-type: none"> <li>■ The MegaCore function informs the local-side device that it is going to claim the write transaction by asserting one of the <code>lt_tsr[5..0]</code> signals and <code>lt_framen</code>. In <a href="#">Figure 9</a>, <code>lt_tsr[0]</code> is asserted indicating that a base address register zero hit.</li> <li>■ The MegaCore function drives the transaction command on <code>l_cmdo[3..0]</code> and address on <code>l_adro[31..0]</code>.</li> <li>■ The 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 5.</li> <li>■ <code>lt_tsr[7]</code> is asserted to indicate that the pending transaction is 64 bits.</li> <li>■ <code>lt_tsr[8]</code> is asserted to indicate that the PCI side of the MegaCore function is busy.</li> </ul>
5	<p>The MegaCore function asserts <code>devseln</code> to claim the transaction. <a href="#">Figure 9</a> also shows the local side asserting <code>lt_rdyn</code>, indicating that it is ready to receive data from the MegaCore function in clock 6.</p> <p>To allow the local side ample time to issue a retry for the write cycle, the MegaCore function does not assert <code>trdyn</code> in the first data phase unless the local side asserts <code>lt_rdyn</code>. If the <code>lt_rdyn</code> signal is not asserted in clock 5 (<a href="#">Figure 9</a>), the MegaCore function delays the assertion of <code>trdyn</code> accordingly.</p>
6	The 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 is the first and last data phase in this cycle.
7	The rising edge of clock 7 registers the valid data from <code>ad[63..0]</code> and drives the data on the <code>l_dato[63..0]</code> bus, registers valid byte enables from <code>cben[7..0]</code> , and drives the byte enables on <code>l_beno[7..0]</code> . At the same time, the MegaCore function asserts the <code>lt_ackn</code> signal to indicate that there is valid data on the <code>l_dato[63..0]</code> bus and a valid byte enable on the <code>l_beno[7..0]</code> bus. Because <code>lt_rdyn</code> is asserted during clock 6, and <code>lt_ackn</code> is asserted in clock 7, data will be transferred in clock 7. <code>lt_dxfrn</code> is asserted in clock 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 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 MegaCore function drives <code>devseln</code> , <code>ack64n</code> , <code>trdyn</code> , and <code>stopn</code> high during this clock cycle.
8	The MegaCore function resets all <code>lt_tsr[11..0]</code> signals because the PCI side has completed the transaction. The MegaCore function also tri-states its control signals.

Table 26. 64-Bit Single-Cycle Target Write Transactions (Part 2 of 2)

Clock Cycle	Event
9	The MegaCore function deasserts <code>lt_framen</code> indicating to the local side that no additional data is in the internal pipeline.

### 64-Bit Target Burst Write Transaction

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

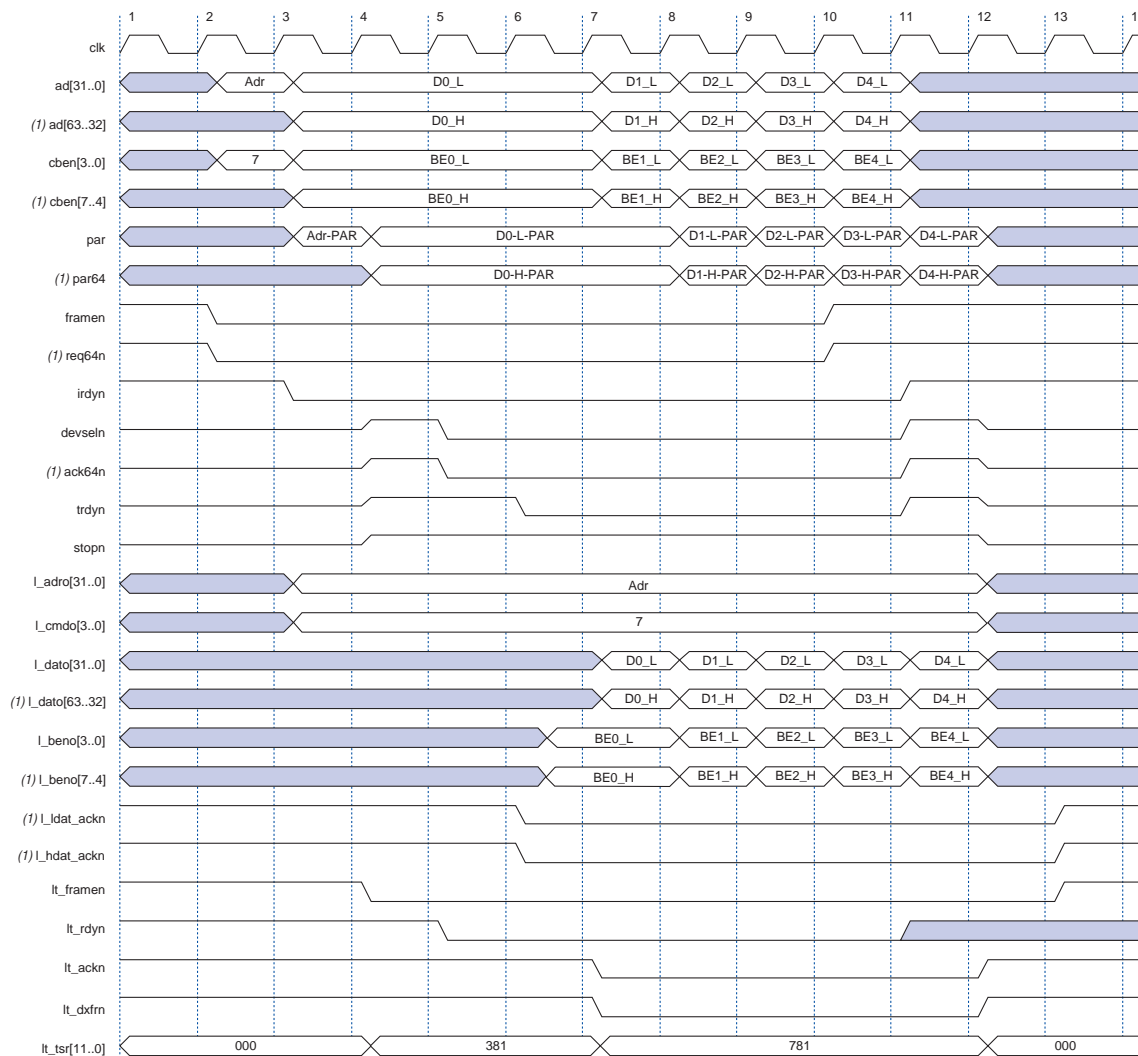
Figure 10 shows a 64-bit zero wait state burst transaction with five data phases. This figure applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for the `pci_mt32` and `pci_t32` functions. The PCI master writes five QWORDS to the MegaCore function during clocks 6 through 10. The local side transfers the same data during clocks 7 through 11 correspondingly. Additionally, Figure 10 shows the `lt_tsr[9]` signal asserted in clock 4 because the master device has the `framen` and `irdyn` signals asserted, thus indicating a burst transaction.



A burst transaction can be identified if both the `irdyn` and `framen` signals are asserted at the same time during a transaction. The MegaCore function asserts `lt_tsr[9]` if both `irdyn` and `framen` are asserted during a valid target transaction. If `lt_tsr[9]` is not asserted during a transaction, it indicates that `irdyn` and `framen` have not been detected or asserted during the transaction. Typically this event indicates that the current transaction is single-cycle. However, this indication is not guaranteed because it is possible for the master to delay the assertion of `irdyn` in the first data phase by up to 8 clocks. In other words, if `lt_tsr[9]` is asserted during a valid target transaction, it indicates that the pending transaction is a burst, but if the `lt_tsr[9]` is not asserted it may or may not indicate that the transaction is single-cycle.



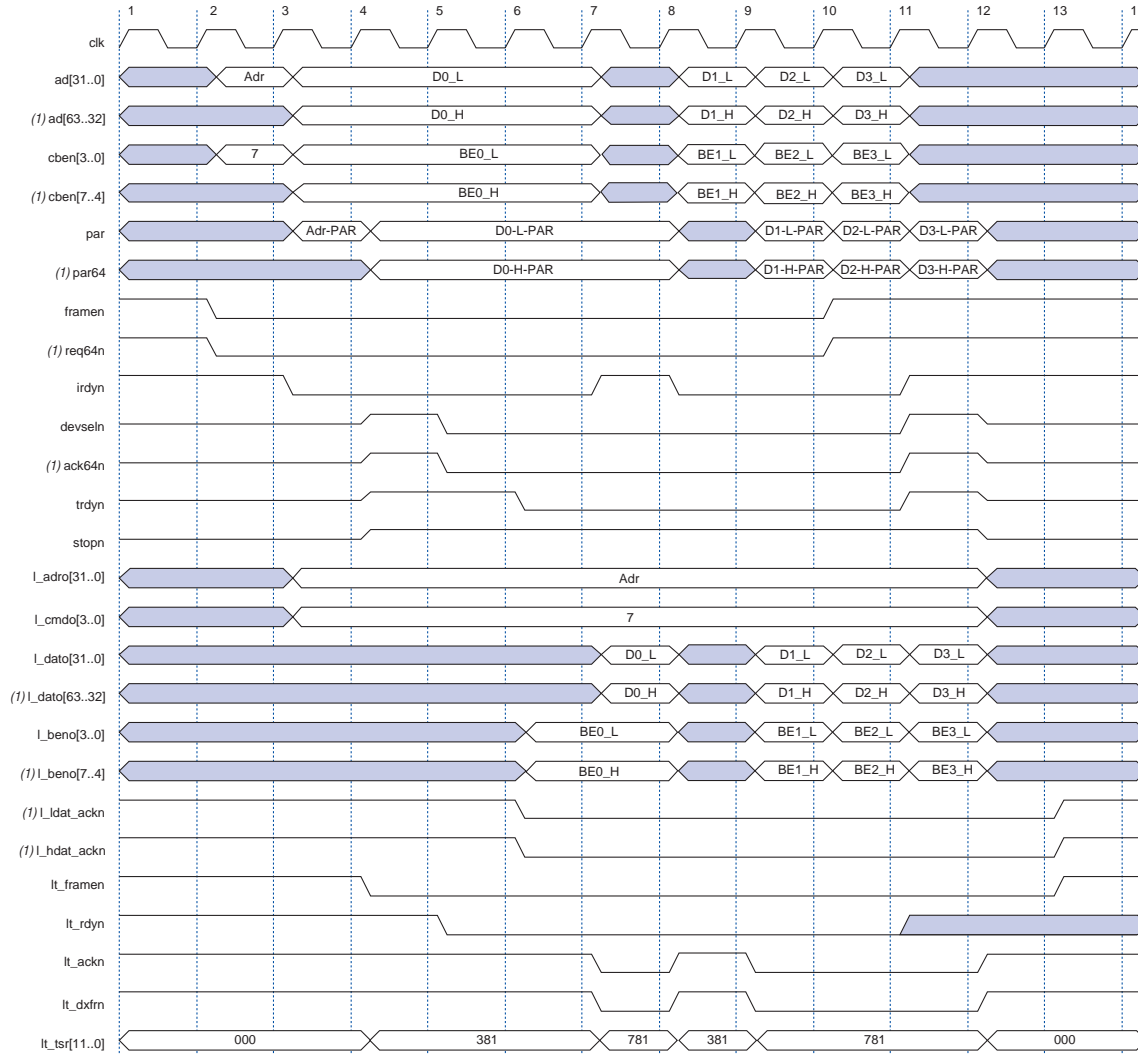
Figure 10. 64-Bit Zero Wait State Target Burst Write Transaction

**Note:**

- (1) These signals do not apply to the `pci_mt32` or `pci_t32` functions for target write transactions. For these transactions, the signals should be ignored.

Figure 11 shows the same transaction as in Figure 10 with the PCI bus master inserting a wait-state. It applies to all PCI functions, except the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`. The PCI bus master inserts a wait state by deasserting the `irdyn` signal in clock 7. The effect of this wait state on the local side is shown in clock 8 with `lt_ackn` deasserted, and as a result `lt_dxfrn` is also deasserted. This transaction prevents data from being transferred to the local side in clock 8 because the internal pipeline of the function does not have valid data.

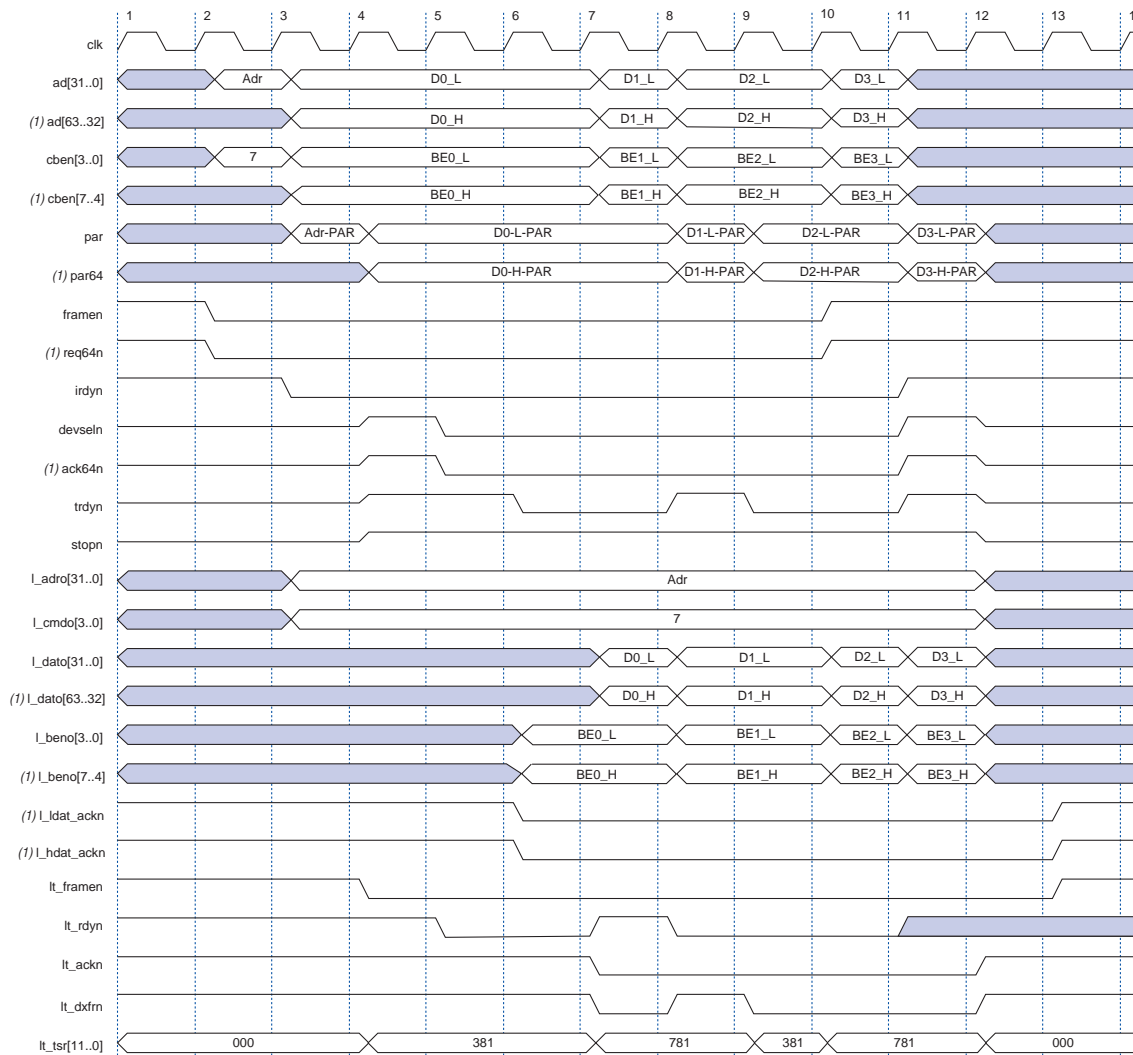
Figure 11. 64-Bit Target Burst Write Transaction with PCI Master Wait State

**Note:**

- (1) These signals do not apply to the `pci_mt32` or `pci_t32` functions for 32-bit target transactions. For these transactions, the signals should be ignored.

Figure 12 shows the same transaction as in Figure 10 with the local side inserting a wait-state. It applies to all PCI functions, except the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`. The local side deasserts `lt_rdyn` in clock 7. The function shows that deasserting `lt_rdyn` in clock 7 suspends the local side data transfer in clock 8 by deasserting `lt_dxfrn`. Because the local side is unable to accept additional data in clock 8, the function deasserts `trdyn` in clock 8 as well, preventing PCI data from being transferred from the master device.

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

**Note:**

- (1) These signals do not apply to the pci\_mt32 or pci\_t32 functions for 32-bit target write transactions. For these transactions, the signals should be ignored.



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

## 32-Bit Target Write Transactions

The PCI MegaCore functions respond to three types of 32-bit target write transactions

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

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

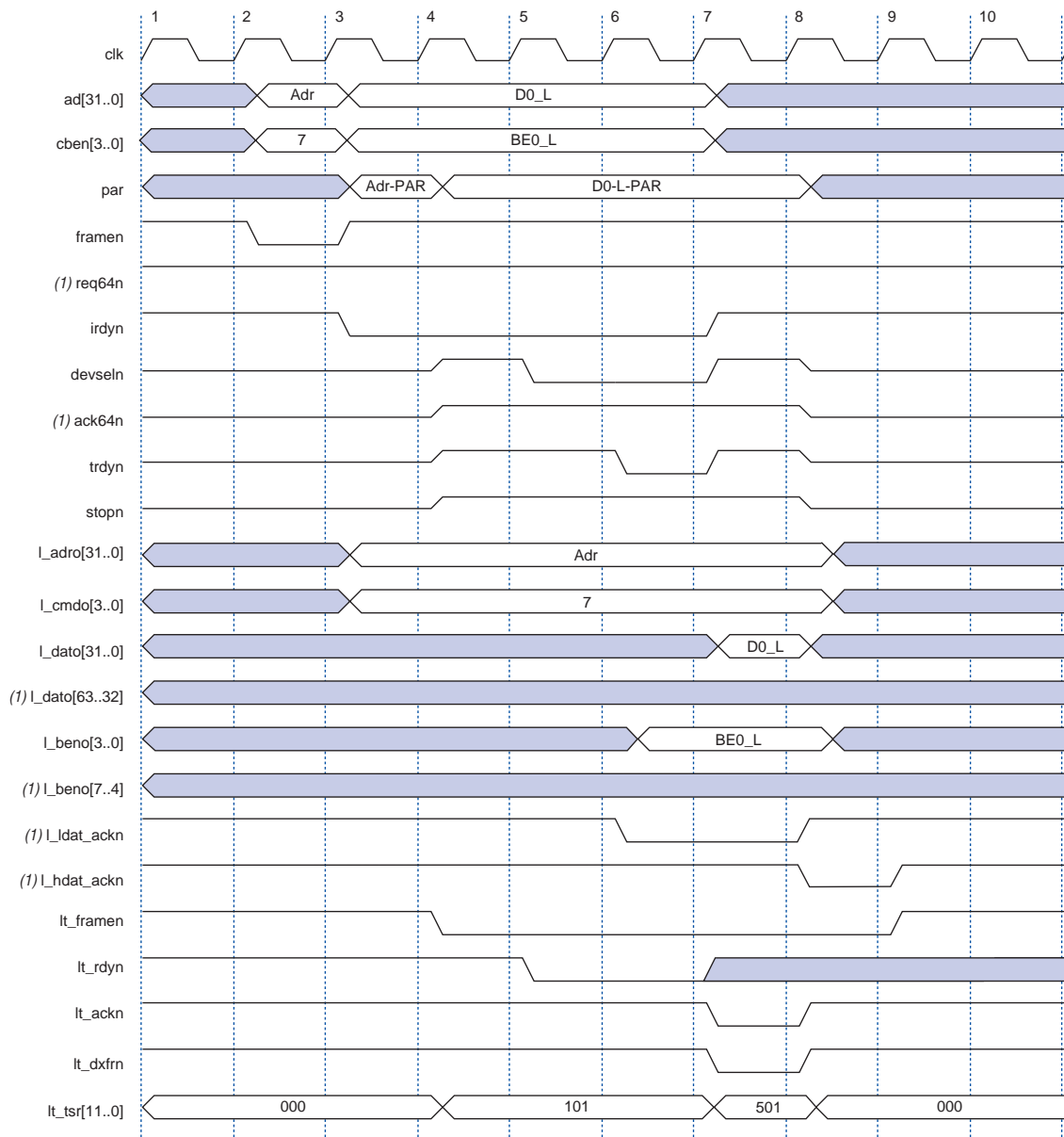
### *32-Bit Memory Write Transaction*

For all MegaCore functions, 32-bit memory write transactions are either single-cycle or burst transactions. For the `pci_mt32` and `pci_t32` functions, the waveforms for 32-bit memory write transactions are described in [Figures 9](#) through [12](#), excluding the 64-bit extension signals as noted. 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 13](#) shows a 32-bit single-cycle memory write transaction. This figure applies to all PCI MegaCore functions, excluding the 64-bit extension signals as noted for the `pci_mt32` and `pci_t32` functions. The sequence of events in [Figure 13](#) is exactly the same as in [Figure 9](#), except for the following:

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

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

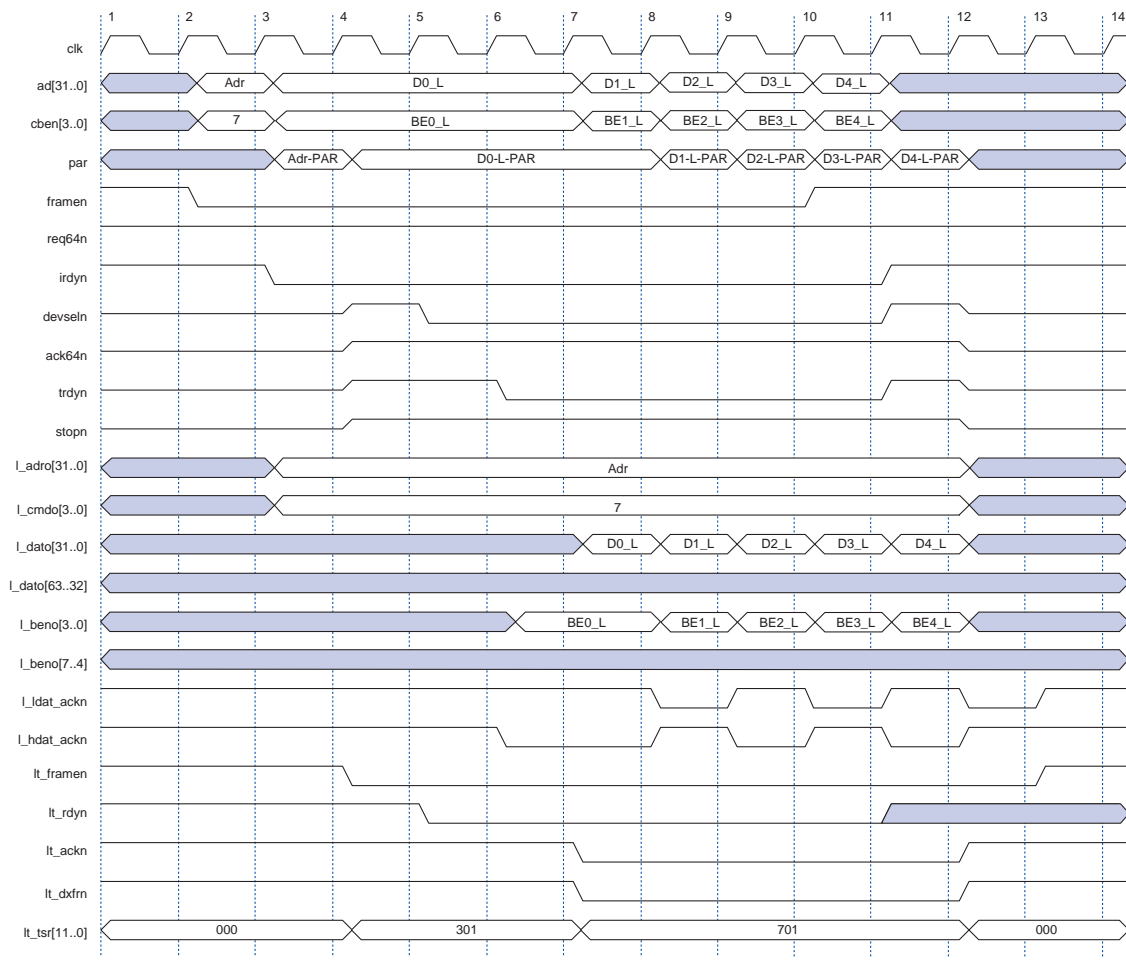
**Note:**

- (1) These signals do not apply to the `pci_mt32` or `pci_t32` functions for 32-bit target write transactions. For these transactions, the signals should be ignored.

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

[Figure 14](#) shows a 32-bit burst memory write transaction; the events are the same for [Figure 10](#). [Figure 14](#) only applies to the `pci_mt64` and `pci_t64` functions. For the `pci_mt32` and `pci_t32` functions, [Figure 10](#) reflects the waveforms for a 32-bit burst memory write transaction, excluding the 64-bit extension signals as noted. The main difference between the two figures is that `l_ldat_ackn` and `l_hdat_ackn` toggle to indicate which DWORD is valid on the local side. In [Figure 14](#), the high DWORD is transferred first because the address used is not a QWORD boundary. This situation occurs because `l_hdat_ackn` is asserted during clock 6 and continues to be asserted until the first DWORD is transferred on the local side during clock 7. The local side is informed that the pending transaction is a 32-bit burst because `lt_tsr[7]` is not asserted and `lt_tsr[9]` is asserted. If the local side cannot handle 32-bit burst transactions, it can disconnect after the first local transfer.

Figure 14. 32-Bit Burst Memory Write Transaction



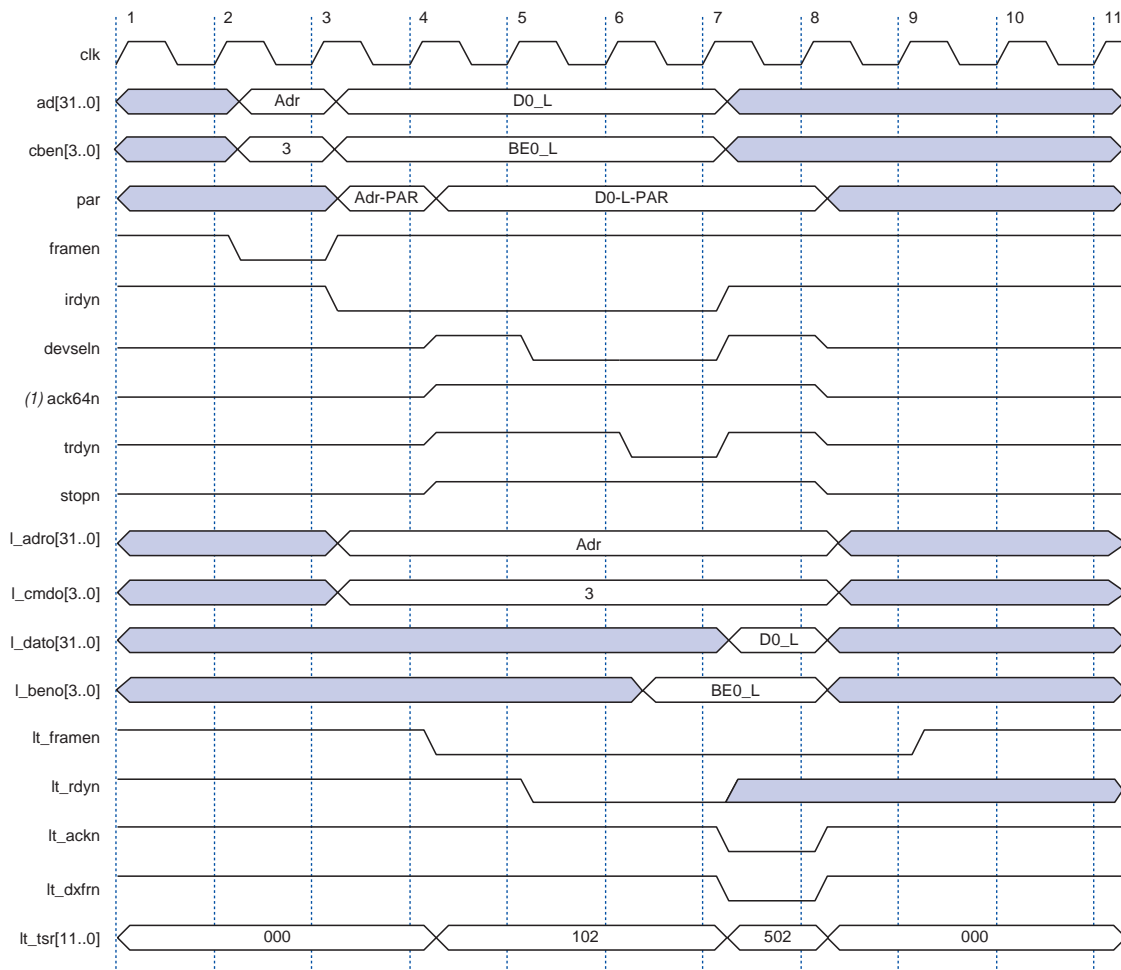
### I/O Write Transaction

I/O write transactions by definition are 32 bits. [Figure 15](#) shows a sample I/O write transaction. This figure applies for PCI MegaCore functions. 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 `cben[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 2.2** for more information on handling invalid combinations of these signals.

Figure 15. I/O Write Transaction



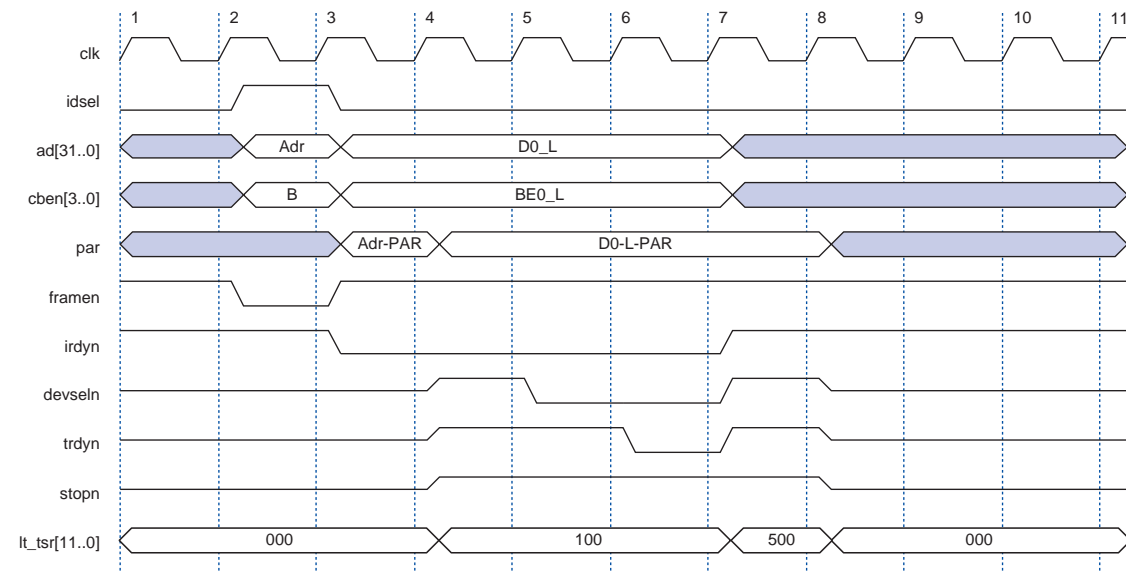


### Configuration Write Transaction

Configuration write transactions are 32 bits. Configuration cycles are automatically handled by the MegaCore functions and do not require local side actions. [Figure 16](#) shows a typical configuration write transaction. This figure applies for PCI MegaCore functions. 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 MegaCore function asserts `trdyn` independent from the `lt_rdyn` signal.

**Figure 16. 32-Bit Configuration Write Transaction**



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

## Target Transaction Terminations

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

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

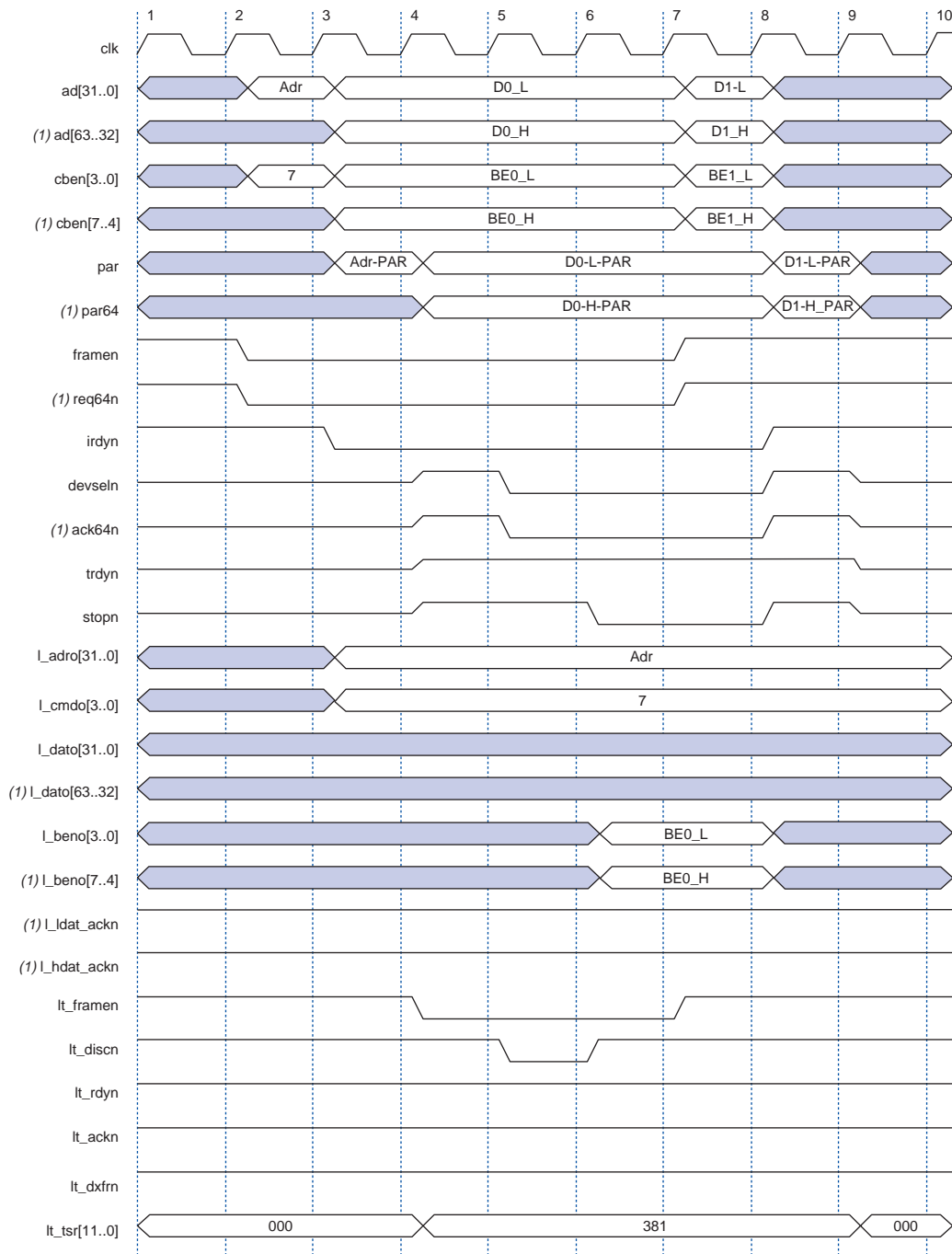


For more details on these termination types, refer to the **PCI Local Bus Specification, Revision 2.2**.

### *Retry*

The local-side device can request a retry, for example, because the device cannot meet the initial latency requirement or because the local resource cannot transfer data at this time. 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 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 MegaCore function. [Figure 17](#) applies to all PCI functions, excluding the 64-bit signals as noted for `pci_mt32` and `pci_t32`.

Figure 17. Target Retry

**Note:**

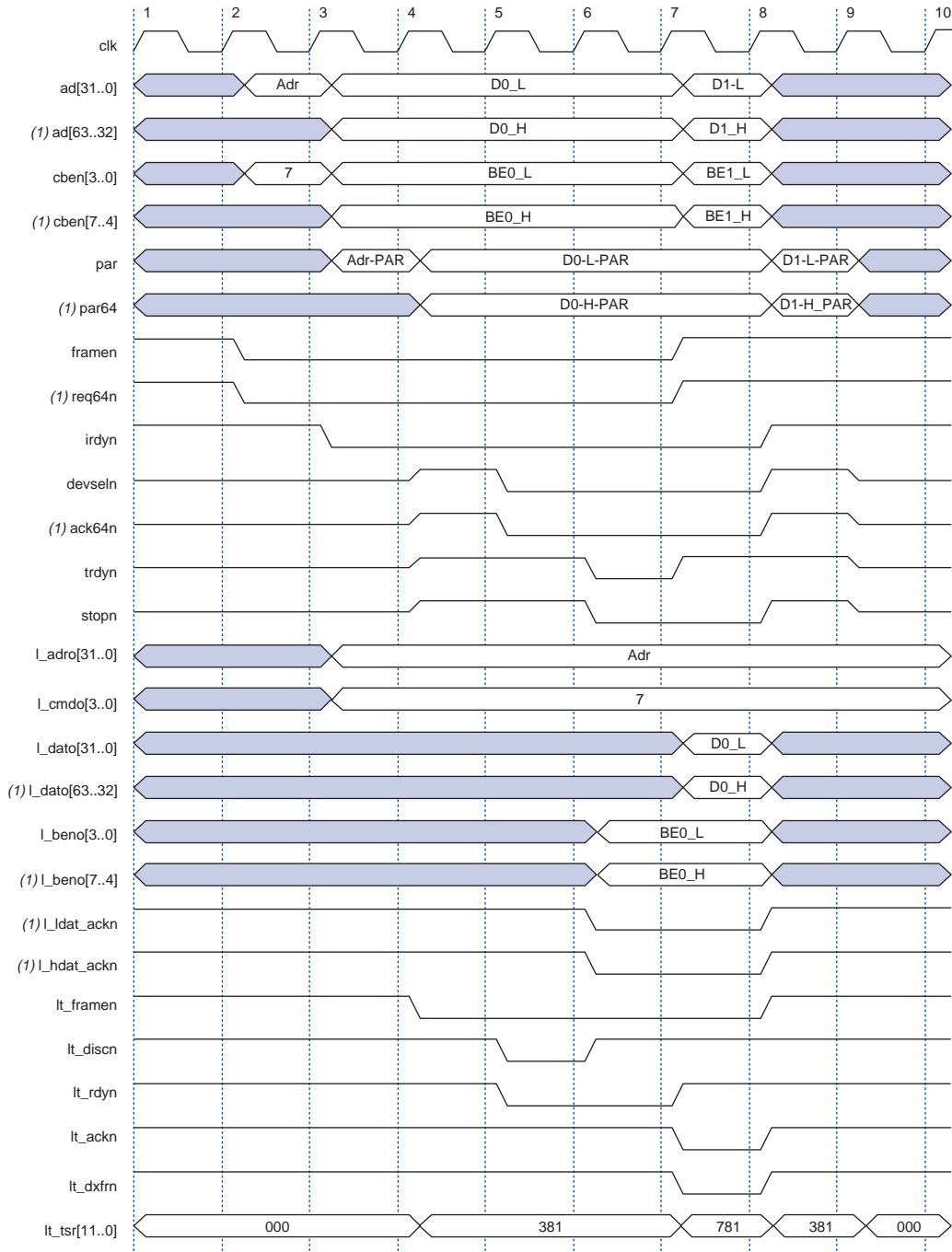
(1) These signals do not apply to the `pci_mt32` and `pci_t32` functions and should be ignored.

### *Disconnect*

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

Figure 18 shows an example of a disconnect during a burst write transaction that ensures only a single data phase is completed. Figure 18 applies to all PCI functions, excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`. In Figure 18, both `lt_rdyn` and `lt_discn` are asserted in clock 5. This transaction informs the MegaCore function that the local side is ready to accept data but also wants to disconnect. As a result, the MegaCore function issues a disconnect with data and accepts only one data phase.

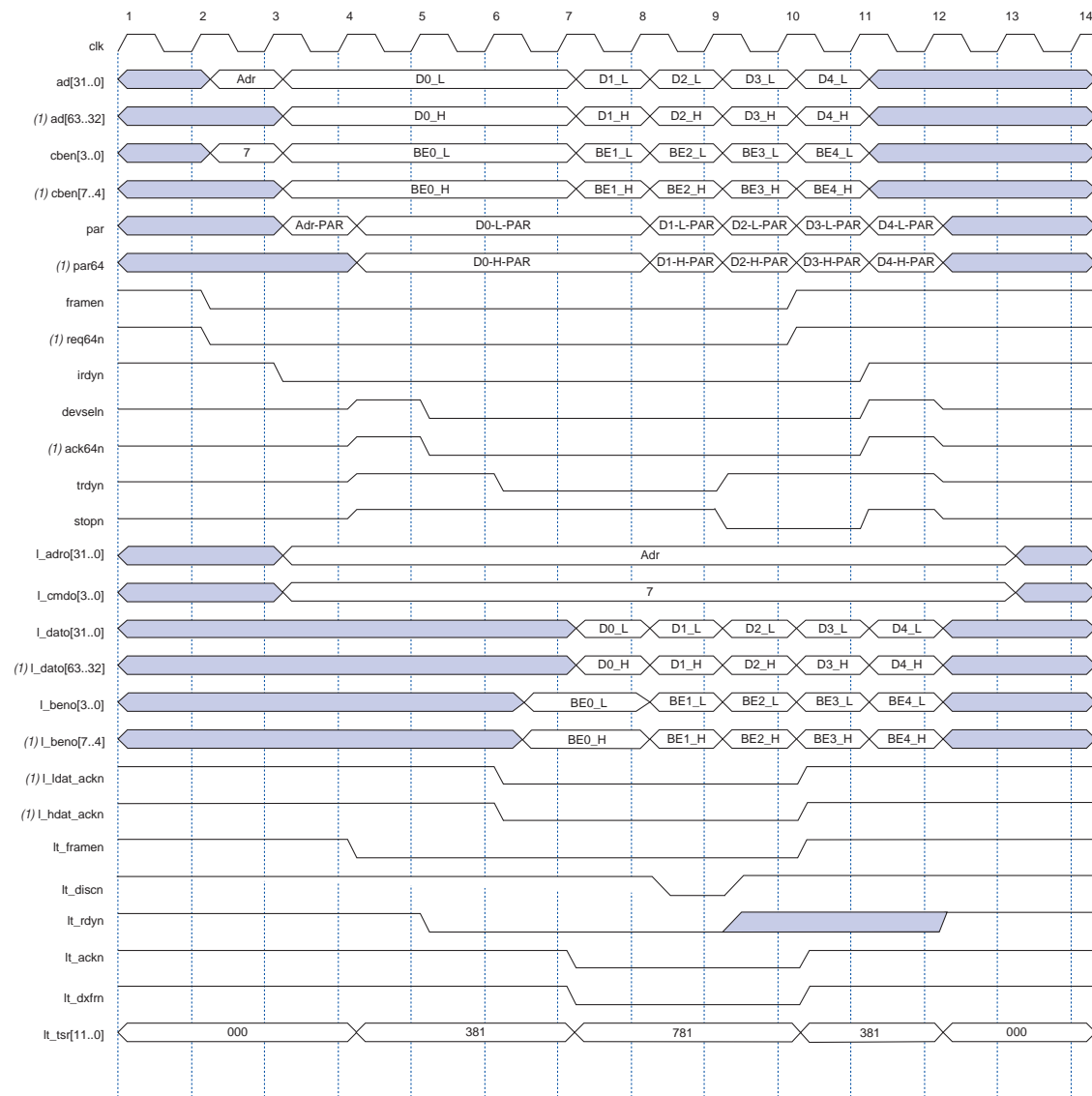
Figure 18. Single Data Phase Disconnect in a Burst Write Transaction

**Note:**

(1) These signals do not apply to the `pci_mt32` and `pci_t32` functions and should be ignored.

Figure 19 shows an example of a disconnect during a burst target write transaction where multiple data phases are completed. Figure 19 applies to all PCI 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`—when it is asserted after `lt_rdyn` during a burst target write transaction.

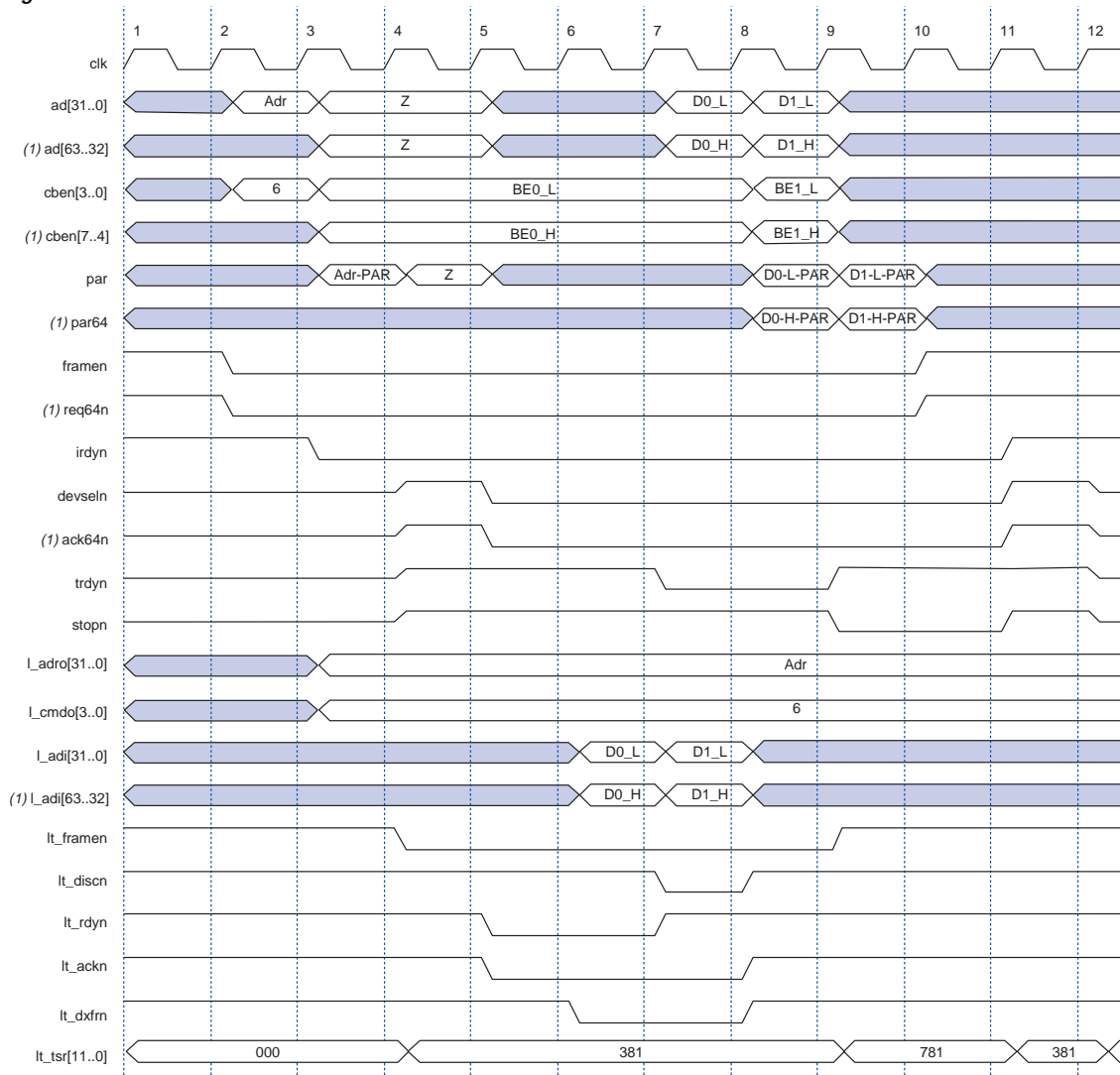
Figure 19. Disconnect in a Burst Write Transaction

**Note:**

(1) These signals do not apply to the `pci_mt32` and `pci_t32` functions and should be ignored.

Figure 20 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 7 indicating that valid data will be available on the local side in clock 8. Then, `lt_discn` is asserted in clock 8 indicating the last data phase to be completed on the local side.

Figure 20. Disconnect in a Burst Read Transaction



**Note:**

(1) These signals do not apply to the `pci_mt32` and `pci_t32` functions and should be ignored.





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

### *Target Abort*

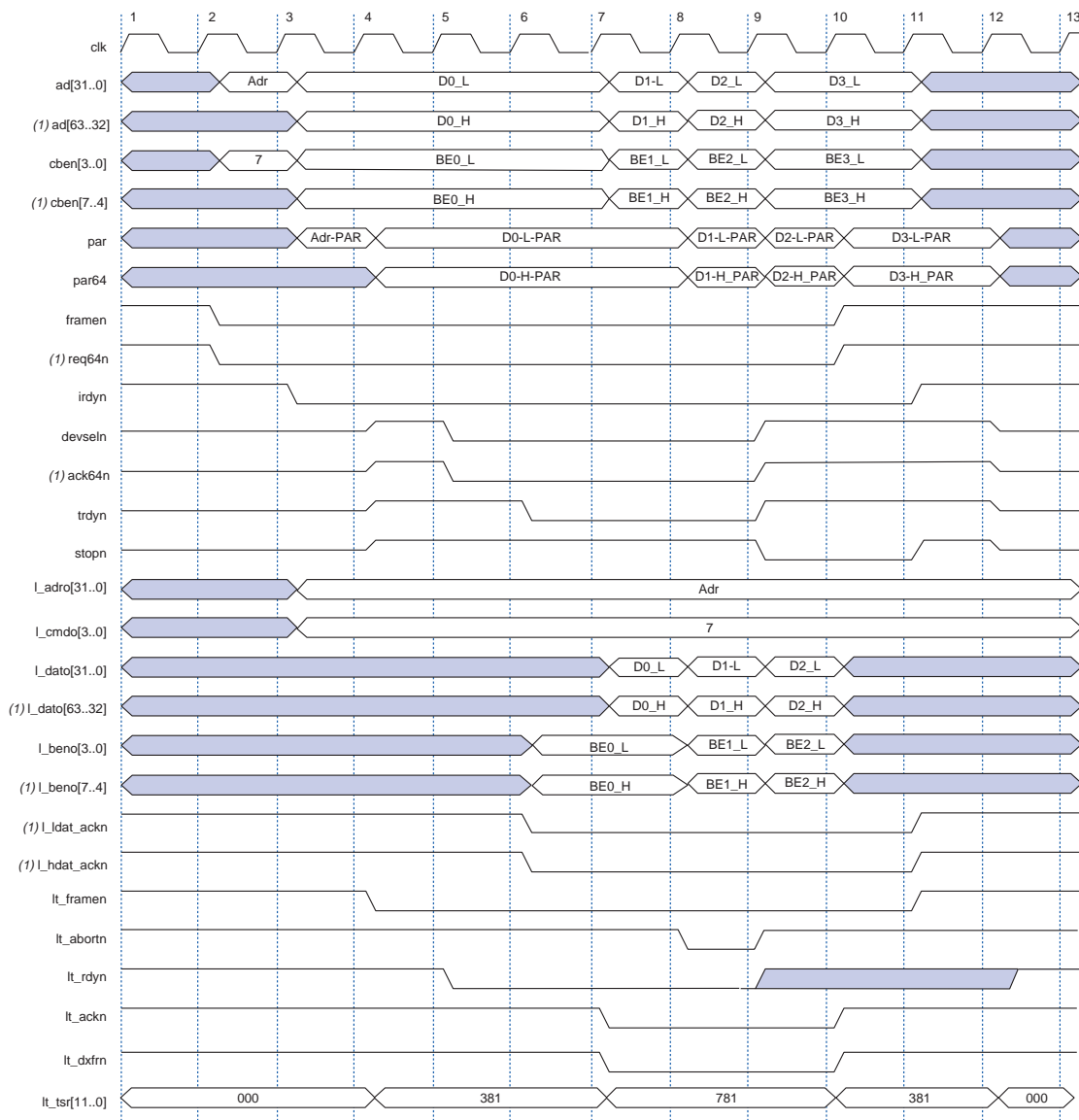
Target abort refers to an abnormal termination because either the local logic detected a fatal error, or the target will never be able to complete the request. An abnormal termination may cause a fatal error for the application that originally requested the transaction. A target abort allows the transaction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

A target device issues an abort by deasserting `devseln` and `trdyn` and asserting `stopn`. A target device must set the `tabort_sig` bit in the PCI status register whenever it issues a target abort. See “[Status Register](#)” on [page 54](#) for more details. [Figure 21](#) shows the MegaCore function issuing an abort during a burst write cycle. It applies to all PCI functions, excluding the 64-bit extension signals as noted for `pci_mt32` and `pci_t32`.



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

Figure 21. Target Abort

**Note:**

(1) These signals do not apply to the `pci_mt32` and `pci_t32` functions and should be ignored.

## 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 27](#) lists the PCI and local side signals that apply for each PCI function.

<i>Table 27. PCI MegaCore Function Signals (Part 1 of 2)</i>		
PCI Signals	<code>pci_mt64</code>	<code>pci_mt32</code>
<code>clk</code>	✓	✓
<code>rstn</code>	✓	✓
<code>gntn</code>	✓	✓
<code>reqn</code>	✓	✓
<code>ad[63..0]</code>	✓	<code>ad[31..0]</code>
<code>cben[7..0]</code>	✓	<code>cben[3..0]</code>
<code>par</code>	✓	✓
<code>par64</code>	✓	
<code>idsel</code>	✓	✓
<code>framen</code>	✓	✓
<code>req64n</code>	✓	
<code>irdyn</code>	✓	✓
<code>devseln</code>	✓	✓
<code>ack64n</code>	✓	
<code>trdyn</code>	✓	✓
<code>stopn</code>	✓	✓
<code>perrn</code>	✓	✓
<code>serrn</code>	✓	✓
<code>intan</code>	✓	✓
Local side signals		
<code>l_adi[63..0]</code>	✓	<code>l_adi[31..0]</code>
<code>l_cbeni[7..0]</code>	✓	<code>l_cbeni[3..0]</code>
<code>l_adro[63..0]</code>	✓	<code>l_adro[31..0]</code>
<code>l_dato[63..0]</code>	✓	<code>l_dato[31..0]</code>
<code>l_beno[7..0]</code>	✓	<code>l_beno[3..0]</code>
<code>l_cmndo[3..0]</code>	✓	✓
<code>l_ldat_ackn</code>	✓	
<code>l_hdat_ackn</code>	✓	
Target local side		
<code>lt_abortn</code>	✓	✓
<code>lt_discn</code>	✓	✓

Table 27. PCI MegaCore Function Signals (Part 2 of 2)

PCI Signals	pci_mt64	pci_mt32
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		
lm_req32n	✓	✓
lm_req64n	✓	
lm_lastn	✓	✓
lm_rdyn	✓	✓
lm_adr_ackn	✓	✓
lm_ackn	✓	✓
lm_dxfrn	✓	✓
lm_tsr[9..0]	✓	✓

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

- 64-bit memory burst master read
- 64-bit memory single-cycle master read
- 64-bit memory burst master write
- 64-bit memory single-cycle master write (only supported if the **64-Bit Only Devices** option is turned on in the wizard)

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

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

A master operation begins when the local-side master interface asserts the `lm_req64n` signal to request a 64-bit transaction or the `lm_req32n` signal to request a 32-bit transaction. The PCI 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 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 any transaction in master mode because the local side provides the function with the exact command. 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 device may require a long time to transfer data to/from the function during a burst transaction. The local-side device 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 2.2**, if the arbiter parks on the `pci_mt64` or `pci_mt32`, the function drives the `ad[31..0]`, `cben[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.

## 64-Bit Master Read Transactions

In master mode, the `pci_mt64` function supports two types of 64-bit read transactions:

- Burst memory read
- Single-cycle read

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



The events for 32-bit master read transactions with `pci_mt32` are the same as 64-bit master read transactions in `pci_mt64` except that `lm_req32n` is used instead of `lm_req64n` and the 64-bit extension signals are not implemented in `pci_mt32`.

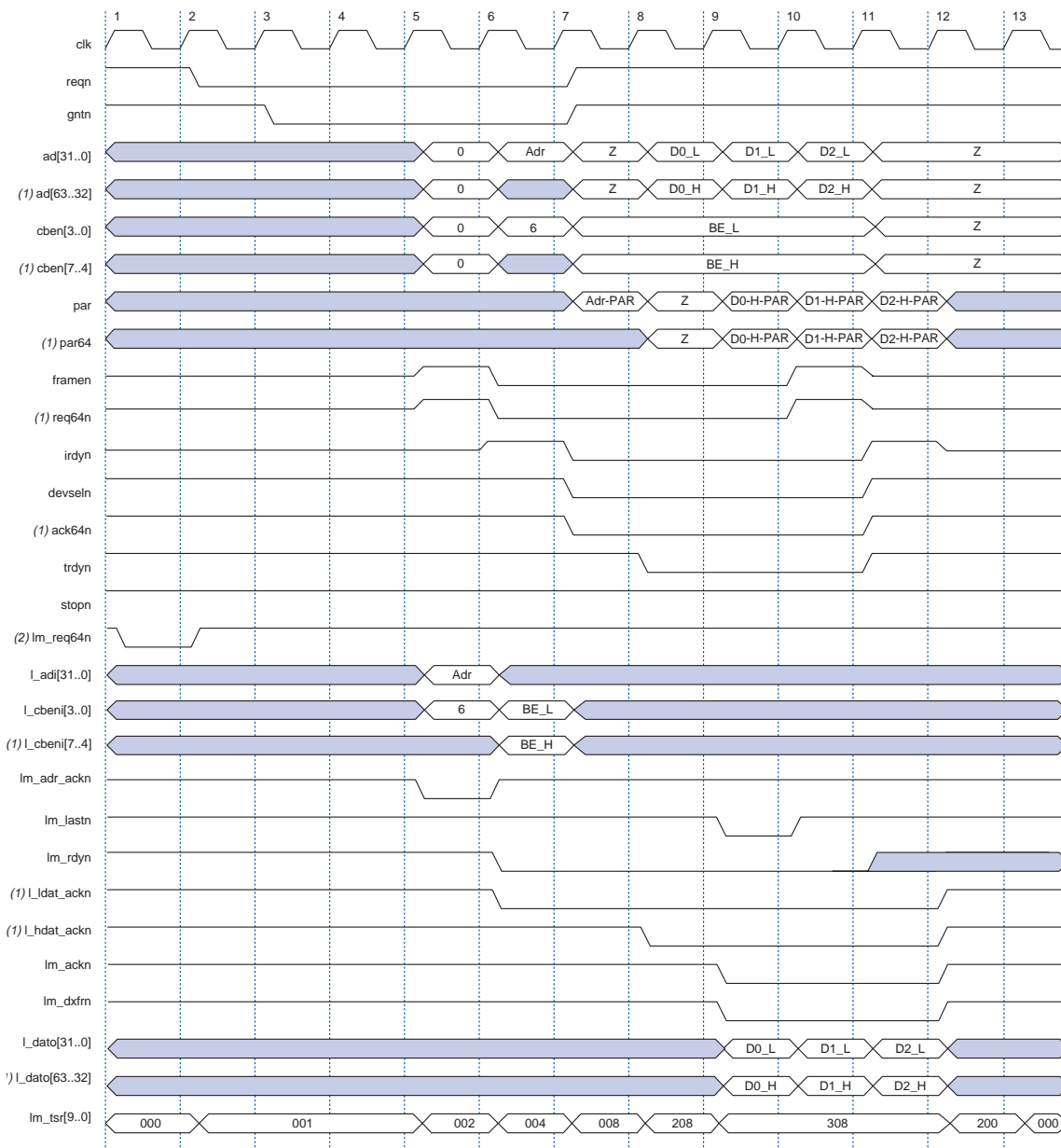
1. The local side asserts `lm_req64n` to request a 64-bit transaction. Consequently, the `pci_mt64` function asserts `reqn` to request bus ownership from the PCI arbiter.

2. When the PCI arbiter grants bus ownership by asserting the `gntn` signal, the `pci_mt64` function asserts `lm_adr_ackn` on the local side to acknowledge the transaction address and command. During the same clock cycle when `lm_adr_ackn` 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_mt64` function turns on the drivers for `framen` and `req64n`.
3. The `pci_mt64` function begins the PCI address phase. During the PCI address phase, the local side must provide the byte enables for the transaction on `l_cbeni[7..0]`; for burst transactions, the local side must ensure that `l_cbeni[7..0] = B"00000000"`. The byte enables provided will be used throughout the master transaction. At the same time, the `pci_mt64` function turns on the driver for `irdyn`.
4. A turn-around cycle on the `ad[63..0]` occurs during the clock immediately following the address phase. During the turn-around cycle, the `pci_mt64` function tri-states `ad[63..0]`, but drives the correct byte enables on `cben[7..0]` for the first data phase. This process is necessary because the `pci_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 next, 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.

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

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

**Notes:**

- (1) This signal does not apply to pci\_mt32 for 32-bit transactions. For these transactions, the signal should be ignored.
- (2) For pci\_mt32, lm\_req64n should be exchanged with lm\_req32n for 32-bit master transactions.



**Table 28** shows the sequence of events for a 64-bit zero-wait-state master burst memory read transaction.

**Table 28. 64-Bit Master Zero-Wait-State Burst Memory Read 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 <a href="#">Figure 22</a> 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	<p>The function turns on its output drivers, getting ready to begin the address phase.</p> <p>The function also asserts <code>lm_adr_ackn</code> to indicate to the local side that it has acknowledged its request. 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>.</p> <p>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.</p>
6	<p>The function begins the 64-bit memory read transaction with the address phase by asserting <code>framen</code> and <code>req64n</code>.</p> <p>At the same time, the local side must provide the byte enables for the transaction on <code>l_cbeni[7..0]</code>. The byte enables provided will be used throughout the master transaction. The local side also asserts <code>lm_rdyn</code> to indicate that it is ready to accept data.</p> <p>The function asserts <code>lm_tsr[2]</code> to indicate to the local side that the PCI bus is in its address phase.</p>
7	<p>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.</p> <p>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.</p> <p>During this clock cycle, the function also asserts <code>lm_tsr[3]</code> to inform the local side that it is in data transfer mode.</p>

**Table 28. 64-Bit Master Zero-Wait-State Burst Memory Read Transaction (Part 2 of 3)**

Clock Cycle	Event
8	<p>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 9.</p> <p>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.</p>
9	<p>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_dato[63..0]</code> data lines.</p> <p>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 10.</p> <p>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 10). The last data phase on the PCI side takes place during clock 10.</p> <p>The function also asserts <code>lm_tsr[8]</code> in the same clock to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock.</p>
10	<p>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 11.</p> <p>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_dato[63..0]</code> data lines. The local side has now received two valid 64-bit data.</p> <p>The function continues to assert <code>lm_tsr[8]</code> informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.</p>

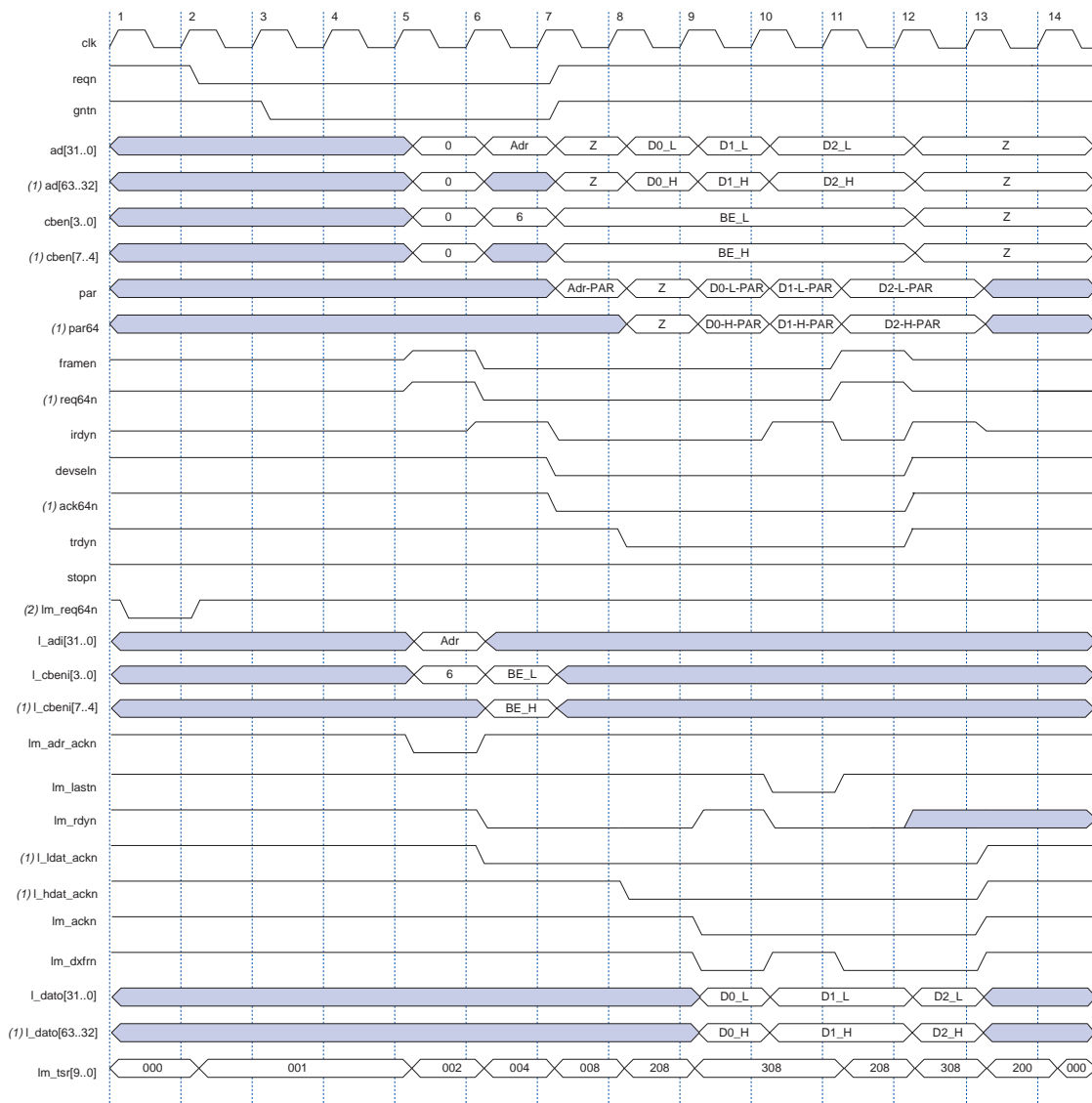
**Table 28. 64-Bit Master Zero-Wait-State Burst Memory Read Transaction (Part 3 of 3)**

Clock Cycle	Event
11	<p>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.</p> <p>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_dato[63..0]</code> bus is valid. The local side has now received three 64-bit words of data.</p> <p>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.</p> <p>The function continues to assert <code>lm_tsr[8]</code> informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.</p>
12	<p>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.</p>

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

Figure 23 shows the same transaction as in Figure 22 with the local side 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 local side deasserts `lm_rdyn` in clock 9. Consequently, on the following clock cycle (clock 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 23. 64-Bit Master Burst Memory Read Transaction with Local Wait State

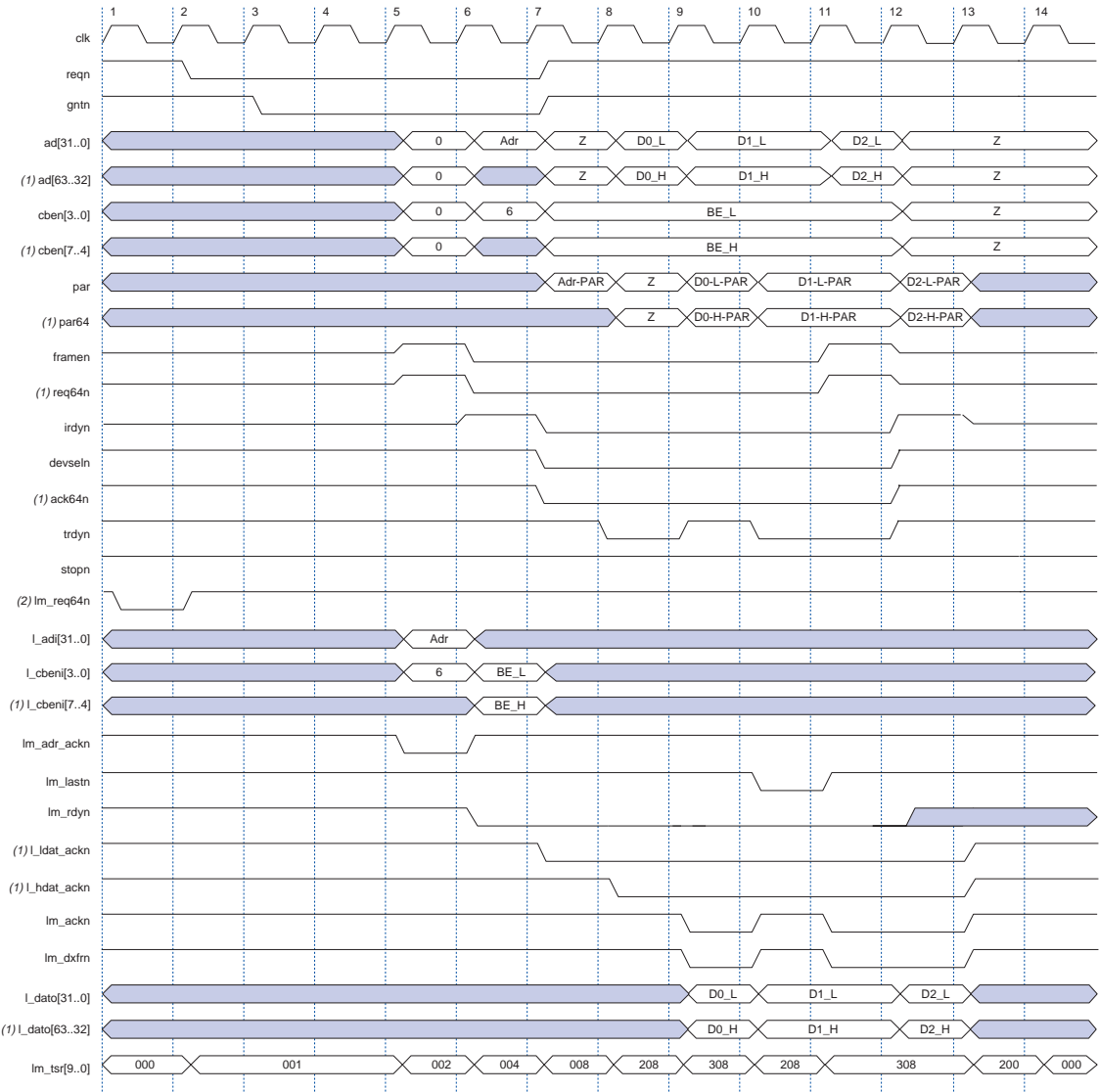
**Notes:**

- (1) This signal does not apply to `pci_mt32` for 32-bit transactions. For these transactions, the signal should be ignored.
- (2) For `pci_mt32`, `lm_req64n` should be exchanged with `lm_req32n` for 32-bit master transactions.

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

Figure 24 shows the same transaction as in Figure 22 with the PCI bus target 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 target inserts a wait state by deasserting `trdyn` in clock 9. Consequently, on the following clock cycle (clock 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 9 and on the local side in clock 10.

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

**Notes:**

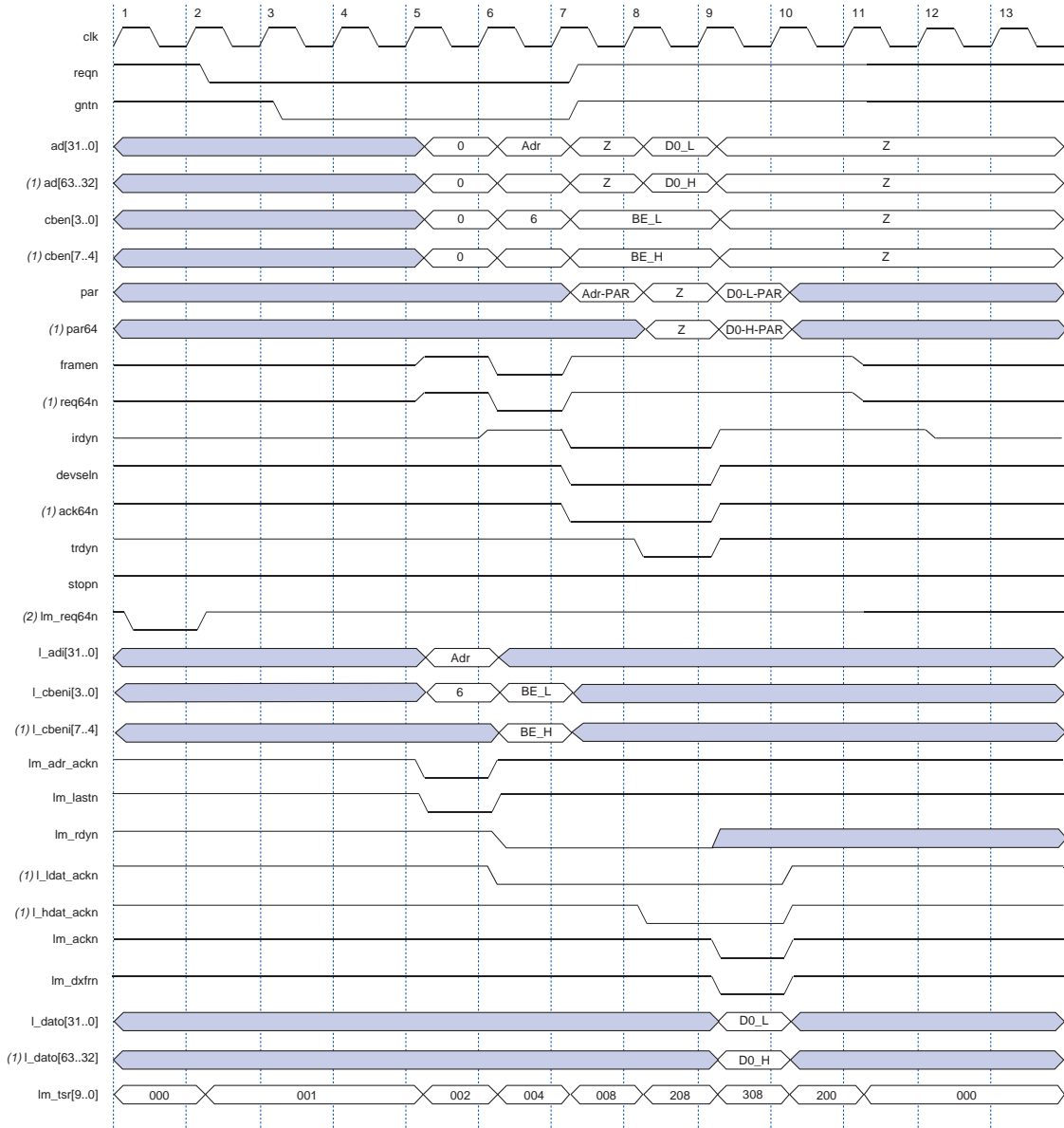
- (1) This signal does not apply to `pci_mt32` for 32-bit transactions. For these transactions, the signal should be ignored.
- (2) For `pci_mt32`, `lm_req64n` should be exchanged with `lm_req32n` for 32-bit master transactions.

### *64-Bit Master Single-Cycle Memory Read Transaction*

The `pci_mt64` function can perform 64-bit master single-cycle memory read transactions. If your application is a system that has only 64-bit PCI devices and the local side wants to transfer one 64-bit data, Altera recommends that you perform a 64-bit single-cycle memory read transaction. However, if your application is a system that has 32- and 64-bit PCI devices and the local side wants to transfer one 64-bit data, Altera recommends that a 32-bit burst memory read transaction is performed.

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

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

**Notes:**

- (1) This signal does not apply to pci\_mt32 for 32-bit transactions. For these transactions, the signal should be ignored.
- (2) For pci\_mt32, lm\_req64n should be exchanged with lm\_req32n for 32-bit master transactions.



## 32-Bit Master Read Transactions

In master mode, the `pci_mt64` and `pci_mt32` functions support three types of 32-bit read transactions:

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

For both the `pci_mt64` and `pci_mt32` functions, 32-bit memory read transactions are either single-cycle or burst. The 32-bit master read transactions are similar to 64-bit master read transactions, but the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid. For `pci_mt32`, the waveforms for 32-bit memory read transactions are described in [Figures 22](#) through 25, excluding the 64-bit extension signals as noted, and in [Figures 27](#) and 28.

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

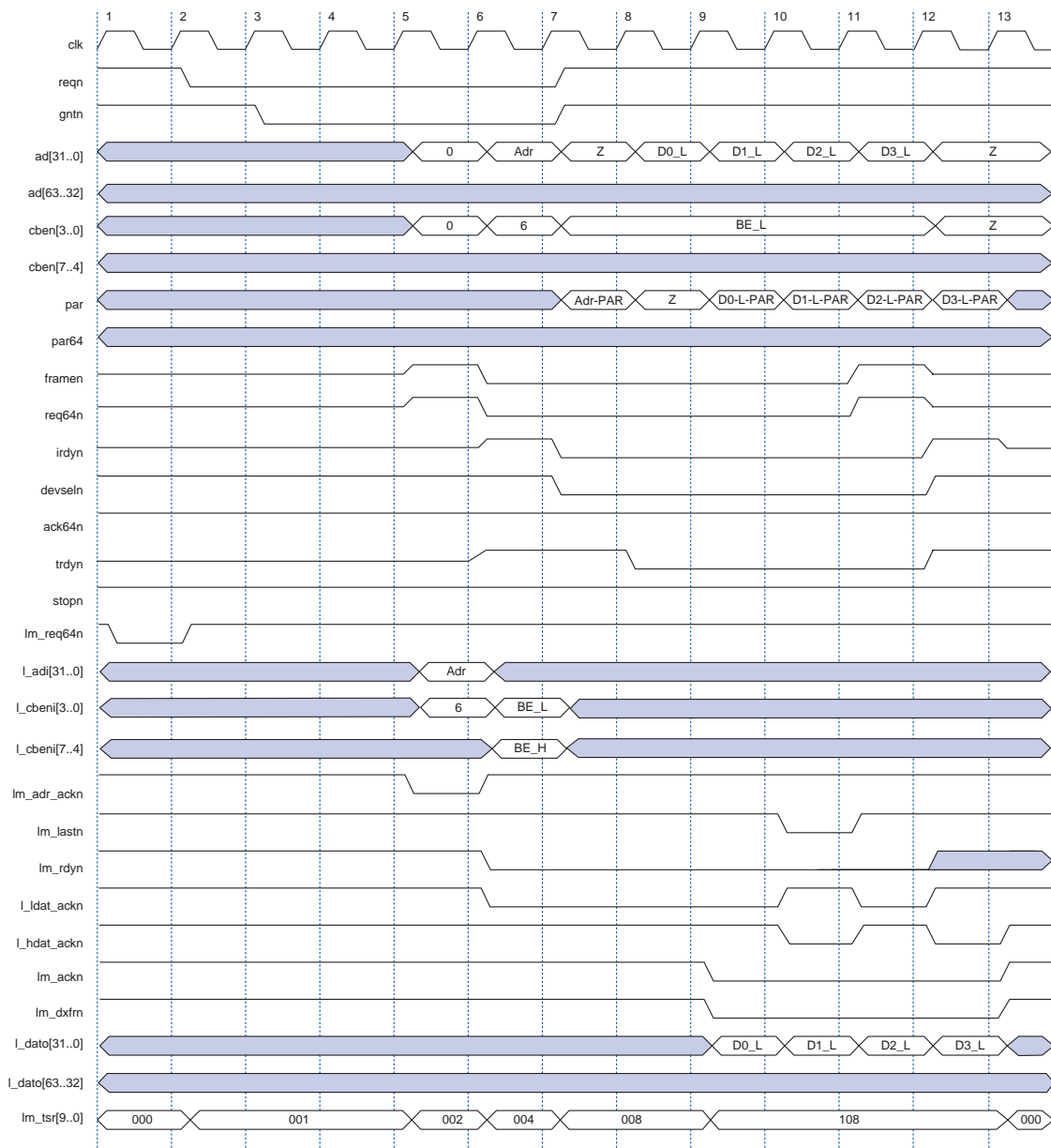
[Figure 26](#) shows the same transaction as in [Figure 22](#), but because the PCI target cannot transfer 64-bit transactions, this figure applies to the `pci_mt64` function only. In this transaction, the local-side master interface requests a 64-bit transaction by asserting `lm_req64n`. The `pci_mt64` function asserts `req64n` on the PCI side. However, the PCI target cannot transfer 64-bit data, and therefore does not assert `ack64n` in clock 7. Accordingly, the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid.

Valid data is only presented on the `l_dato[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 address is on a QWORD boundary (i.e., `ad[2..0]=B"000"`) or not. 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.

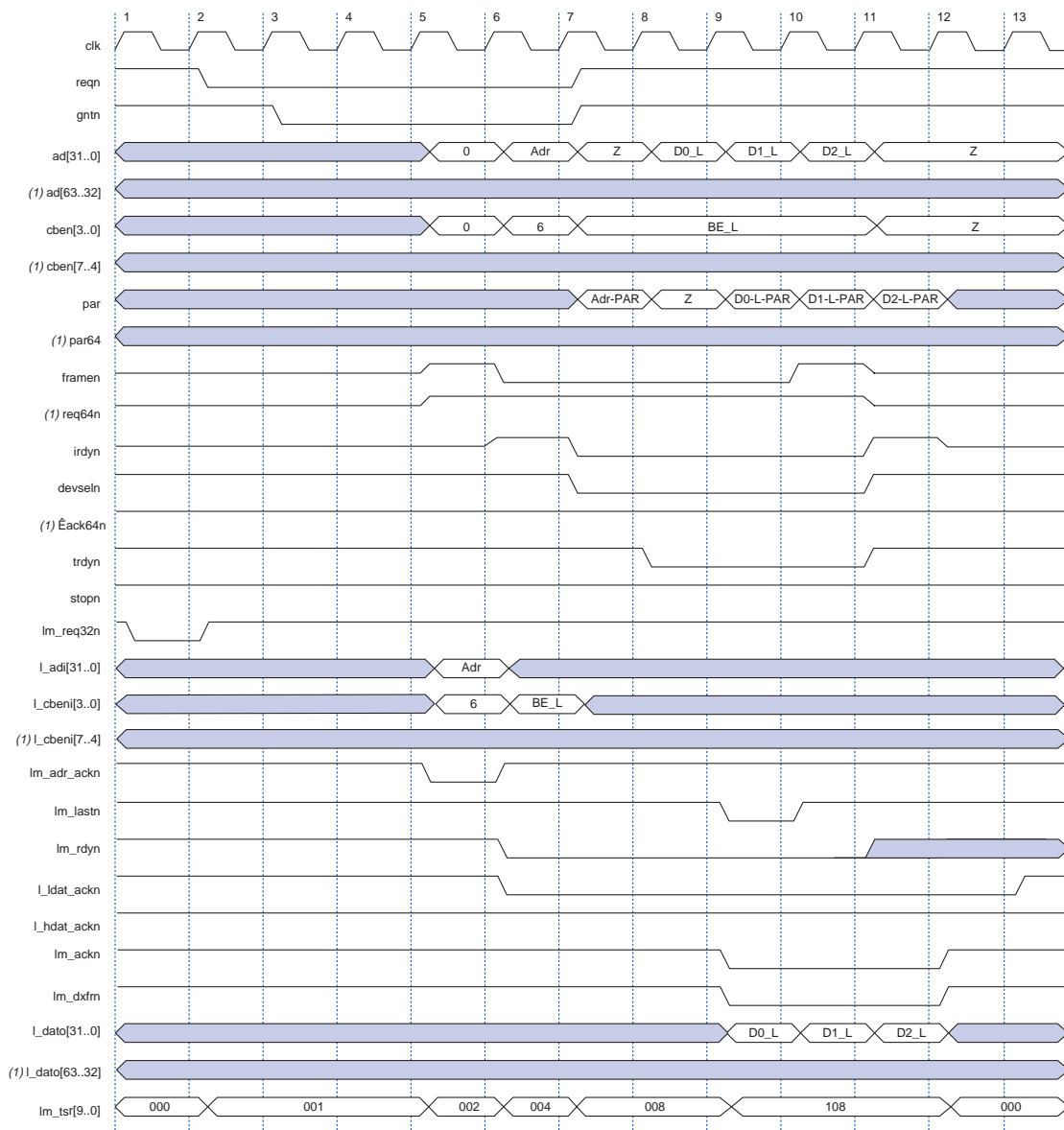
Figure 26. 32-Bit PCI &amp; 64-Bit Local-Side Master Burst Memory Read Transaction



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

Figure 27 shows the same transaction as in Figure 22, but the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. 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_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 27. 32-Bit PCI &amp; 32-Bit Local-Side Master Burst Memory Read Transaction

**Note:**

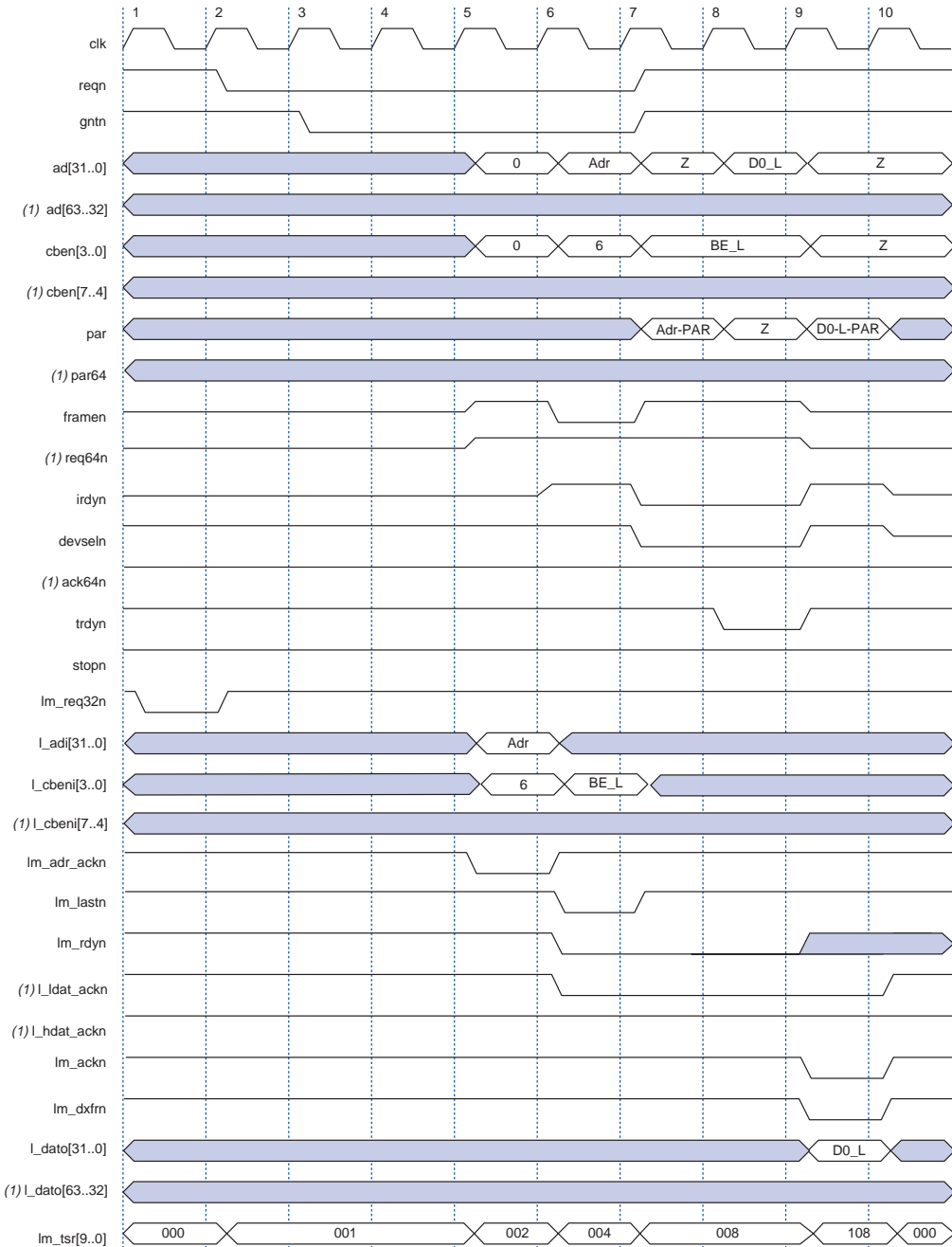
- (1) This signals does not apply to `pci_mt32` for 32-bit master read transactions. For these transactions, the signal should be ignored.

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

Figure 28 shows the same transaction as in Figure 27, but the local side master interface transfers only one data phase. This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`. This waveform also applies to the following types of single-cycle transactions:

- I/O read
- Configuration read

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

**Note:**

- (1) This signal does not apply to `pci_mt32` for 32-bit master read transactions. For these transactions, the signals should be ignored.

## 64-Bit Master Write Transactions

In master mode, the `pci_mt64` function supports 64-bit memory write transactions. If the **64-Bit Only Devices** option is used, the `pci_mt64` function can perform single-cycle write transactions; however, this option is only recommended for use in specific application environments. For all other cases, Altera recommends performing a 32-bit memory burst write if the local side wants to transfer a single 64-bit data.



Refer to “Parameters” on page 37 for more information on the **64-Bit Only Devices** option.

For 64-bit master write transactions, the sequence of events can be divided into the following steps:



The steps are the same for 32-bit transactions using the `pci_mt32` function, except that the `lm_req32n` signal must be used on the local side to request a 32-bit transaction.

1. The local side asserts `lm_req64n` to request a 64-bit transaction. Consequently, the `pci_mt64` function 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 `pci_mt64` function asserts `lm_adr_ackn` on the local side to acknowledge the transaction's address and command. During the same clock cycle when `lm_adr_ackn` is asserted, the local side should provide the address on `l_adi[31..0]` and the command on `l_cbeni[3..0]`. At the same time, the `pci_mt64` function turns on the drivers for `framen` and `req64n` signals.
3. The `pci_mt64` function begins the PCI address phase. During the PCI address phase, the local side must provide the byte enables for the transaction on `l_cbeni[7..0]`; for burst transactions, the local side must ensure that `l_cbeni[7..0]=B"00000000"`. The byte enables provided are used throughout the transaction. At the same time, the `pci_mt64` function turns on the driver for `irdyn`.
4. If the address of the transaction matches one of the base address registers of a PCI target, the PCI target asserts `devseln` to claim the transaction. One or more data phases follow next, depending on the type of write transaction.

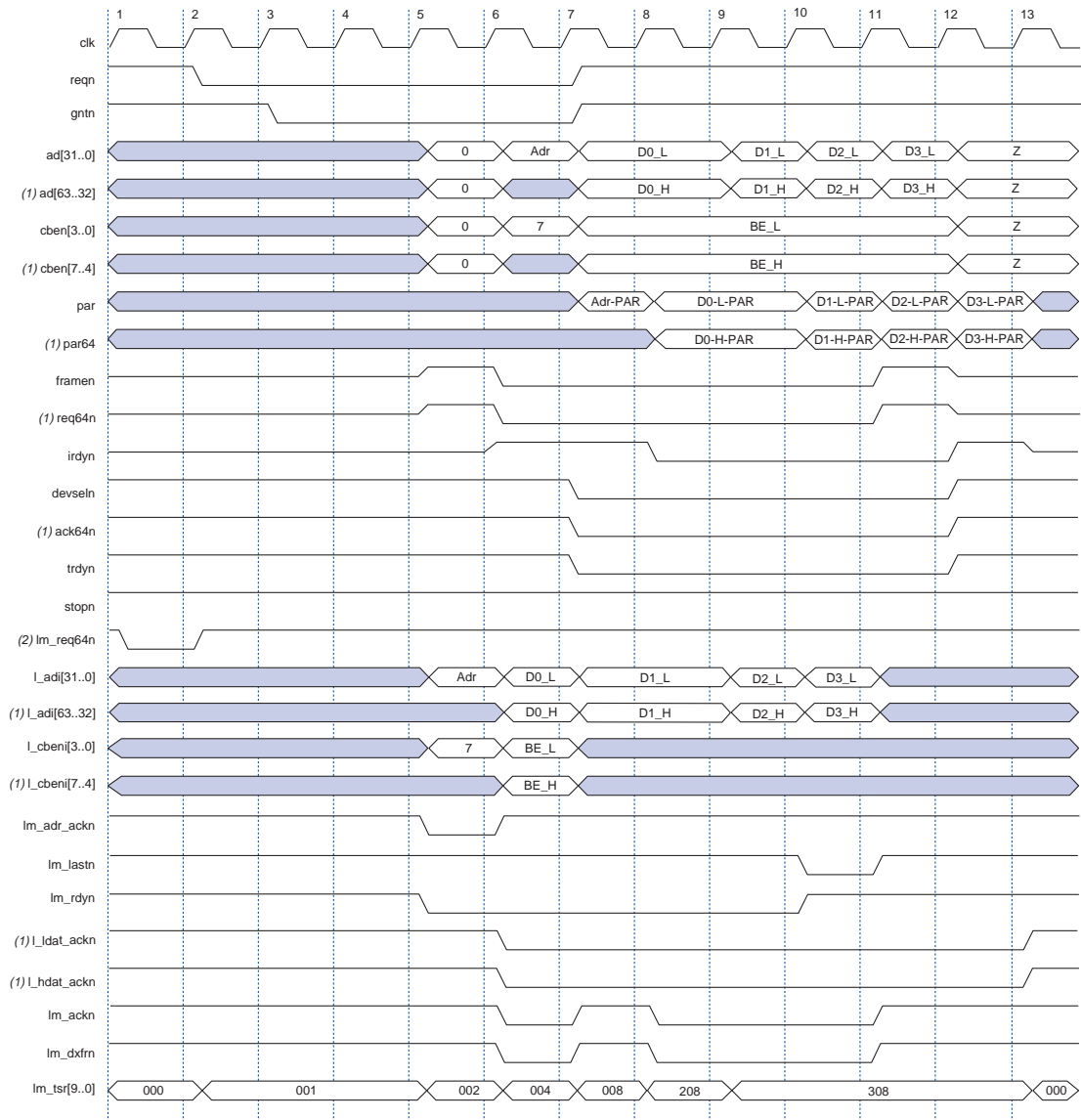
The `pci_mt64` and `pci_mt32` functions treat memory write and memory write and invalidate in the same way. Any additional requirements for the memory write and invalidate command must be implemented by the local-side application.

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

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



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

**Notes:**

- (1) This signal does not apply to `pci_mt32` for 32-bit master read transactions. For these transactions, the signal should be ignored.
- (2) For `pci_mt32`, `lm_req64n` should be exchanged with `lm_req32n` for 32-bit master transactions.

**Table 29** shows the sequence of events for a 64-bit zero wait state master burst memory write transaction.

<i>Table 29. 64-Bit Zero Wait State Master Burst Memory Write Transaction (Part 1 of 3)</i>	
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 <a href="#">Figure 22</a> 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	<p>The function turns on its output drivers, getting ready to begin the address phase.</p> <p>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>.</p> <p>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 MegaCore function asserts <code>irdyn</code> if the local side is ready to send data.</p> <p>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.</p>
6	<p>The PCI MegaCore function begins the 64-bit memory write transaction with the address phase by asserting <code>framen</code> and <code>req64n</code>.</p> <p>At the same time, the local side must provide the byte enables for the transaction on <code>l_cbeni[7..0]</code>.</p> <p>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 QWORD from <code>l_adi[63..0]</code>.</p> <p>The PCI MegaCore function asserts <code>lm_tsr[2]</code> to indicate to the local side that the PCI bus is in its address phase.</p>

**Table 29. 64-Bit Zero Wait State Master Burst Memory Write Transaction (Part 2 of 3)**

Clock Cycle	Event
7	<p>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.</p> <p>During this clock cycle, the function also asserts <code>lm_tsr[3]</code> to inform the local side that it is in data transfer 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 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>.</p>
8	<p>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 9.</p> <p>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 QWORD from <code>l_adi[63..0]</code>.</p>
9	<p>Because <code>irdyn</code> and <code>trdyn</code> are asserted, the second 64-bit data is transferred to the PCI side on the rising edge of clock 10.</p> <p>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 QWORD from <code>l_adi[63..0]</code>.</p> <p>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>.</p>

**Table 29. 64-Bit Zero Wait State Master Burst Memory Write Transaction (Part 3 of 3)**

Clock Cycle	Event
10	<p>Because <code>irdyn</code> and <code>trdyn</code> are asserted, the third 64-bit data is transferred to the PCI side on the rising edge of clock 11.</p> <p>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 QWORD from <code>l_adi[63..0]</code>. 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[63..0]</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.</p> <p>The function also asserts <code>lm_tsr[8]</code> in the same clock to inform the local side that a data phase was completed successfully on the PCI bus during the previous clock.</p>
11	<p>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 12.</p> <p>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.</p> <p>The function continues to assert <code>lm_tsr[8]</code>, informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.</p>
12	<p>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.</p> <p>The function continues to assert <code>lm_tsr[8]</code>, informing the local side that a data phase was completed successfully on the PCI bus during the previous clock.</p>
13	<p>The function deasserts <code>lm_tsr[3]</code>, informing the local side that the data transfer mode is completed.</p>

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

Figure 30 shows the same transaction as in Figure 29 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 9. Consequently, on the following clock cycle (clock 10), the `pci_mt64` or `pci_mt32` function suspends data transfer on the local side by deasserting the `lm_dxfrn` signal. Because there is no data transfer on the local side in clock 10, the function suspends data transfer on the PCI side by deasserting the `irdyn` signal in clock 11.

The timing diagram illustrates the L2 cache controller's operation over 14 clock cycles. Key signals include:

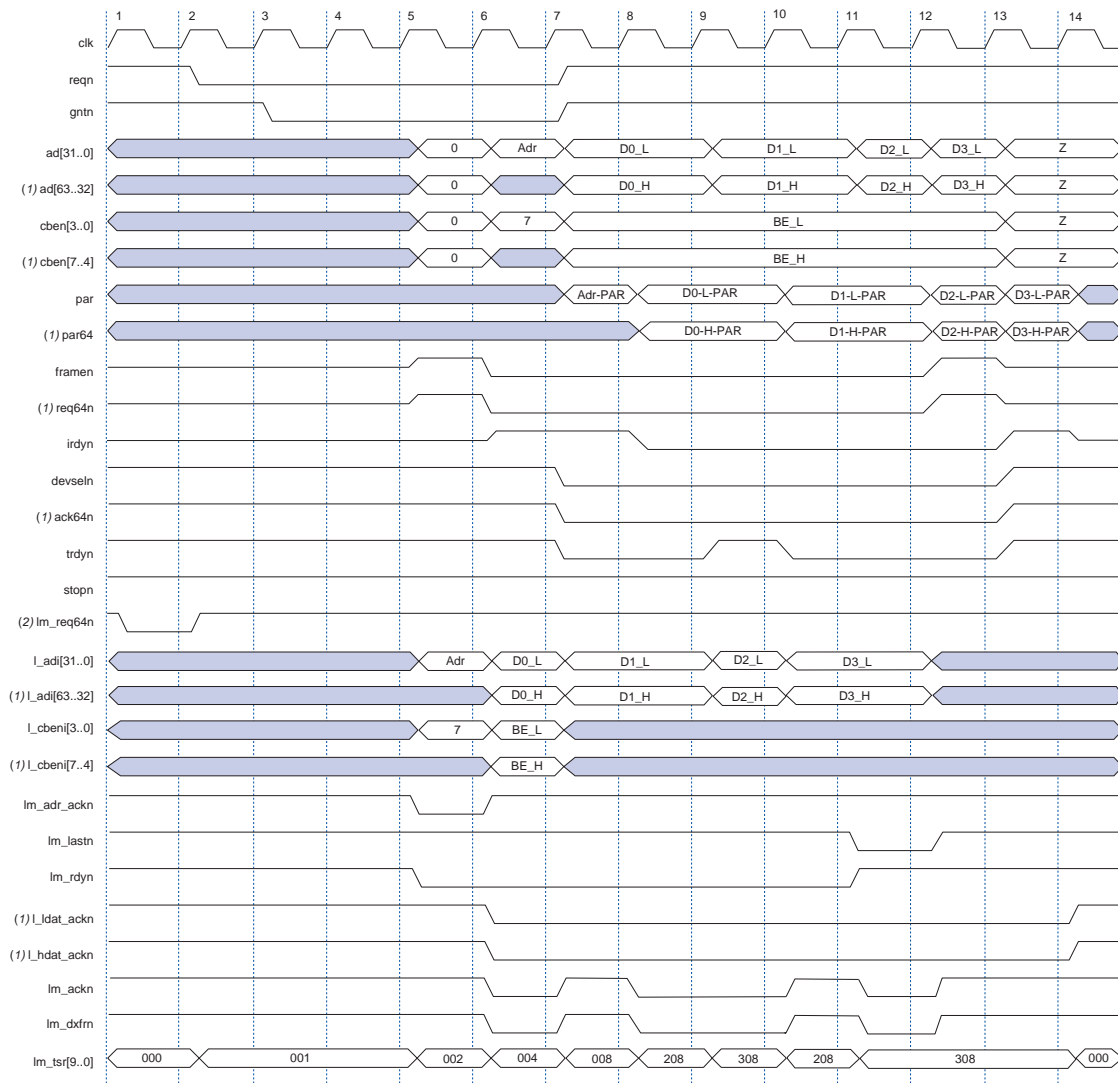
- clk**: System clock, periodic square wave.
- reqn**: Request signal, active from cycle 2 to 7.
- gntn**: Grant signal, active from cycle 3 to 7.
- ad[31..0]**: Address bus, showing the request address (0) and subsequent data addresses (D0\_L, D1\_L, D2\_L, D3\_L, Z).
- (1) adj[63..32]**: Address bus, showing the request address (0) and subsequent data addresses (D0\_H, D1\_H, D2\_H, D3\_H, Z).
- cben[3..0]**: Cache enable signal, active from cycle 2 to 7.
- (1) cben[7..4]**: Cache enable signal, active from cycle 2 to 7.
- par**: Parity signal, active from cycle 2 to 7.
- (1) par64**: Parity signal, active from cycle 2 to 7.
- framen**: Frame enable signal, active from cycle 2 to 7.
- (1) req64n**: Request signal, active from cycle 2 to 7.
- irdyn**: Invalid request signal, active from cycle 2 to 7.
- devseln**: Device select signal, active from cycle 2 to 7.
- (1) ack64n**: Acknowledge signal, active from cycle 2 to 7.
- trdyn**: Transfer request signal, active from cycle 2 to 7.
- stopn**: Stop signal, active from cycle 2 to 7.
- (2) lm\_req64n**: Local memory request signal, active from cycle 2 to 7.
- l\_adj[31..0]**: Local address bus, showing the request address (0) and subsequent data addresses (D0\_L, D1\_L, D2\_L, D3\_L, Z).
- (1) l\_adj[63..32]**: Local address bus, showing the request address (0) and subsequent data addresses (D0\_H, D1\_H, D2\_H, D3\_H, Z).
- l\_cben[3..0]**: Local cache enable signal, active from cycle 2 to 7.
- (1) l\_cben[7..4]**: Local cache enable signal, active from cycle 2 to 7.
- lm\_adr\_ackn**: Local memory address acknowledge signal, active from cycle 2 to 7.
- lm\_lastn**: Local memory last signal, active from cycle 2 to 7.
- lm\_rdyn**: Local memory request signal, active from cycle 2 to 7.
- (1) l\_idat\_ackn**: Local internal data acknowledge signal, active from cycle 2 to 7.
- (1) l\_hdat\_ackn**: Local internal data acknowledge signal, active from cycle 2 to 7.
- lm\_ackn**: Local memory acknowledge signal, active from cycle 2 to 7.
- lm\_dxfrn**: Local memory data transfer signal, active from cycle 2 to 7.
- lm\_tsr[9..0]**: Local memory transfer signal, active from cycle 2 to 7.

- (1) This signal does not apply to `pci_mt32` for 32-bit master read transactions. For these transactions, the signal should be ignored.
- (2) For `pci_mt32`, `lm_req64n` should be exchanged with `lm_req32n` for 32-bit master transactions.

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

Figure 31 shows the same transaction as in Figure 29 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 9. Consequently, on the following clock cycle (clock 10), the `pci_mt64` or `pci_mt32` function deasserts the `lm_ackn` and `lm_dxfrn` signals on the local side. Data transfer is suspended on the PCI side in clock 9 and on the local side in clock 10. Also, because `lm_lastn` is asserted and `lm_rdyn` is deasserted in clock 11, the `lm_ackn` and `lm_dxfrn` signals remain deasserted after clock 12.

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

**Notes:**

- (1) This signal does not apply to pci\_mt32 for 32-bit master read transactions. For these transactions, the signal should be ignored.
- (2) For pci\_mt32, l\_m\_req64n should be exchanged with l\_m\_req32n for 32-bit master transactions.



## 32-Bit Master Write Transactions

In master mode, the `pci_mt64` and `pci_mt32` functions support three types of 32-bit write transactions:

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

For both the `pci_mt64` and `pci_mt32` MegaCore functions, 32-bit memory write transactions are either single-cycle or burst. The 32-bit master write transactions are similar to 64-bit master write transactions, except the upper address `ad[63..32]` and the upper command/byte enables `cben[7..4]` are invalid. For `pci_mt32`, the waveforms for 32-bit memory write transactions are described in [Figures 29](#) through [31](#), excluding the 64-bit extension signals as noted, and in [Figures 33](#) and [34](#).

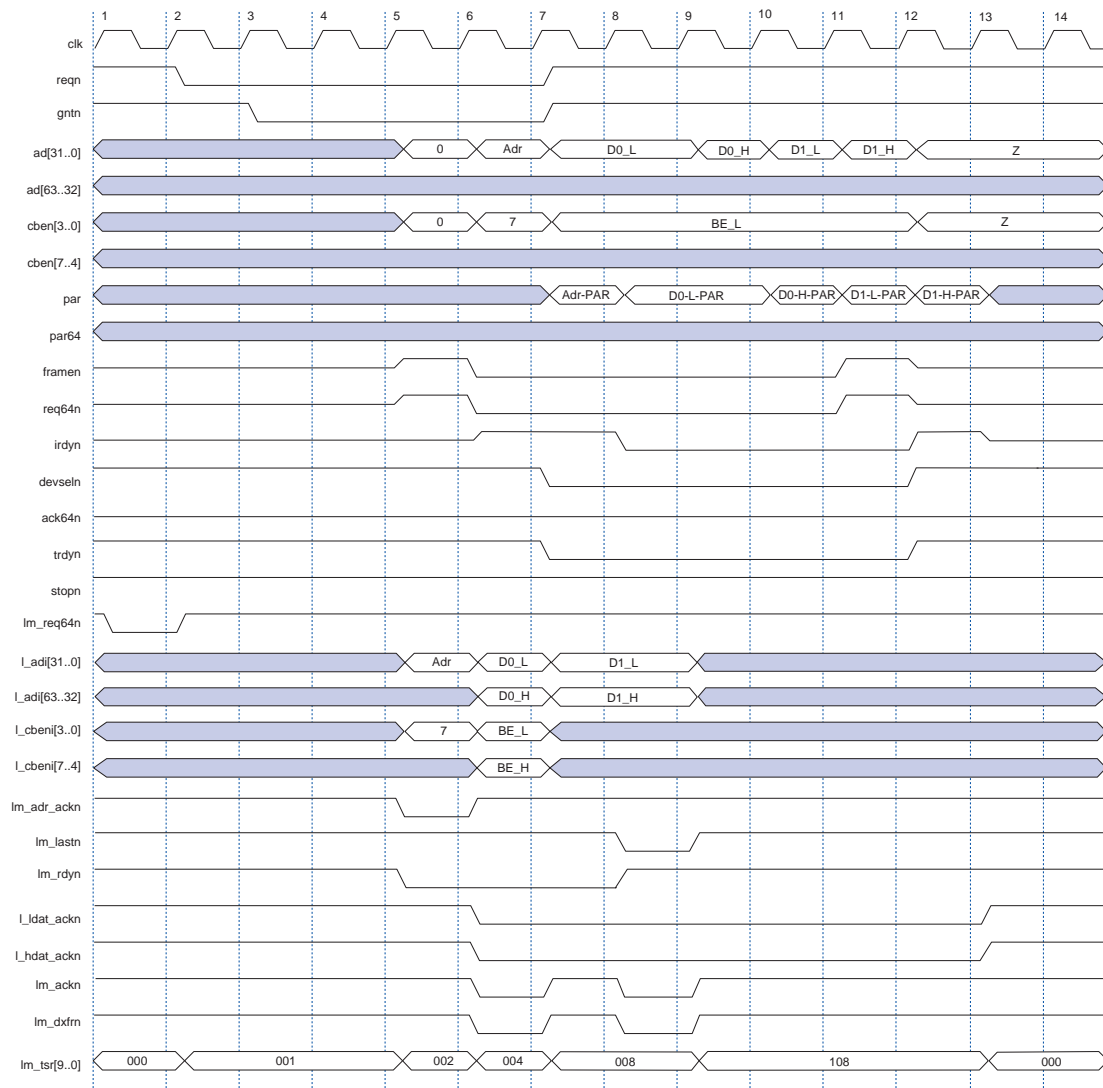
### *32-Bit PCI & 64-Bit Local-Side Master Burst Memory Write Transaction*

[Figure 32](#) shows the same transaction as in [Figure 29](#), but the PCI target cannot transfer 64-bit transactions. This figure applies to `pci_mt64` only. 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 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 `l_adi[31..0]` on the PCI side first, and the upper 32-bit data `l_adi[63..32]` afterwards.

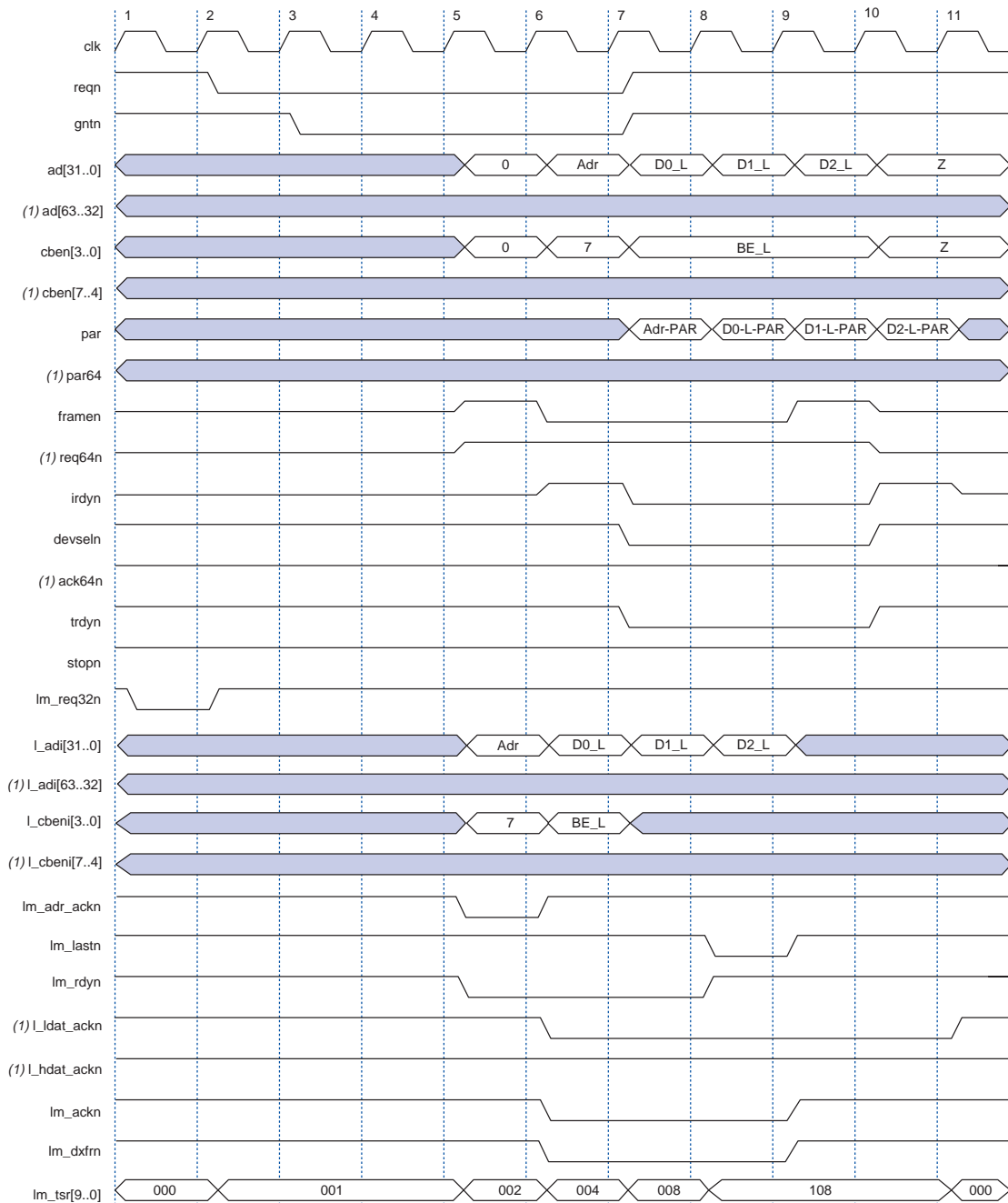
Figure 32. 32-Bit PCI &amp; 64-Bit Local-Side Master Burst Memory Write Transaction



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

Figure 33 shows the same transaction as in Figure 29, but the local side master interface requests a 32-bit transaction by asserting `lm_req32n`. 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 33. 32-Bit PCI &amp; 32-Bit Local-Side Master Burst Memory Write Transaction

**Note:**

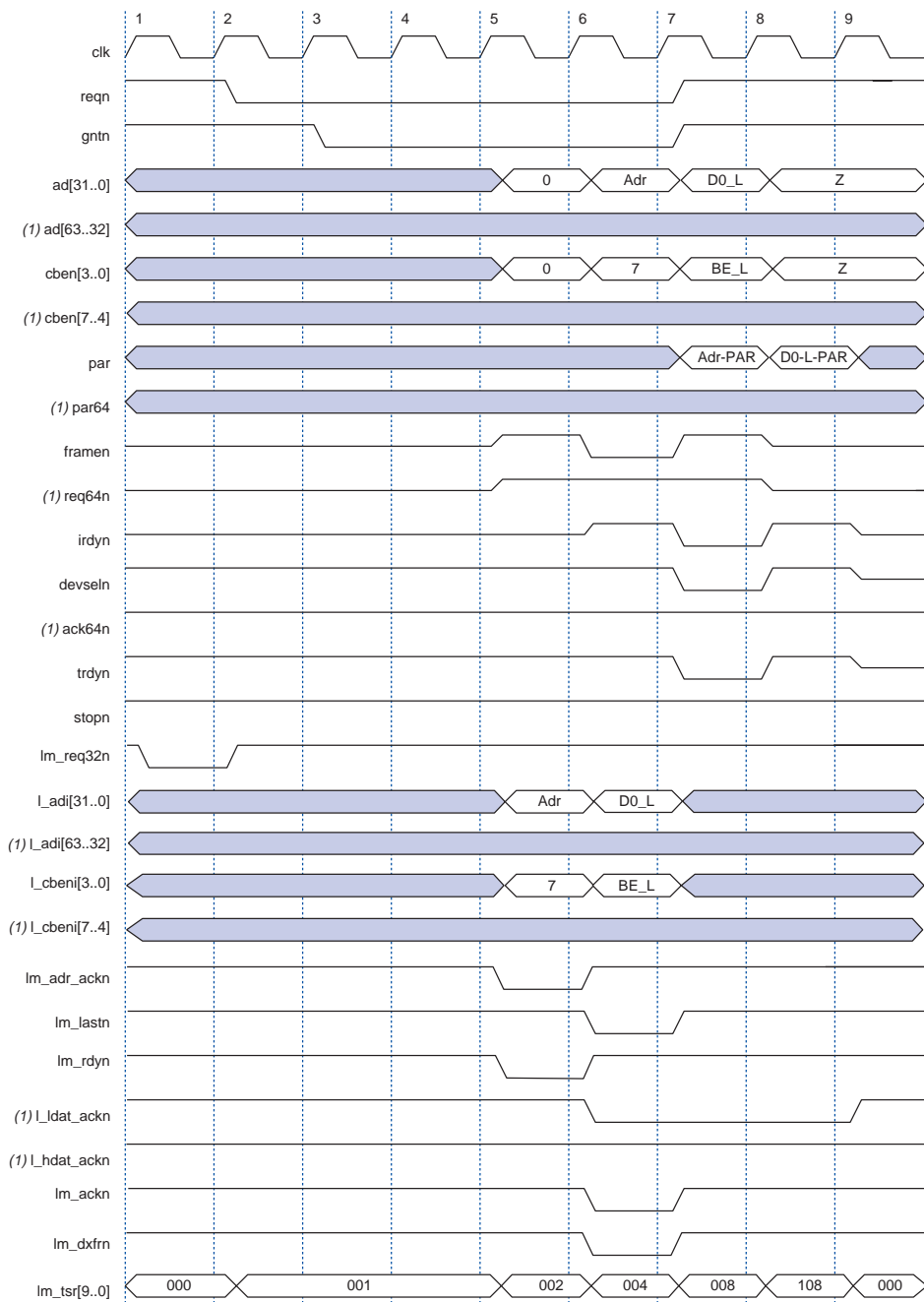
- (1) This signal does not apply to `pci_mt 32` for 32-bit master read transactions. For these transactions, the signal should be ignored.

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

Figure 34 shows the same transaction as in Figure 33, but the local side master interface transfers only one data phase. This figure applies to both the `pci_mt64` and `pci_mt32` MegaCore functions, excluding the 64-bit extension signals as noted for `pci_mt32`. This waveform also applies to the following types of single-cycle transactions:

- I/O write
- Configuration write

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

**Note:**

- (1) This signal does not apply to `pci_mt32` for 32-bit master read transactions. For these transactions, the signal should be ignored.

## 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 it ends the transaction because the latency timer expired, it asserts `lm_tsr[4]` (`tsr_lat_exp`) until the beginning of the next master transaction.



The PCI MegaCore functions allow the option of disabling the latency timer for embedded applications. See “Parameters” on [page 37](#) for more information.

### *Retry*

The target issues a retry by asserting `stopn` and `devseln` during the first data phase. When the `pci_mt64` or `pci_mt32` function detects a retry condition (see “[Retry](#)” on [page 96](#) for details), it ends the cycle and asserts `lm_tsr[5]` until the beginning of the next transaction. This process informs the local-side device that it has ended the transaction because the target issued a retry.



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

### *Disconnect Without Data*

The target device issues a disconnect without data if it is unable to transfer additional data during the transaction. The signal pattern for this termination is described in “[Disconnect](#)” on page 98. When the `pci_mt64` or `pci_mt32` function ends the transaction because of a disconnect without data, it asserts `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 98. When the `pci_mt64` or `pci_mt32` function ends the transaction because of a disconnect with data, it asserts `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 103. When the `pci_mt64` or `pci_mt32` function ends the transaction because of a target abort, it asserts the `tabort_rcvd` signal, which is the same as the PCI status register bit 12. Therefore, the signal remains asserted until it is reset by the host.

### *Master Abort*

The `pci_mt64` or `pci_mt32` function terminates the transaction with a master abort when no target claims the transaction by asserting `devseln`. Except for special cycles and configuration transactions, a master abort is considered to be a catastrophic failure. When a cycle ends in a master abort, the `pci_mt64` or `pci_mt32` function informs the local-side device by asserting the `mabort_rcvd` signal, which is the same as the PCI status register bit 13. Therefore, the signal remains asserted until it is reset by the host.

## Host Bridge Operation

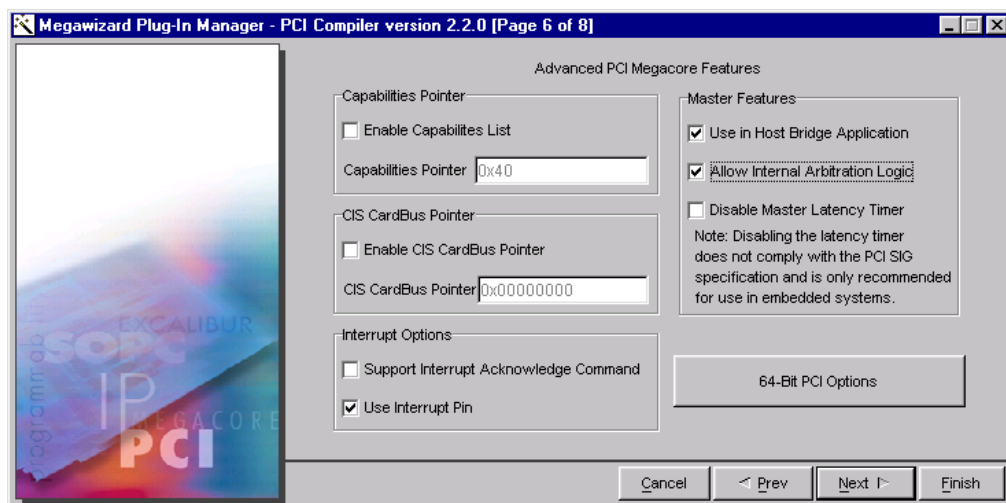
This section describes using the `pci_mt64` and `pci_mt32` MegaCore functions as a host bridge application in a PCI system. The `pci_mt64` and `pci_mt32` functions support the following advanced master features, which should be enabled when using the functions in a host bridge application:

- Use in host bridge application
- Allow internal arbitration logic



The host bridge features can be enabled through the **Advanced PCI MegaCore Features** screen of the PCI wizard. See [Figure 35](#).

Figure 35. Advanced Master Features



## Using the PCI MegaCore Function as a Host Bridge

Selecting the **Use in Host Bridge Application** option 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.

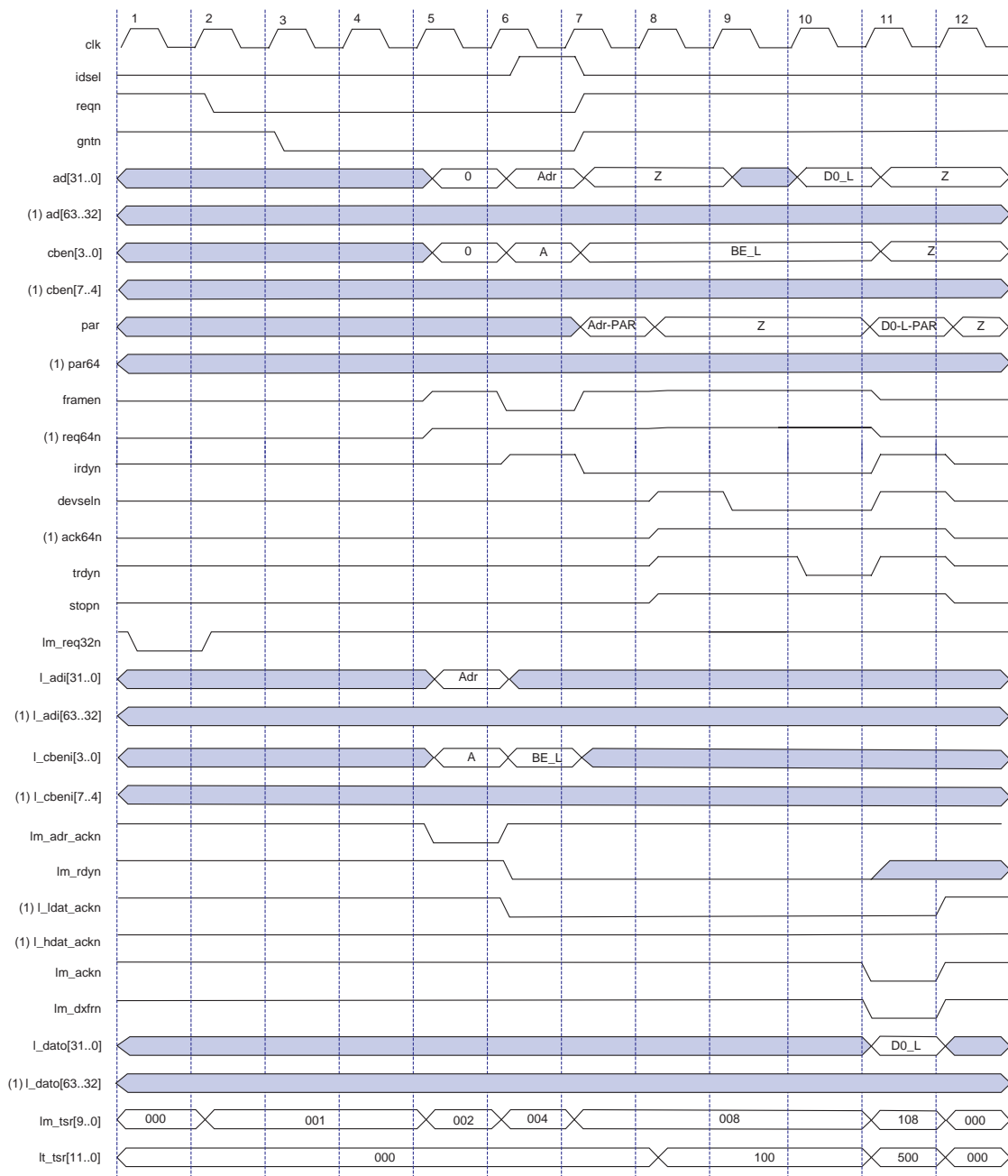


To perform configuration transactions to internal configuration space, the `idsel` signal must be connected following the PCI specification requirements.

*PCI Configuration Read Transaction from the pci\_mt64 Local Master Device to the Internal Configuration Space*

Figure 36 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 28 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 36 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 36. Configuration Read from Internal Configuration Space in Self-Configuration Mode

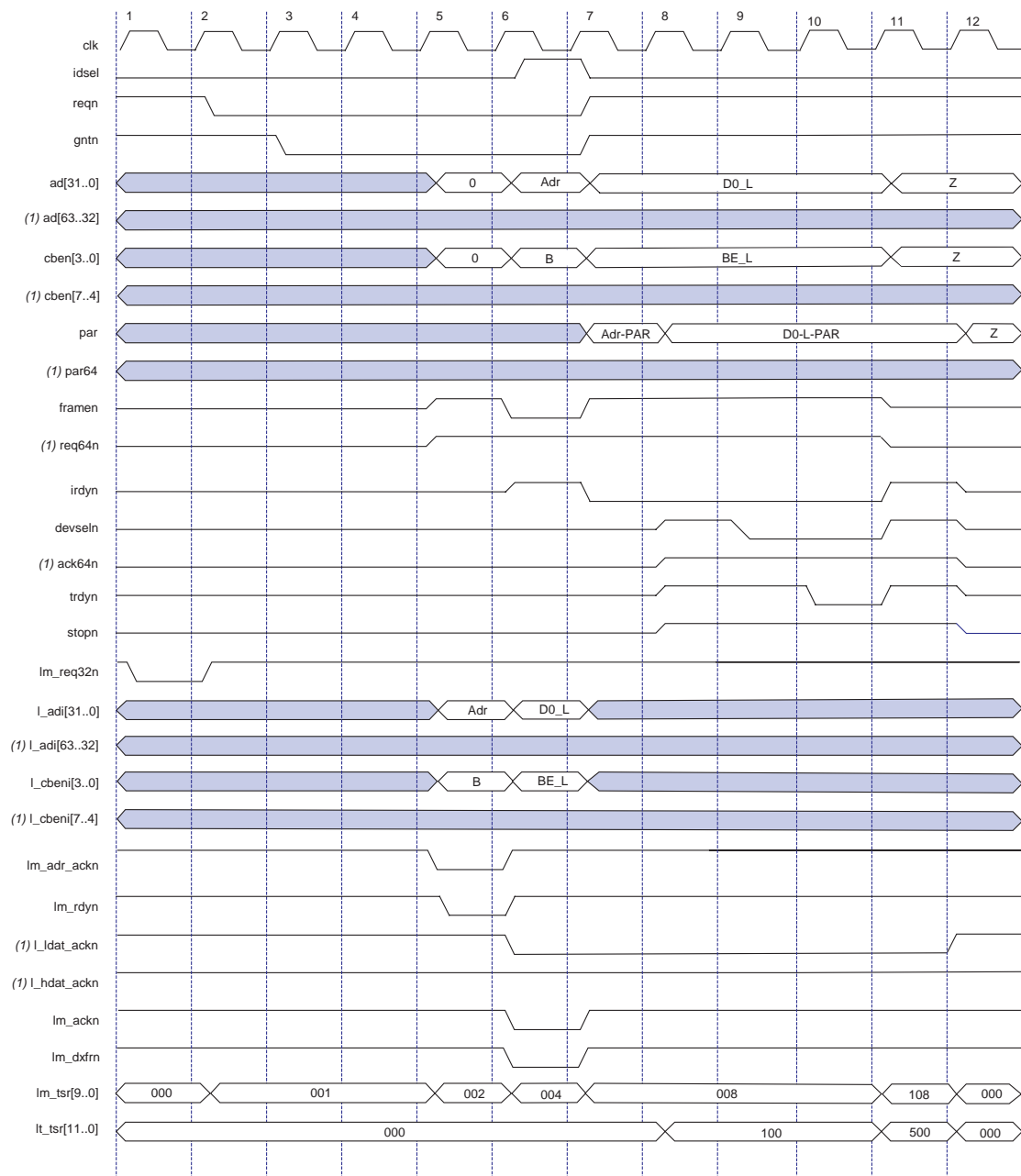
**Note:**

- (1) This signal does not apply to `pci_mt32` for 32-bit master read transactions. For these transactions, the signal should be ignored.

*PCI Configuration Write Transaction from the pci\_mt64 Local Master Device to the Internal Configuration Space*

Figure 37 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 34 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 37 applies to both the pci\_mt64 and pci\_mt32 MegaCore functions, excluding the 64-bit extension signals as noted for pci\_mt32.

Figure 37. Configuration Write to Internal Configuration Space in Self-Configuration Mode

**Note:**

- (1) This signal does not apply to `pci_mt32` for 32-bit master write transactions. For these transactions, the signal should be ignored.

## Implementing Internal Bus Arbitration Logic

Many applications that utilize the `pci_mt64` or `pci_mt32` MegaCore functions as a host bridge will implement other central resource functionality in the same PLD as the PCI interface. The **Allow Internal Arbitration Logic** feature enables the design to include the PCI bus arbiter in the same PLD 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. Enabling the **Allow Internal Arbitration Logic** option removes the tri-state buffer from the `reqn` signal output, which allows the signal to be connected to internal PLD logic. The `reqn` signal will not require the use of a device I/O or board traces with the **Allow Internal Arbitration Logic** option enabled.

## 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 `pci_mt64` and `pci_t64` 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 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.

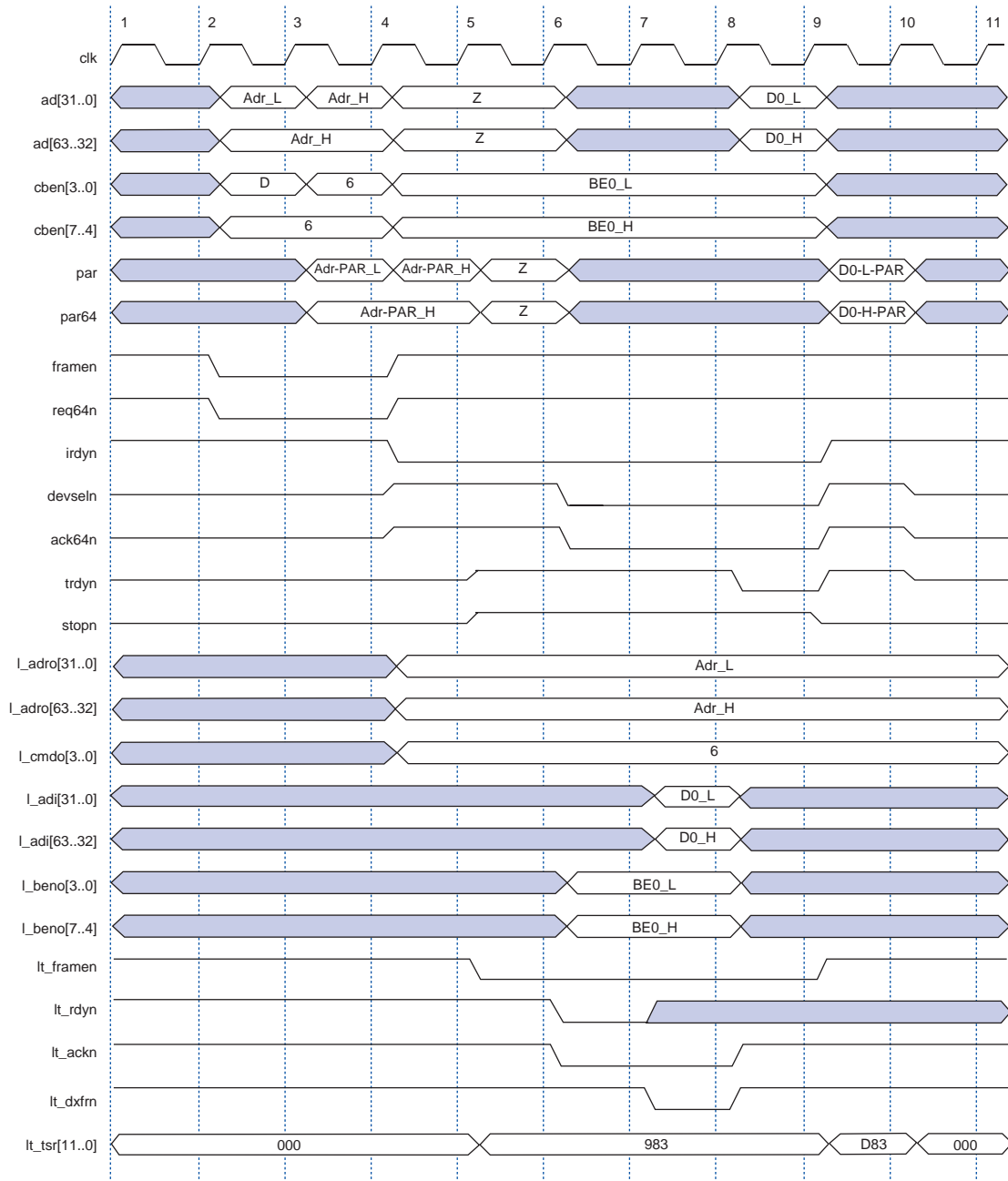
#### *64-Bit Address, 64-Bit Data Single-Cycle Target Read Transaction*

Figure 38 shows the waveform for a 64-bit address, 64-bit data single-cycle target read transaction. Figure 38 is exactly the same as Figure 1, except that Figure 38 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 148 are applicable for 64-bit addressing transactions, except for the differences described in the previous paragraph.

Figure 38. 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_adi[63..0]` bus, the DAC command on `l_cbeni[3..0]`, and the transaction command on the `l_cbeni[7..4]`. When the PCI bus arbiter grants the bus to the `pci_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]`.

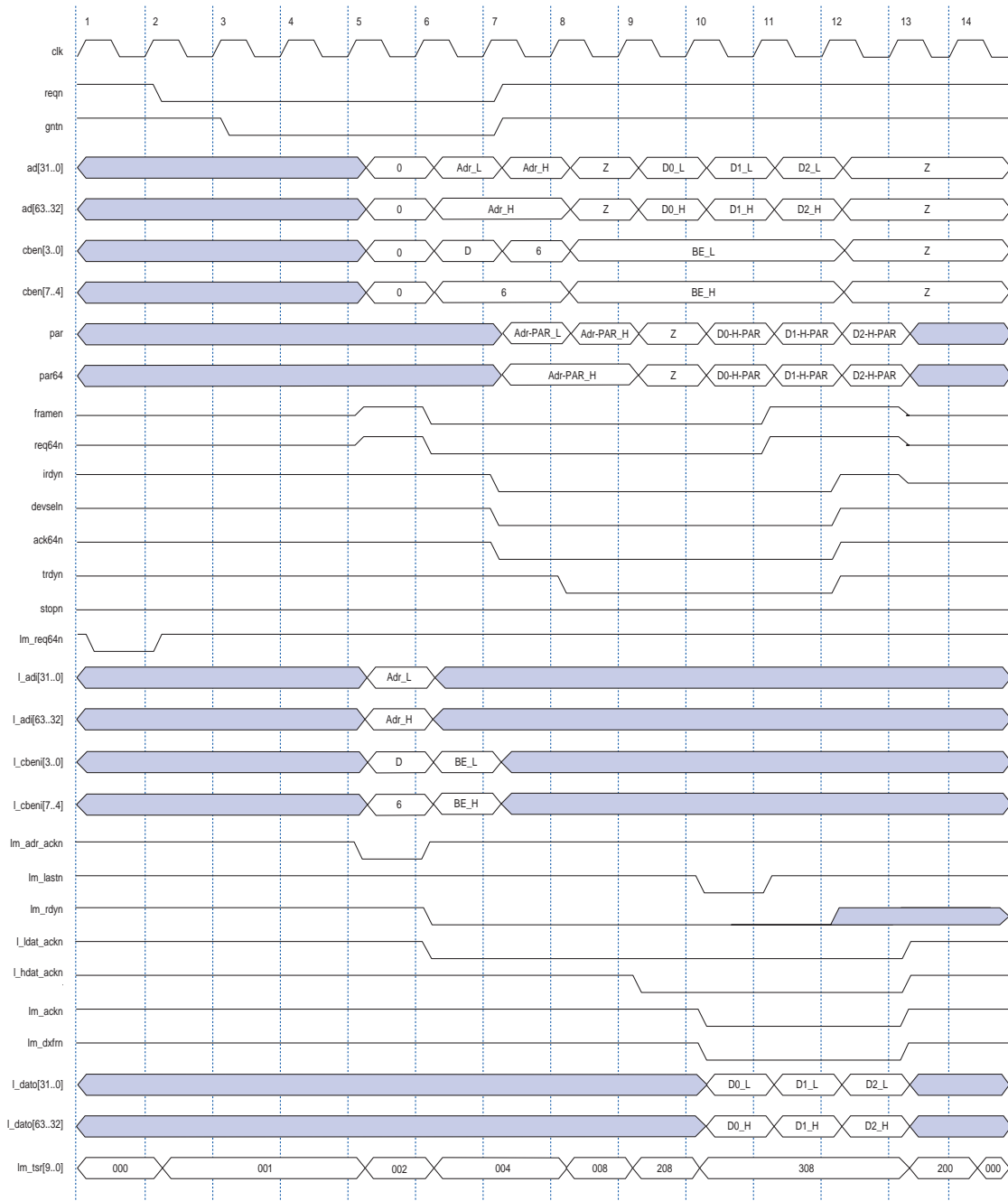
### *64-Bit Address, 64-Bit Data Master Burst Memory Read Transaction*

Figure 39 shows the waveform for a 64-bit address, 64-bit data master burst memory read transaction. Figure 39 is exactly the same as Figure 22, except that Figure 39 has two address phases (as described in the previous paragraph).



All 32-bit addressing transactions described in “Master Mode Operation” on page 151 are applicable for 64-bit addressing transactions, except for the differences described in the previous paragraph.

Figure 39. 64-Bit Address, 64-Bit Data Master Burst Memory Read Transaction



The Altera® `pci_mt64`, `pci_mt32`, `pci_t64`, and `pci_t32` MegaCore® functions are fully compliant with the **PCI Local Bus Specification, Revision 2.2**. These functions are designed to meet 66-MHz PCI timing requirements when implemented in Altera FPGAs that are 66-MHz PCI compliant.

## Pipelining the Local-Side Design

The designer should design the local-side logic to operate at 66 MHz to maintain 66-MHz internal timing for all logic that uses the 66-MHz PCI clock. If the local-side design is pipelined, the overall design (user logic plus the PCI MegaCore function) should compile for 66-MHz register-to-register performance.

Altera designed the PCI MegaCore functions with four to six logic levels for most of the register-to-register paths. Higher levels of logic would have made it more difficult to compile the design and still achieve 66-MHz performance on the PCI clock.

## Designing to the PCI Function Local Side

Altera designed the PCI function local-side signals to offer maximum design flexibility. The PCI function's local side provides input and output signals that inform the designer's local-side logic of the transactions occurring on the PCI bus.

The local-side outputs from the PCI MegaCore functions are often registered. If the signals are not registered, there will be at most one level of logic before the designer's local-side logic has access to the signal.

The local-side inputs to the PCI MegaCore functions do not include registers at the periphery of the function. In a 66-MHz application, the designer's local side logic should register the inputs to the local side of the PCI MegaCore function. Additionally, the local master `lm_rdyn` and `lm_lastn` signals and the local target `lt_rdyn` and `lt_discn` signals are likely to affect the design.

## Design Examples

The `pci_mt64` and `pci_mt32` reference designs—which are included with the PCI compiler—can be used as guidelines when designing to the PCI MegaCore function local side. Refer to the **PCI Compiler Data Sheet, FS 10: `pci_mt64` MegaCore Function Reference Design**, and **FS 12: `pci_mt32` MegaCore Function Reference Design** for more information on the reference designs.



This Appendix provides detailed information about obtaining and using PCI Constraint files. You should integrate the PCI constraint into your project to ensure that the project achieves PCI timing requirements. PCI constraints are provided as a Tool Command Language (Tcl) file (.tcl); if you run the Tcl script in the Quartus II software, it generates the project-specific .csf and .esf files.

## PCI Constraint File Contents

Constraint files may include one or more of the following assignments:

- PCI Signal Pin Assignments
- Timing Assignments
- Logic Option Assignments
- Logic Location Assignments

The PCI Constraint files assume that you have not set any other project specific assignments for your project. You must use the PCI constraints before you make any other constraints in your project. Upon using the Altera PCI constraint files, all other constraints are removed and you must manually add any project specific assignments including The User Library settings.



Sample constraint files are provided at `<path>\pci_compiler_v2.2.0\<PCI core>\const_files`. Additional constraint files can be obtained at [http://www.altera.com/pci\\_cf](http://www.altera.com/pci_cf).

## Generate a Constraint File for Your Project

The Altera-provided PCI constraint files do not include assignments that are specific to your project. You can use the PCI compiler wizard to generate a project-specific constraint file. See “Getting Started” on page 9. for more information on using the wizard.

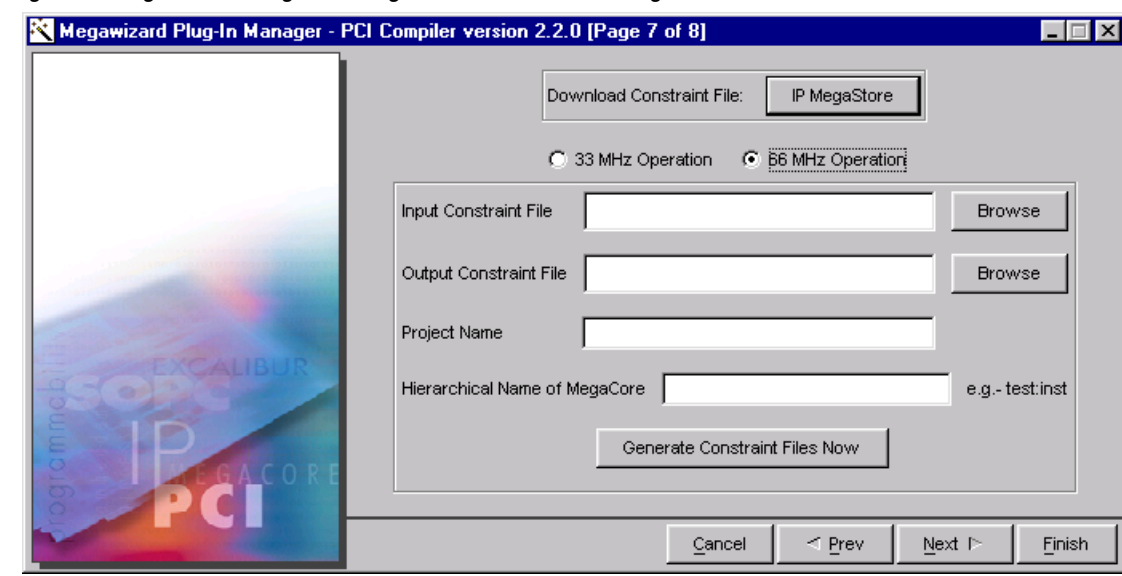


PCI constraint files are specific to your project, including device speed, density, package, MegaCore version, and version of the Quartus II software.

The constraint files contain generic text strings, which the PCI compiler wizard replaces with your project-specific information. The following screen shot shows the PCI compiler wizard page in which you specify the generic constraint files the wizard should convert to project-specific constraint files. The wizard uses the information provided in this page to replace specific generic text strings in the Altera-provided constraint files to generate a project-specific constraint file.

The information described in the following sections is required for the PCI compiler wizard to customize a generic constraint file for your project as shown in [Figure 1](#).

**Figure 1. MegaWizard Plug-In Manager Constraint File Settings**



## PCI System Speed

A setting of 66 or 33 MHz is required to ensure that the appropriate timing constraints are annotated to the constraint file. The following table shows the values used for the specified generic text strings depending on the PCI system speed selected.

<i>Table 1. PCI System Speed Descriptions (Part 1 of 2)</i>			
Text	Description	PCI System Speed	
		66 MHz	33 MHz
pci_qfmax	Maximum Frequency	66 MHz	33MHz

*Table 1. PCI System Speed Descriptions (Part 2 of 2)*

Text	Description	PCI System Speed	
		66 MHz	33 MHz
pci_tco	Clock-to-output	6.0 ns	11.0 ns
pci_toff	Time Off	14.0 ns	28.0 ns
pci_tsu	Setup Time	3.0 ns	7.0 ns
pci_th	Hold Time	0.0 ns	0.0 ns
pci_ptp	Setup time for point-to-point signals	5.0 ns	10.0 ns

## Input Constraint File

This field should contain the path to the generic file that you obtained from Altera.

## Output Constraint File

This field should contain the path to the output project-specific constraint file or Tcl script file.

## Project Name

The wizard replaces the project name string from this field to replace the string `chip_name` in the generic constraint file. You should enter the top-level name for your design files. For example, if your top-level design is called `top.edf`, you should set the project name to `top`.

## Hierarchical Name of MegaCore

Replace the string in the Hierarchical Name of MegaCore field with the hierarchical path to the PCI core in your project. The hierarchical path can be determined using the Hierarchy Display in the Quartus II software. The hierarchical path provided must be the path where you have instantiated the wrapper file generated by the PCI Compiler MegaWizard. For example a path of `|my_pci_top:u1` indicates that you have instantiated the output of the PCI compiler wizard `my_pci_top` into your top level design with the instance `u1`. As another example you should provide the following string `|pci_int:u2|my_pci_top:u1` if the wrapper file generated by the PCI compiler wizard was instantiated in the file `pci_int` and that was instantiated in your top-level design file. You must provide the hierarchical path for the wrapper file because the wizard automatically appends the remaining hierarchy for the `pci_mt64` MegaCore function when customizing the generic constraint file for your project. If you are using the wrapper file generated by the PCI compiler wizard as your top-level design file, you should leave this field blank.



If you provide an incorrect hierarchical path, the `.tcl` file will appear to work correctly. However, you will not have the proper constraints set for your project and you may not need PCI timing requirements. In that case, the Quartus II software will generate many warnings indicating that you have made an assignment to a node that does not exist.

## How to Use PCI Tcl Scripts in the Quartus II Software

The `.tcl` file contains all necessary information to create a CSF and ESF. Use the following steps to generate a project-specific CSF and ESF.

1. Generate a project-specific Tcl file using the PCI compiler wizard as indicated above
2. Type `source <Tcl script filename>` in the Quartus II Tcl Console. For example:

```
source c:\altr_app\pci_top.tcl
```



The Tcl file deletes the current project's CSF and ESF, and generates a new CSF and ESF that includes the PCI MegaCore function assignments.

After you have integrated the assignments into your project, add additional project assignments to your CSF and ESF.



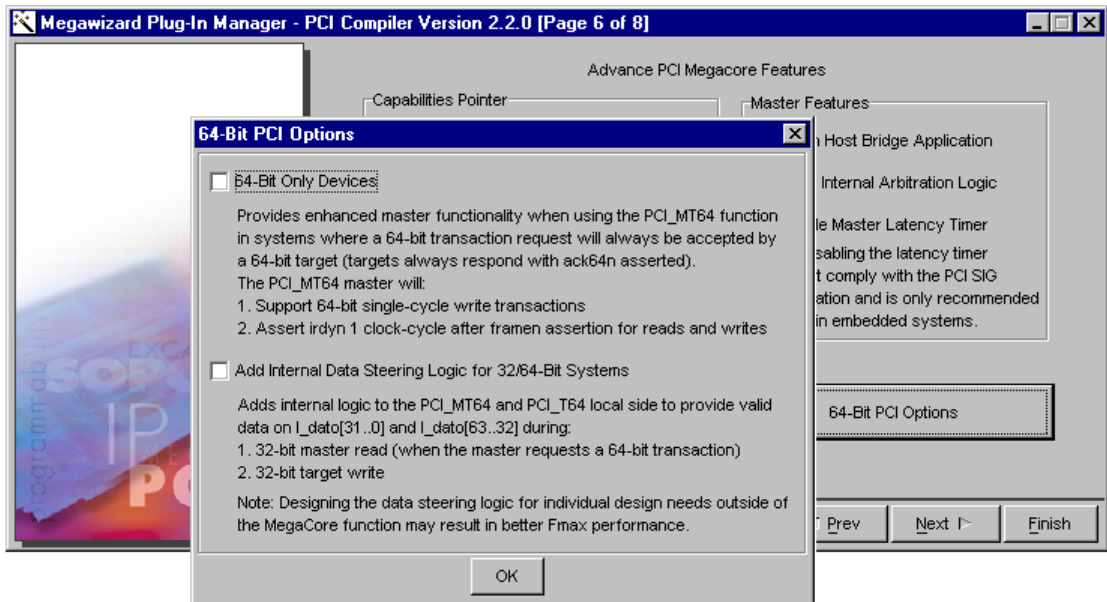
### Introduction

The PCI MegaCore® functions provided with the PCI Compiler version 2.2.0 include many new features that enhance functionality, improve system performance, and use resources more efficiently. This white paper discusses the following two PCI MegaCore function options that are specific to 64-bit PCI systems:

- 64-Bit Only Devices
- Add Internal Data Steering Logic for 32/64-Bit Systems

To enable these options, click the **64-Bit PCI Options** button on the sixth page of the wizard. These options are disabled by default (see [Figure 1](#)).

*Figure 1. Turning On 64-Bit PCI Options with the Wizard*



### 64-Bit Only Devices Option

The **64-Bit Only Devices** option applies to the pci\_mt64 MegaCore function master mode operation. This option can be used if both of the following items apply to your system:

- You are using an embedded system that has only 64-bit PCI devices
- All 64-bit master transactions are claimed by 64-bit targets that respond with `ack64n` asserted

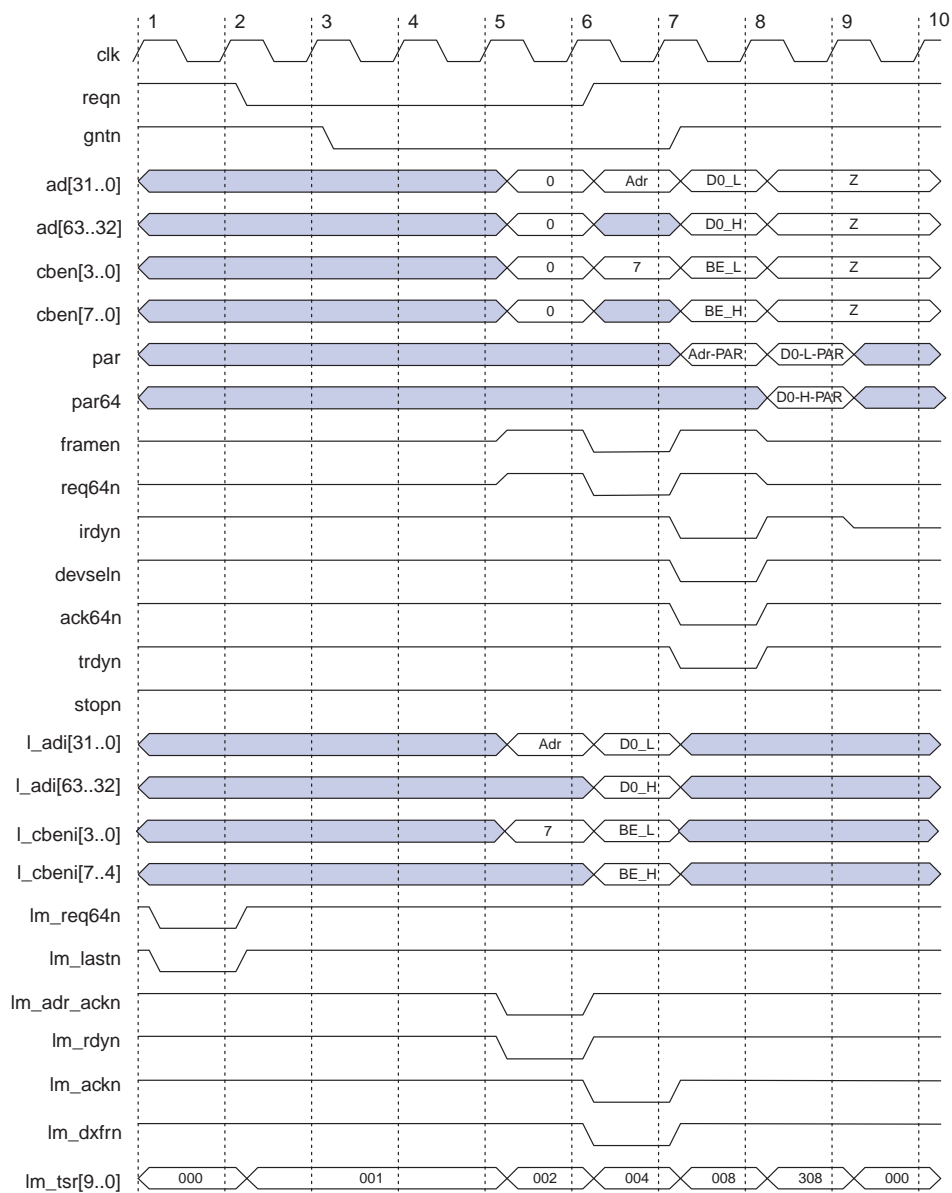
When you turn on the **64-Bit Only Devices** option, the `pci_mt64` MegaCore function assumes that all 64-bit master transactions are claimed by a 64-bit target with `ack64n` asserted. The `pci_mt64` master can then:

- 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` 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. By using the **64-Bit Only Devices** option, the `pci_mt64` function can support 64-bit single-cycle master write transactions because the target is guaranteed to be a 64-bit device. [Figure 2](#) shows an example of a single-cycle write transaction where the `pci_mt64` function is the master.

Figure 2. PCI 64-Bit Single-Cycle Master Memory Write Operation



In addition to providing support for PCI 64-bit single-cycle master memory writes, the **64-Bit Only Devices** option allows `pci_mt64` to provide the same low initial latency on `irdyn` for 64-bit master write transactions that is provided for all other master transactions. In other words, the `pci_mt64` function does not delay `irdyn` assertion to wait for the target `devseln` assertion. Instead, during 64-bit master write transactions, initial `irdyn` assertion is one clock cycle after the local side transfers the first QWORD (`lm_dxfrn` is asserted). This process is shown in [Figure 2](#).

## Add Internal Data Steering Logic for 32/64-Bit Systems Option

To comply with the *PCI Local Bus Specification, Revision 2.2*, the `pci_mt64` and `pci_t64` functions must support 32-bit memory transactions. To interface to a 64-bit local side, these functions provide the option of enabling data steering logic on `l_dato[63..32]` and `l_beno[7..4]`. In the event of 32-bit PCI operation and 64-bit local side operation, this option drives data received from the PCI bus on both the low data and byte enable output signals (`l_dato[31..0]` and `l_beno[3..0]`, respectively) and the high data and byte enable output signals (`l_dato[63..32]` and `l_beno[7..4]`, respectively). The local side logic must use the `l_ldat_ackn` and `l_hdat_ackn` signals to write data and byte enables into the low and high DWORD storage areas, respectively.

To enable the data steering logic, turn on the **Add Internal Data Steering Logic for 32/66-Bit Systems** option.



The data steering logic adds delay to the critical timing path for 66-MHz operation. This logic may be better integrated into the local-side application; therefore, Altera recommends that you leave this option disabled and instead add the functionality to your local side logic.

If any of the following statements is true for your application, the data steering logic should be implemented either inside the MegaCore function or in local side logic:

- The application is an add-on card that can operate in either a 64-bit or 32-bit PCI slot
- A 32-bit target may claim a 64-bit memory transaction initiated by the `pci_mt64` function
- A 32-bit master may initiate a transaction that is claimed by the `pci_mt64` or `pci_t64` function

Figure 3 shows an example of the `pci_mt64` and `pci_t64` function target behavior during a PCI 32-bit memory write transaction with the data steering logic disabled. Figure 4 shows the same transaction with the data steering logic enabled. The behavior of `l_dato[63..32]` and `l_beno[7..4]` differs between Figure 3 and Figure 4.

Figure 3. PCI Target 32-Bit Burst Memory Transaction with Data Steering Logic Disabled

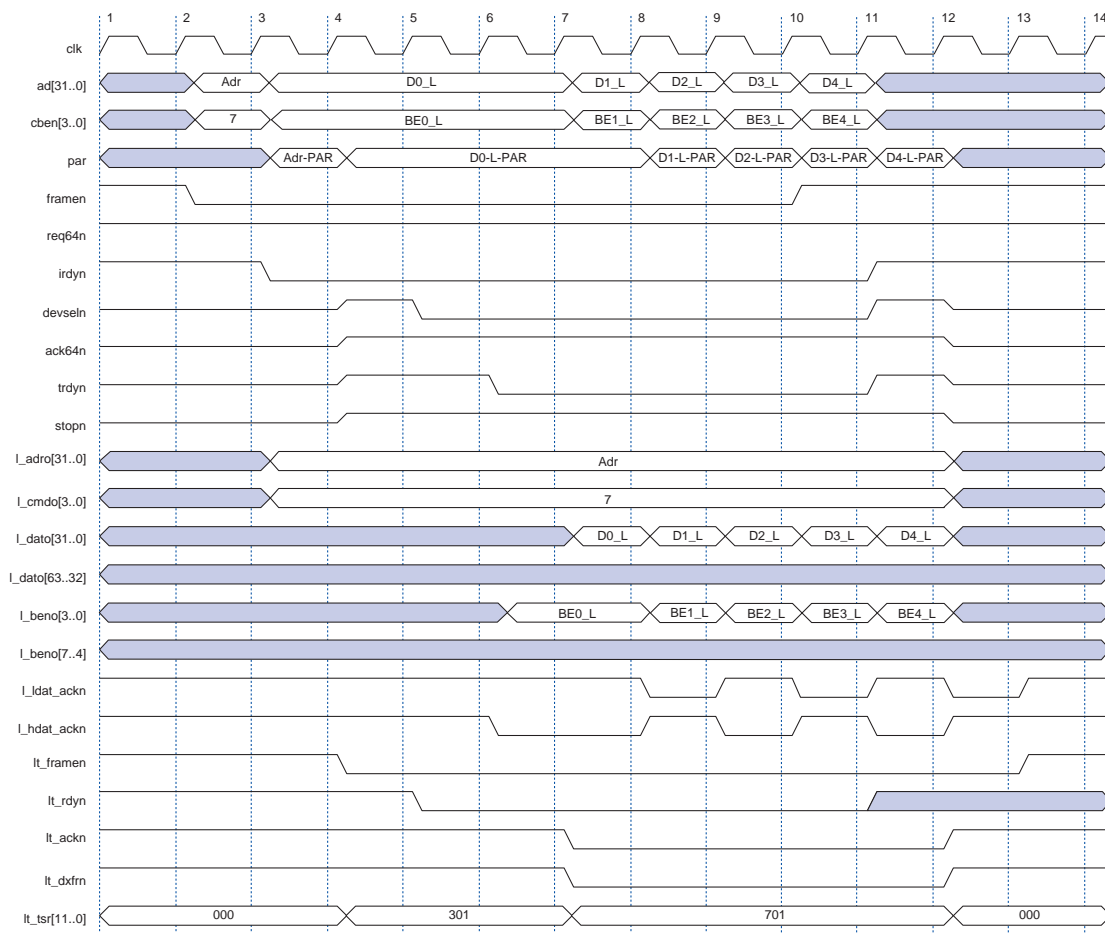


Figure 4. PCI Target 32-Bit Burst Memory Transaction with Data Steering Logic Enabled

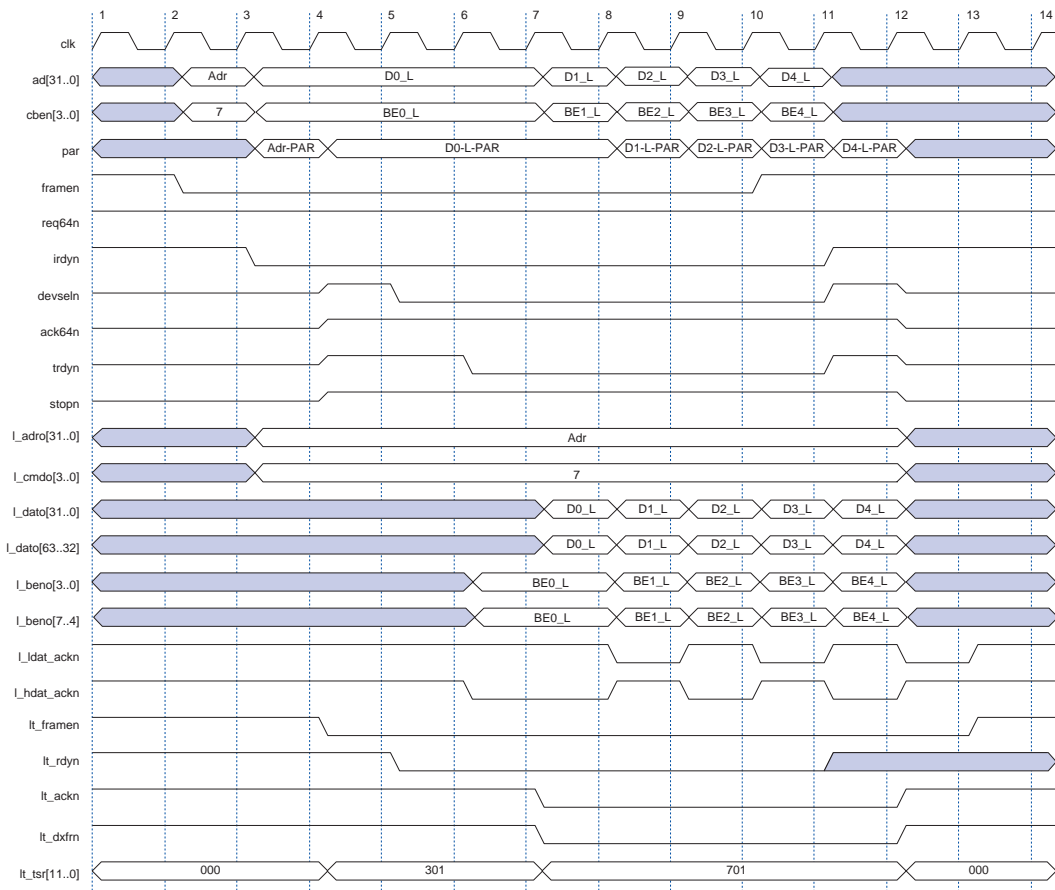
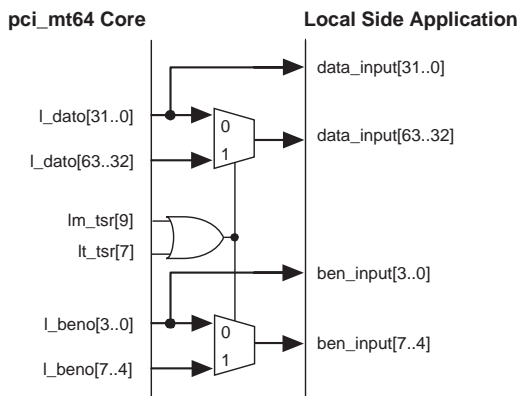


Figure 5 shows a block diagram of the data steering logic needed when using the `pci_mt64` and `pci_t64` functions in 32- or 64-bit systems. This data steering logic can be added and optimized in the local side application logic for 66-MHz operation.

*Figure 5. Logic Required for 32-bit PCI Transfers to 64-Bit Local Side Application*







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

*Table 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 <a href="#">(1)</a>	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 <a href="#">(1)</a>	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.

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

Name	Format	Default Value	Description
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.
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 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 "Base Address Registers" on page 58.
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 "Base Address Registers" on page 58.
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 "Base Address Registers" on page 58.
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.

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

Name	Format	Default Value	Description
EXP_ROM_BAR	String	H"FF000000"	Expansion ROM. This value controls the number of bits in the expansion ROM BAR that are read/write and will be decoded during a memory transaction.
HARDWIRE_BAR $n$	Hexadecimal	H"FF000000"	<p>Hardwire base address register. <math>n</math> corresponds to the base address register number and can be from 0 to 5.</p> <p>HARDWIRE_BAR<math>n</math> 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<math>n</math>_ENA bit is not set to 1.</p> <p>When the corresponding HARDWIRE_BAR<math>n</math>_ENA bits are set to 1, the function returns the value in HARDWIRE_BAR<math>n</math> 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<math>n</math> parameter. The corresponding BAR<math>n</math> parameter is still used to define the programmable setting of the individual BAR such as address space type and number of decoded bits.</p>
HARDWIRE_EXP_ROM	Hexadecimal	H"FF000000"	<p>Hardwire expansion ROM BAR.</p> <p>HARDWIRE_EXP_ROM is the default expansion ROM base address. This parameter is ignored when HARDWIRE_EXP_ROM_ENA is set to 0.</p> <p>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.</p> <p>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.</p>

Table 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 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 <b>PCMCIA Specification, version 2.2</b> . 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.

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

Name	Format	Default Value	Description
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.
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 <i>intan</i> . When the parameter is set to H"00", <i>intan</i> will be stuck at $V_{CC}$ and the <i>l_irqn</i> 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:**

- (1) These parameters affect master functionality, therefore, they only affect the *pci\_mt64* and *pci\_mt32* 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 shows the bit definition for ENABLE\_BITS.

Table 2. Bit Definition of the ENABLE\_BITS Parameter (Part 1 of 4)

Bit Number	Bit Name	Default Value	Definition
5..0	HARDWIRE_BAR <sub>n</sub> _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.

Table 2. Bit Definition of the *ENABLE\_BITS* Parameter (Part 2 of 4)

Bit Number	Bit Name	Default Value	Definition
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 <code>EXP_ROM_BAR</code> 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 <code>H"00000000"</code> when the expansion ROM BAR is read.
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 <code>CAP_PTR</code> .
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 <code>H"00000000"</code> during a configuration read to the <code>CIS_PTR</code> 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 an APEX or a FLEX device is used to implement the function and is also used to implement a PCI bus arbiter, 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. Bit Definition of the *ENABLE\_BITS* Parameter (Part 3 of 4)

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> cores 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 <b>PCI Local Bus Specification, version 2.2</b> 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.
15	DISABLE_LAT_TMR (1)	1	Disable master latency timer. This bit controls whether the latency timer circuitry will operate as indicated in the <b>PCI Local Bus Specification, version 2.2</b> . 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.

Table 2. Bit Definition of the ENABLE\_BITS Parameter (Part 4 of 4)

Bit Number	Bit Name	Default Value	Definition
16	PCI_64BIT_SYSTEM	0	<p>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:</p> <ul style="list-style-type: none"> <li>■ Support 64-bit single-cycle master write transactions</li> <li>■ Assert <code>irdyn</code> one clock cycle after the assertion of <code>framen</code> for read and write transactions.</li> </ul> <p>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.</p>
31..17	Reserved	0	Reserved.

**Note:**

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





*Notes:*



*Notes:*