



**PCI-X MegaCore Function  
User Guide**

**Version 1.0  
August 2000**

ACCESS, Altera, AMPP, APEX, APEX 20K, Atlas, FLEX, FLEX 10K, FLEX 10KA, FLEX 10KE, FLEX 6000, FLEX 6000A, MAX, MAX+PLUS, MAX+PLUS II, MegaCore, MultiCore, MultiVolt, NativeLink, OpenCore, Quartus, System-on-a-Programmable-Chip, and specific device designations are trademarks and/or service marks of Altera Corporation in the United States and other countries. Product design elements and mnemonics used by Altera Corporation are protected by copyright and/or trademark laws.

Altera Corporation acknowledges the trademarks of other organizations for their respective products or services mentioned in this document, including the following: Verilog is a registered trademark of Cadence Design Systems, Incorporated. Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

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

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

1. Life support devices or systems are devices or systems that (a) are intended for surgical implant into the body or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

Products mentioned in this document are covered by one or more of the following U.S. patents: 5,873,113; 5,872,463; 5,870,410; 5,861,760; 5,859,544; 5,850,365; 5,850,152; 5,850,151; 5,848,005; 5,847,617; 5,845,385; 5,844,854; RE35,977; 5,838,628; 5,838,584; 5,835,998; 5,834,849; 5,828,229; 5,825,197; 5,821,787; 5,821,773; 5,821,771; 5,815,726; 5,815,024; 5,815,003; 5,812,479; 5,812,450; 5,809,281; 5,809,034; 5,805,516; 5,802,540; 5,801,541; 5,796,267; 5,793,246; 5,790,469; 5,787,009; 5,771,264; 5,768,562; 5,768,372; 5,767,734; 5,764,583; 5,764,569; 5,764,080; 5,764,079; 5,761,099; 5,760,624; 5,757,207; 5,757,070; 5,744,991; 5,744,383; 5,740,110; 5,732,020; 5,729,495; 5,717,901; 5,705,939; 5,699,020; 5,699,312; 5,696,455; 5,693,540; 5,694,058; 5,691,653; 5,689,195; 5,668,771; 5,680,061; 5,672,985; 5,670,895; 5,659,717; 5,650,734; 5,649,163; 5,642,262; 5,642,082; 5,633,830; 5,631,576; 5,621,312; 5,614,840; 5,612,642; 5,608,337; 5,606,276; 5,606,266; 5,604,453; 5,598,109; 5,598,108; 5,592,106; 5,592,102; 5,590,305; 5,583,749; 5,581,501; 5,574,893; 5,572,717; 5,572,148; 5,572,067; 5,570,040; 5,567,177; 5,565,793; 5,563,592; 5,561,757; 5,557,217; 5,555,214; 5,550,842; 5,550,782; 5,548,552; 5,548,228; 5,543,732; 5,543,730; 5,541,530; 5,537,295; 5,537,057; 5,525,917; 5,525,827; 5,523,706; 5,523,247; 5,517,186; 5,498,975; 5,495,182; 5,493,526; 5,493,519; 5,490,266; 5,488,586; 5,487,143; 5,486,775; 5,485,103; 5,485,102; 5,483,178; 5,477,474; 5,473,266; 5,463,328; 5,444,394; 5,438,295; 5,436,575; 5,436,574; 5,434,514; 5,432,467; 5,414,312; 5,399,922; 5,384,499; 5,376,844; 5,375,086; 5,371,422; 5,369,314; 5,359,243; 5,359,242; 5,353,248; 5,352,940; 5,309,046; 5,350,954; 5,349,255; 5,341,308; 5,341,048; 5,341,044; 5,329,487; 5,317,212; 5,317,210; 5,315,172; 5,301,416; 5,294,975; 5,285,153; 5,280,203; 5,274,581; 5,272,368; 5,268,598; 5,266,037; 5,260,611; 5,260,610; 5,258,668; 5,247,478; 5,247,477; 5,243,233; 5,241,224; 5,237,219; 5,220,533; 5,220,214; 5,200,920; 5,187,392; 5,166,604; 5,162,680; 5,144,167; 5,138,576; 5,128,565; 5,121,006; 5,111,423; 5,097,208; 5,091,661; 5,066,873; 5,045,772; 4,969,121; 4,930,107; 4,930,098; 4,930,097; 4,912,342; 4,903,223; 4,899,070; 4,899,067; 4,871,930; 4,864,161; 4,831,573; 4,785,423; 4,774,421; 4,713,792; 4,677,318; 4,617,479; 4,609,986; 4,020,469 and certain foreign patents.

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

Copyright © 2000 Altera Corporation. All rights reserved.



Printed on Recycled Paper.



I.S. EN ISO 9001



## About this User Guide

How to Contact Altera .....	v
Typographic Conventions .....	vi

## Getting Started

Before You Begin.....	9
Quartus Walk- Through.....	11
Using Third-Party EDA Tools.....	15

## MegaCore Overview

Features .....	21
General Description.....	22
Terminology & Abbreviations.....	23
PCI-2.2/PCI-X Bus Interface Signals.....	24

## Specifications

PCI Bus Commands.....	37
Target Mode Operation.....	38
Master Mode Operation.....	92
Split Transactions.....	133
Decode & Configuration.....	141



*Notes:*



## About this User Guide

This user guide provides comprehensive information about the Altera® `pci_x` MegaCore function.



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

### How to Contact Altera

For additional information about Altera products, consult the sources shown in [Table 1](#).






<i>Table 1 .How to Contact Altera</i>			
Information Type	Access	USA & Canada	All Other Locations
Altera Literature Services	Telephone hotline	(1)	(888) 3-ALTERA (1)
	Electronic mail	<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	Telephone hotline	(800) SOS-EPLD	(408) 544-7000
	Fax	(408) 544-7606	(408) 544-7606
Technical support	Telephone hotline (6:00 a.m. to 6:00 p.m. Pacific Time)	(800) 800-EPLD	(408) 544-7000 (1)
	Fax	(408) 544-6401	(408) 544-6401 (1)
	Electronic mail	<a href="mailto:sos@altera.com">sos@altera.com</a>	<a href="mailto:sos@altera.com">sos@altera.com</a>
	FTP site	<a href="ftp.altera.com">ftp.altera.com</a>	<a href="ftp.altera.com">ftp.altera.com</a>
General product information	Telephone	(408) 544-7104	(408) 544-7104 (1)
	World-wide web site	<a href="http://www.altera.com">http://www.altera.com</a>	<a href="http://www.altera.com">http://www.altera.com</a>

**Note:**

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

## Typographic Conventions

The *pci\_x MegaCore Function User Guide* uses the typographic conventions shown in [Table 2](#).

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



# Getting Started

## Contents

August 2000

Before You Begin.....	9
Installing the MegaCore Files.....	9
MegaCore Directory Structure.....	10
Quartus Walk- Through.....	11
Design Entry .....	12
Compilation & Functional Simulation.....	13
Timing Analysis .....	14
Configuring a Device .....	14
Using Third-Party EDA Tools.....	15
Generating VHDL & Verilog HDL Functional Models from the Quartus Software....	15
Synthesis Compilation & Post-Routing Simulation with the Quartus Software.....	16



*Notes:*





The Altera® `pci_x` MegaCore™ function provides solutions for integrating 64-bit PCI-X peripheral devices, including network adapters, graphic accelerator boards, and embedded control modules. The functions are optimized for Altera devices, greatly enhancing your productivity by allowing you to focus efforts on the custom logic surrounding the PCI-X interface.

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

## Before You Begin

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

### Installing the MegaCore Files

Depending on your platform, use the following instructions:

#### *Windows 95/98 & Windows NT 4.0*

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

1. Click **Run** (Start menu).
2. Type `<path name>\<filename>.exe`, where `<path name>` is the location of the downloaded MegaCore function and `<filename>` is the filename of the function.
3. Click **OK**. The **MegaCore Installer** dialog box appears. Follow the on-line instructions to finish installation.

### UNIX

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

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

### MegaCore Directory Structure

Altera `pci_x` MegaCore function files are organized into several directories; the top-level directory is `\megacore` (see [Table 1](#)).



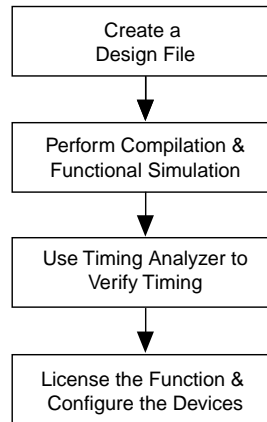
Altera updates MegaCore files from time-to-time. Therefore, Altera recommends that you do not save your project-specific files in the MegaCore directory structure.

<i>Table 1 .PCI MegaCore Directories</i>	
Directory	Description
<code>\lib</code>	Contains encrypted lower-level design files and other supporting files, i.e., symbol, an AHDL include, and Verilog simulation wrapper files. After installing the MegaCore function, you should set a user library in the Altera Quartus software that points to this directory. This library allows you to access all of the necessary MegaCore files.
<code>\&lt;pci_x&gt;\doc</code>	Contains supporting documents, i.e., <b>readme</b> file, walk-through file, and configuration design documents for the MegaCore function.
<code>\&lt;pci_x&gt;\examples\pcix_top</code>	Contains a Quartus block design file ( <b>.bdf</b> ), Quartus constraint files ( <b>.csf</b> , <b>.esf</b> , <b>.psf</b> ) and the decode/configuration module.
<code>\&lt;pci_x&gt;\examples\sim</code>	Contains a sample Quartus simulation file ( <b>.vwf</b> ).

## Quartus Walk-Through

This section describes an entire design flow using an Altera `pci_x` MegaCore function and the Quartus development system (see [Figure 1](#)).

*Figure 1. Example pci\_x Design Flow with the Quartus Software*



The following instructions assume that:

- You are using the `pci_x` MegaCore function.
- All files are located in the default directory, `c:\megacore`. If the files are installed in a different directory on your system, substitute the appropriate path name.
- You are using a PC; UNIX users should alter the steps as appropriate.
- You are familiar with the Quartus software.
- Quartus version 2000.05 or higher is installed in the default location (Refer to the **readme** file in `/doc` directory for the latest software support).
- You are using the OpenCore feature to test-drive the function or you have licensed the function.



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

The sample design process uses the following steps:

1. Create a BDF that instantiates the `pci_x` MegaCore function.
2. Modify user-specific configuration space registers in the design file `pcix_config.v`.
3. Perform a compilation and run functional simulations to evaluate and verify the functionality.
4. Examine the timing analysis results to verify that the PCI(X) timing specifications are met.
5. If you have licensed the `pci_x` function, configure a targeted Altera EP20K400EFC672-1X device with the completed design.

### Design Entry

The following steps explain how to create a BDF that instantiates an Altera PCI MegaCore function.



Refer to Quartus Help for detailed instructions on creating and editing block diagrams.

1. Copy the design file, `pcix_config.v` in the `\lib` directory into a new directory, e.g., `c:\altr_app`.
2. Modify user-specific configuration space registers in the design file, `pcix_config.v`.



In order to meet PCI-X 66-MHz timing, you may need to synthesize your modified `pcix_config.v` file in a third-party synthesis tool and generate an EDIF output file. An example of an EDIF file, `pcix_config.edf`, generated from Exemplar Leonardo Spectrum is provided in the `\pci_x\examples\pci_top` directory.

3. Run the Quartus software.
4. Create a new BDF named `pci_top.bdf` using the schematic shown in the `\pci_x\examples\pci_top` directory as an example. The block symbol file, `pci_x.bsf`, is located in the `\lib` directory. You may skip this step by saving `\pci_x\examples\pci_top\pci_top.bdf` as a new design and going to Step 5.

5. Using the Quartus software, save your BDF into a your working directory (e.g., `c:\altr_app`). You will be prompted to create a new project with this file. Choose **Yes** to create a new project.
6. The Quartus **New Project** wizard will open and select the present working directory and your new BDF as the project name and top-level design entity. If necessary, change any of the default settings in this dialog box and choose **Next**.
7. Specify the user library, `\lib` directory, for the `pci_x` MegaCore function using the User Library Pathnames feature. Add additional design files for your project as necessary and choose **Finish**.

After you have entered your design, you are ready to perform compilation to synthesize, and place and route your design.

## Compilation & Functional Simulation

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

1. Open your project in the Quartus software and choose the **Compile Mode** command (Processing menu).
2. Choose **Start Compile** (Processing menu) to compile your design. (This step performs a full compilation, including place and route, and timing analysis.)
3. When compilation completes, change to **Simulate Mode** (Processing menu) to functionally simulate your design.
4. Create a vector waveform file (`.vwf`) describing a PCI-X transaction. Save the file in your working directory. An example vector file, `cfg_wr_rd.vwf`, is provided in the `\pci_x\examples\sim` directory. This waveform simulates configuration write and read.
5. In the **Quartus Simulator Settings** dialog box (Processing menu), choose the **Mode** tab and select **Functional**. Click **Apply**.
6. Choose the **Time/Vectors** tab and specify the `.vwf` that you just created as the source of vector stimuli and choose **Apply**.
7. Choose **Run Simulation** (Processing menu) to simulate your design and view the simulation results.

After you verify that your design is functionally correct, you can use the Quartus timing analysis tool results to verify that all of the PCI-X signals in your design meet the PCI-X timing requirements.

### Timing Analysis

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

1. Choose the **Compile Mode** command (Processing menu). Example constraint files (**.csf**, **.esf**, **.psf**) for the project in `\pci_x\examples\pcix_top` are provided to meet PCI-X 66-MHz timing requirements.
2. Open the **Compilation Report** (Processing menu) and expand the **Timing Analysis** section.
3. The Quartus software lets you perform the following five types of timing analysis to verify your design:
  - **f<sub>MAX</sub>**: The **f<sub>MAX</sub>** results report the maximum clock frequency and identify the longest delay paths between registers.
  - **t<sub>SU</sub>**: The **t<sub>SU</sub>** results report the setup times of the registers.
  - **t<sub>H</sub>**: The **t<sub>H</sub>** results report the hold times of the registers.
  - **t<sub>CO</sub>**: The **t<sub>CO</sub>** results report the clock-to-output delays of the registers.
  - **t<sub>PD</sub>**: The **t<sub>PD</sub>** results report the combinatorial pin-to-pin delays.

You are now ready to configure your targeted Altera EP20K400EFC672-1X device.

### Configuring a Device

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

- The Altera Stand-Alone Programmer (ASAP2) includes an LP6 Logic Programmer card and a Master Programming Unit (MPU). You should use the PLMJ1213 programming adapter with the MPU to program a serial configuration device, which loads the configuration data to the APEX device during power-up. A Programmer Object File (.pof) is used to program the configuration device. The Altera Stand-Alone Programmer is typically used in the production stage of the design flow.

- The MasterBlaster™ communications cable is a standard PC serial or USB port hardware interface. An SRAM Object File (.sof) is used to configure the APEX device. The MasterBlaster cable is typically used in the prototyping stage of the design flow.
- The ByteBlasterMV™ parallel port download cable provides a hardware interface to a standard parallel port. The SOF is used to configure the APEX device. The ByteBlasterMV cable is typically used in the prototyping stage.



For more information, refer to the [ByteBlasterMV Parallel Port Download Cable Data Sheet](#) and [MasterBlaster Serial/USB Communications Cable Data Sheet](#).

## Using Third-Party EDA Tools

This section describe how to generate a VHDL or Verilog HDL functional model, and describe the design flow to compile and simulate your custom Altera `pci_x` MegaCore design with a third-party EDA tool.

### Generating VHDL & Verilog HDL Functional Models from the Quartus Software

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

1. Open the Quartus project containing your **pci\_top.bdf** file.
2. Choose the third-party EDA tool that you will use for simulation through the **EDA Tool Settings** dialog box (Project menu).
3. After selecting a simulation tool, you may choose to change the default settings by choosing the **Settings** tab.
4. After a successful compilation, Quartus will generate either a **pci\_top.vo** functional Verilog HDL model or **pci\_top.vho** functional VHDL model of your design. Quartus will also generate a **pci\_top.vo** or **pci\_top\_vhd.sdo** file containing the timing information.
5. Compile the **pci\_top.vo** or **pci\_top.vho** output files in your third-party simulator to perform functional simulation using Verilog HDL or VHDL.

To use the Quartus NativeLink feature to automatically start your simulation environment, review Quartus Help and the Quartus NativeLink Guidelines on simulating Verilog HDL and VHDL output files for the EDA tool of your choice.

## Synthesis Compilation & Post-Routing Simulation with the Quartus Software

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

1. Create your custom design instantiating the `pci_x` MegaCore function.
2. Synthesize the design using your third-party EDA tool. Your EDA tool should treat the `pci_x` MegaCore instantiation as a black box by either setting attributes or ignoring the instantiation.



For more information on setting compiler options in your third-party EDA tool, refer to the [Quartus NativeLink Guidelines](#).

3. After compilation, generate a hierarchical EDIF netlist file targeting the APEX device family in your third-party EDA tool.
4. Create a new project in Quartus from your EDIF file using the **New Project** wizard. Add your design file that contains the custom instantiation of the `pci_x` MegaCore function to the current project. Add the `pci_x\lib` directory to your User Libraries for the project.
5. Choose **EDA Tool Settings** (Project menu).
6. In the **EDA Tool Settings** dialog box, select the EDA tool for your EDIF netlist from the **Design Entry/Synthesis Tool** drop-down list box. Change the default tool setting through the **Settings** box as necessary.
7. In the **EDA Tool Settings** dialog box, select the EDA tool for your simulation from the **Simulation Tool** drop-down list box. Change the default tool settings through the **Settings** box as necessary.
8. Make logic option and/or place-and-route assignments for your custom logic using the **Assignment Organizer** (Tools menu).
9. Compile your design. The Quartus Compiler synthesizes and performs place-and-route on your design, and generates output and programming files.
10. Import your Quartus-generated output files (`.edo`, `.vho`, `.vo`, or `.sdo`) into your third-party EDA tool for post-route, device-level, and system-level simulation.



To use the Quartus NativeLink feature to automatically start your EDA tools for synthesis and simulation, review Quartus Help and the Quartus NativeLink Guidelines to setup your project for the EDA tools of your choice.



*Notes:*



# MegaCore Overview

## Contents

August 2000

Features .....	21
PCI-X/PCI-2.2 Supported Features.....	21
General Description.....	22
Terminology & Abbreviations.....	23
PCI-2.2/PCI-X Bus Interface Signals.....	24
Local-Side Target Interface Signals.....	26
Local-Side Master Interface Signals .....	29

2

MegaCore  
Overview



*Notes:*

## Features

This section describes the features of the `pci_x` MegaCore™ function. The `pci_x` function is a parameterized MegaCore function implementing a 64-bit PCI-X/PCI-2.2 master/target interface module that has been designed and optimized for Altera high density and high performance devices. The `pci_x` MegaCore function is fully compliant with the functional and timing requirements of the PCI Special Interest Group (PCI SIG) *PCI Local Bus Specification, Revision 2.2* and *PCI-X Addendum, Revision 1.0*.

The `pci_x` function isolates the PCI(X) bus interface state machines (and logic) from the buffer management state machines (and logic). The function isolates the internal modules as much as possible from the differences in signaling between the PCI-X and PCI-2.2 bus protocols thus making local bus behavior similar regardless of whether the external bus is operating with PCI-X or PCI-2.2 protocol.

### PCI-X/PCI-2.2 Supported Features

The `pci_x` function is based on the Compaq™ `pci_x` function and supports the following features:

- Flexible 64-bit master/target interface that can be customized for specific peripheral requirements
- Based upon industry standard PCI-X core from Compaq Computer—implemented in several application specific integrated circuits (ASICs)
- Fully compliant with the timing and functional requirements of the PCI Special Interest Group's (PCI SIG) ***PCI Local Bus Specification, Revision 2.2*** and ***PCI-X Addendum, Revision 1.0***
- PCI-X bus operation up to 66 MHz
- Achieves up to 33% higher system speed than PCI-2.2 at the same frequency
- Supports all PCI-2.2/PCI-X commands
- Dynamically handles 64-bit and 32-bit data operation
- Automatic byte enable generation in PCI-X protocol
- User-configurable configuration space
- Fully synchronous design
- No-risk OpenCore feature allows designers to instantiate and simulate designs in the Quartus software prior to purchase

## General Description

PCI-X is an evolutionary design and enhancement of the industry standard PCI bus. PCI-X enables 64-bit designs to operate at speeds up to 133MHz. PCI-X achieves this performance by implementing a register-to-register protocol and improved transaction processing. Called Split Transactions, this feature enables multi-threading split transactions rather than delayed serial processing inherent in PCI-2.2 protocol. PCI-X allows flexible protocol usage, i.e., it can be used with both legacy PCI-2.2 systems as well as newer PCI-X systems.

The Altera® MegaCore function (ordering code: PLSM-PCI/X) is a high performance, flexible implementation of the 64-bit PCI-X master/target interface. Because this function handles the complex PCI-X protocol and stringent timing requirements internally, designers can focus their engineering efforts on value-added custom development, significantly shortening design cycles and thus reducing time-to-market.

In addition, the Altera `pci_x` function is based upon the industry standard PCI-X core from Compaq Computer. This core has been rigorously tested by Compaq and its partners to ensure PCI-X 1.0 and PCI-2.2 compliance. The Compaq core also has been implemented in several ASICs to further demonstrate compliance and stability. All protocol and timing requirements of PCI-X and PCI-2.2 are supported by the core. The high-speed PCI-X interface is targeted to a broad range of applications such as servers and networks that require high-bandwidth protocols such as Gigabit Ethernet and Fibre Channel.

Optimized for Altera high-density APEX devices, designers have ample resources for custom local logic after implementing the PCI-X interface. The high performance of APEX devices also enables the `pci_x` function to support unlimited cycles of zero-wait-state memory-burst transactions.

Additionally, the configuration space is user-configurable and parameterized, providing scalability, adaptability, and efficient silicon implementation. As a result, the same `pci_x` function can be used in multiple PCI-X projects with different requirements. For example, the `pci_x` function can offer up to six base address registers (BARs) for multiple local-side devices. However, some applications require only one contiguous memory range. PCI-X designers can choose to instantiate only one BAR, which reduces logic cell consumption. After designers define the parameter values, the Quartus software automatically and efficiently modifies the design and implements the logic.

This user guide should be used in conjunction with the latest PCI-2.2/PCI-X specifications, published by the PCI Special Interest Group (SIG). Users should be fairly familiar with the PCI-2.2/PCI-X standards when using this function.

## Terminology & Abbreviations

**Table 1** lists common PCI terms and abbreviations used throughout this user guide.

<i>Table 1. Commonly Used Terms &amp; Abbreviations</i>	
Terms	Description
ADB	Allowable Disconnect Boundary. A PCI-X term used to describe a 128-byte, address-aligned boundary
Completer	The PCI-X device that issues a split response for a transaction and is responsible for completing the transaction using a split completion.
DAC	Dual address cycle. In 64-bit addressing, the lower address is written to on the first clock cycle and the upper address space is written to on the second clock cycle.
DWORD	PCI-X DWORD-type transaction.
Dword	Address-aligned four bytes.
Local bus	Refers to the internal bus to which the <code>pci_x</code> module is designed.
Local master	The <code>pci_x</code> initiates local to PCI-2.2/PCI-X cycles.
Local master bus	Internal bus for transactions where the <code>pci_x</code> wishes to initiate PCI-2.2/PCI-X cycles.
Local target	The <code>pci_x</code> is the target of PCI-2.2/PCI-X to local interface cycles.
Local target bus	Internal bus for transactions where the <code>pci_x</code> is the target of PCI-2.2/PCI-X cycles.
MegaCore	Refers to the PCI(X) to local interface module. Also referred to as <code>pci_x</code> function.
MLT	Master Latency Timer. See PCI-2.2/PCI-X Bus Specification.
MR	PCI memory read command.
MRB	PCI-X memory read block cycle.
MRD	PCI-X memory read DWORD cycle.
MRL	PCI-2.2 memory read line command.
MRM	PCI-2.2 memory read multiple command.
MWB	PCI-X memory write block cycle.
PCI-2.2	Refers to a PCI-2.2 cycle only and not a PCI-X cycle.
PCI(X)	Refers to both a PCI-2.2 cycle and a PCI-X cycle.
PCI-X	Refers to a PCI-X cycle only and not a PCI-2.2 cycle.
Qword	Address-aligned eight bytes.
RDH	PCI-X reserved drive high (RDH) byte enable usage.
Requester	The PCI-X device whose request has received a split response.
SC	PCI-X split completion cycle.
SCM	PCI-X split completion message cycle.
Speculative / non-speculative	Memory regions can be speculative and non-speculative. A speculative memory region is one in which read side affects do not occur (prefetching is allowed). A non-speculative memory region is one in which read side affects do occur (prefetching is not allowed).
Split completion	Data cycle run by the completer with the originally requested data or completion status.
Split response	Response given by completer to requester indicating it wishes to split the request.

## PCI-2.2/PCI-X Bus Interface Signals

The following PCI-2.2/PCI-X signals are used by the `pci_x` function:

- *Input*—Standard input-only signal
- *Output*—Standard output-only signal
- *Bidirectional*—Tri-state input/output signal
- *Sustained tri-state (STS)*—Signal that is driven by one agent at a time (e.g., device or host operating on the PCI bus). An agent that drives a sustained tri-state pin low must actively drive it high for one clock cycle before tri-stating it. Another agent cannot drive a sustained tri-state signal any sooner than one clock cycle after it is released by the previous agent.
- *Open-drain*—Signal that is wire-ORed with other agents. The signaling agent asserts the open-drain signal, and a weak pull-up resistor deasserts the open-drain signal. The pull-up resistor may require two or three PCI-X/PCI-2.2 bus clock cycles to restore the open-drain signal to its inactive state.

**Table 2** lists the PCI-X/PCI-2.2 bus interface input and output signals, describing the address, data, command, and byte enables of the transaction. Unless otherwise noted, the signals in **Table 2** interface with either PCI-X or PCI-2.2 bus protocol.

Name	Type	Polarity	Description
<code>pci_clk</code>	Input	—	Clock. The clock provides the reference signal for all other PCI-X/PCI-2.2 interface signals, except <code>rst_n</code> and <code>inta_n</code> .
<code>pci_gnt_n</code>	Input	Low	Grant. The <code>gnt_n</code> input indicates to the bus master device that it has control of the PCI-X/PCI-2.2 bus. Every master device has a pair of arbitration lines ( <code>gnt_n</code> and <code>req_n</code> ) that connect directly to the arbiter.
<code>pci_idsel</code>	Input	High	Initialization device select. The <code>idsel</code> input is a device select for configuration transactions.
<code>pci_reset_n</code>	Input	—	Reset. The <code>rst_n</code> input initializes the PCI-X/PCI-2.2 interface circuitry, and can be asserted asynchronously to the PCI-X/PCI-2.2 bus clock edge. When active, the PCI-X/PCI-2.2 output signals are tri-stated and the open-drain signals, such as <code>serr_n</code> , float.
<code>pci_req_n</code>	Output	Low	Request. The <code>req_n</code> output indicates to the arbiter that the bus master wants to gain control of the PCI-X/PCI-2.2 bus to perform a transaction.
<code>pci_ad[63..0]</code>	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>irdy_n</code> and <code>trdy_n</code> are both asserted. In the case of a 32-bit data phase, only <code>ad[31..0]</code> will hold valid data.



Table 2 .PCI-X/PCI-2.2 Bus Interface Signals (Part 2 of 3)

Name	Type	Polarity	Description
pci_cbe_n[7..0]	Tri-State	Low	Command/byte enable. The cben[7..0] bus is a time-multiplexed command/byte enable bus. During the address phase, this bus indicates the command; during the data phase, this bus indicates byte enables.
pci_par	Tri-State	—	Parity. The par signal is even parity across the 32 least significant address/data bits and 4 least significant command/byte enable bits. In other words, the number of “1”s on ad[31..0], cben[3..0], and par equal an even number. The parity of a data phase is presented on the bus during the clock following the data phase.
pci_par64	Tri-State	—	Parity 64. The par64 signal is even parity across the 32 most significant address/data bits and the 4 most significant command/byte enable bits. In other words, the number of “1”s on ad[63..32], cben[7..4], and par64 equal an even number. The parity of a data phase is presented on the bus during the clock cycle following the data phase.
pci_ack64_n	STS	Low	Acknowledge 64-bit transfer. The target asserts ack64_n to indicate that the target can transfer data using 64 bits. The ack64_n signal has the same timing as devsel_n.
pci_devsel_n	STS	Low	Device select. Target asserts devsel_n to indicate that the target has decoded its own address and accepts the transaction.
pci_frame_n	STS	Low	Frame. The frame_n signal is an output from the current bus master and indicates the beginning and duration of a bus operation. When frame_n is initially asserted, the address and command signals are present on the ad[63..0] and cben[7..0] buses respectively. The frame_n signal remains asserted during the data operation and is deasserted to identify the end of a transaction.
pci_irdy_n	STS	Low	Master ready. The irdy_n 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, irdy_n indicates valid data. In a read transaction, irdy_n indicates that the master is ready to accept data.
pci_perr_n	STS	Low	Parity error. The perr_n signal indicates a data parity error. The perr_n signal is asserted one clock following the par and par64 signals, or two clocks following a data phase with a parity error. The pci_x asserts the perr_n signal if (1) parity error is detected on either the par or par64 signals, and (2) the perr_n bit (bit 6) of the command register is set.
pci_req64_n	STS	Low	Request 64-bit transfer. The req64_n signal is an output from the current bus master and indicates that the master is requesting a 64-bit transaction. The req64_n signal has the same timing as frame_n.

Name	Type	Polarity	Description
pci_stop_n	STS	Low	Stop. The stop_n signal is a target device request that indicates to the bus master to terminate the current transaction. The stop_n signal is used in conjunction with trdy_n and devsel_n to indicate the type of termination initiated by the target.
pci_trdy_n	STS	Low	Target ready. The trdy_n signal is a target output, indicating that the target can complete the current data transaction. In a read operation, trdyn indicates that the target is providing data. In a write operation, trdyn indicates that the target is ready to accept data.
pci_inta_n	Open-Drain	Low	Interrupt A. The inta_n signal is an active-low interrupt to the host and must be used for any single-function device requiring an interrupt capability.
pci_serr_n	Open-Drain	Low	System error. Indicates system error and address parity error. The pci_x asserts serr_n if a parity error is detected during an address phase and the serr_n enable bit (bit 8) of the command register is set.

## Local-Side Target Interface Signals

Tables 3 through 7 describe the pci\_x local-side target interface signals. Because there are two local address buses in the pci\_x, all local side target and master interface signals are separate. Unless otherwise noted, the following local side target signals interface with either the PCI-X or PCI-2.2 protocol.

Table 3 lists the local-side target signal naming conventions.

Signal prefix	Description
lto*	Local target output signal prefix
lti*	Local target input signal prefix

*Local Target Address, Data, Command, and Byte Enable Signals*

**Table 4** lists the local-side interface inputs and outputs, describing the address, data, command, and byte enables of the transaction.

Name	Type	Polarity	Description
lti_readdata[63..0]	Input	—	Target to master read data. Provides read data from the target to the master.
lto_addr[63..0]	Output	—	Master to target address.
lto_be_[7..0]	Output	—	Master to target byte enable. Indicates (to the local side) valid byte lanes for each data phase of the transfer.
lto_cmd[3..0]	Output	—	Master to target command. Indicates to the local side the PCI command of the current transaction. Command encoding is identical to PCI-2.2 protocol, except that the dual address cycle (DAC) command is reserved.
lto_writedata[63..0]	Output	—	Master to target write data transfer. Provides the data written from the master to the target device.

*Local Target Control Signals*

**Table 5** lists the local-side interface inputs and outputs, describing control logic information for the transaction.

Name	Type	Polarity	Description
lti_disc	Input	High	Target to master disconnect. Indicates that target is ending the cycle after data has transferred.
lti_rdy	Input	High	Target to master ready. Indicates that the target is either driving valid data (read cycle), or that the target can accept data (write cycle).
lti_retry	Input	High	Master to target 64-bit access. Indicates that the target can not process the current data cycle and that the cycle should be retried on the expansion bus. Also, lti_retry can be used to support delayed transactions.
lto_64access	Output	High	Master to target 64-bit access. Indicates that the transaction on the expansion bus was initiated as a 64-bit transaction. The lto_64access signal is necessary for the target to properly decode the cycle.

**Table 5. Local Target Control Signals (Part 2 of 2)**

Name	Type	Polarity	Description
lto_cyc	Output	High	Master to target cycle. Indicates valid address and command data is being driven. However, lto_cyc does not indicate either (1) that the transaction has been claimed on the expansion bus, or (2) for which target device the cycle is intended. Target devices may use lto_cyc to begin the decode process, but must wait for lto_cycvalid to confirm that it is the target of the cycle.
lto_cycvalid	Output	High	Master to target cycle valid. Indicates that the transaction will be claimed on the expansion bus and that the target device receiving the signal is the intended target of the cycle.
lto_writeburst	Output	High	Master to target burst. Valid only for (1) memory write, and (2) memory write and invalidate cycles. Indicates that the write transaction still has at least one more data phase remaining.

#### *Local Target Control Signals Interfacing Only with PCI-X Protocol*

**Table 6** lists additional local side interface inputs and outputs, describing control logic information for the transaction. These signals interface only with PCI-X protocol.

**Table 6. Local Target Control Signals Interfacing Only with PCI-X Protocol (Part 1 of 2)**

Name	Type	Polarity	Description
lti_discadb	Input	High	Target to master disconnect at ADB. Indicates that the target device will terminate data transfer at the next ADB.
lti_split	Input	High	Target to master split transaction request. Indicates the target intends to split the transaction.
lto_busno[7..0]	Output	—	Master to target bus number. Indicates the bus number of the originating device.
lto_deviceno[4..0]	Output	—	Master to target device number. Indicates the device number of the originating device.
lto_functionno[2..0]	Output	—	Master to target function number. Indicates the function number of the originating device.
lto_scexception	Output	High	Master to target split completion exception. Indicates the currently active split completion cycle is an exception.
lto_scmessage	Output	High	Master to target split completion message. Indicates the currently active split completion cycle is a message, i.e., does not carry data.
lto_tbc[11..0]	Output	—	Master to target transaction byte count. Indicates the number of bytes to be transferred. Also, for type 0 configuration write cycles, bits[7..0] carry the secondary number.

**Table 6. Local Target Control Signals Interfacing Only with PCI-X Protocol (Part 2 of 2)**

<code>lto_tbcmodified</code>	Output	High	Master to target transaction byte count modified. Indicates the byte count of the current transaction has been modified by the master device.
<code>lto_tagno[4..0]</code>	Output	—	Master to target tag number. Indicates the tag number of the current cycle. The <code>lto_tagno[4..0]</code> signal is not used for split completion cycles.

### Local Target Error Reporting Signals

**Table 7** lists the local side interface inputs and outputs, describing error-reporting information for the transaction.

**Table 7. Local Target Error Reporting Signals**

Name	Type	Polarity	Description
<code>lti_abort</code>	Input	High	Target to master abort. Indicates the target has encountered a fatal error, or can never process the current cycle, and that the cycle should be aborted on the expansion bus.
<code>lti_perr</code>	Input	High	Target to master parity error. Indicates that the target has detected a read data parity error and is forwarding the information to the master. The master device can use the forwarded information to drive incorrect parity on the expansion bus and force a parity error.
<code>lti_serr</code>	Input	High	Target to master system error. Instructs the master to assert the system error signal ( <code>serrn</code> ) on the expansion bus.
<code>lto_perr</code>	Output	High	Master to target parity error. Indicates that the master has either detected or observed a data parity error on its expansion bus and is forwarding the information to the target.
<code>lto_serr</code>	Output	High	Master to target system error. Indicates a system error has occurred on the expansion bus.

### Local-Side Master Interface Signals

Tables 9 through 13 describe the `pci_x` local-side master interface signals. Because there are two local address buses in the `pci_x`, all local side target and master interface signals are separate. Unless otherwise noted, the following local-side master signals interface with either PCI-X or PCI-2.2 protocols.

**Table 8** lists the local-side master signal naming conventions.

<i>Table 8. pci_x Local -Side Master Signal Naming Conventions</i>	
Signal prefix	Description
lmo*	Local master output signal prefix
lmi*	Local master input signal prefix

### *Local Master Arbitration Signals*

**Table 9** lists the local side interface inputs and outputs, describing the arbitration handshaking required to request ownership of the bus.

<i>Table 9. Local Master Arbitration Signals</i>			
Name	Type	Polarity	Description
lmi_cyc	Input	High	Master to target request. When asserted, lmi_cyc indicates that the master device wants to run a transaction on the expansion bus. However, if the master device does not intend to immediately run another transaction after the current one, the lmi_cyc signal must be deasserted on the clock following the assertion of lmo_ack.
lmi_cycvalid	Input	High	Master to target valid cycle information. Indicates that the address, command, and data phases are valid, and that other cycle information signals are valid. The lmi_cycvalid signal must deassert after the master samples lmo_ack.
lmo_ack	Output	High	Target to master acknowledgment. When the target device asserts lmo_ack, the target is indicating that the master device's address and cycle information have been captured and that the data phase has been entered. Master devices also step forward to the next data phase when they sample lmo_ack.

*Local Master Address, Data, Command & Byte Enable Signals*

**Table 10** lists the local-side interface inputs and outputs, describing the address, data, command, and byte enable signals of the transaction.

Name	Type	Polarity	Description
<code>lmi_addr[63..0]</code>	Input	—	Master to target address. Provides the address to be driven on the expansion bus. The address must have byte resolution for I/O cycles.
<code>lmi_be[7..0]</code>	Input	—	Master to target byte enable. Indicates which byte lanes are valid for the current data phase. Byte enables are valid for each data phase of a multiple data phase transfer.
<code>lmi_cmd[3..0]</code>	Input	—	Master to target command. Provides the command to be driven on the expansion bus. PCI-X command encoding is identical to PCI-2.2 encoding except that the DAC command is reserved.
<code>lmi_dphasecnt[9..0]</code>	Input	—	Master to target data phase count. Indicates the number of Qwords or data phases for a 64-bit bus that the master device wants to transfer to/from the expansion bus.
<code>lmi_writedata[63..0]</code>	Input	—	Master to target read data. Provides the target device with the write data from the master device.
<code>lmo_readdata[63..0]</code>	Output	—	Target to master read data. Provides the data that was read by the target to the master device. Read data is qualified by the <code>lmo_xferhi</code> and <code>lmo_xferlo</code> signals, so not all 64-bits of data may be valid.

*Local Master Control Signals*

**Table 11** lists the local side interface inputs and outputs, describing control logic information for the transaction.

Name	Type	Polarity	Description
<code>lmi_disc</code>	Input	High	Master to target disconnect. Indicates the master wishes to end the cycle, overriding the original <code>lmi_dphasecnt</code> cycle negotiated.
<code>lmo_cycdone</code>	Output	High	Target to master cycle done. Indicates the cycle on the expansion bus is complete.
<code>lmo_retry</code>	Output	High	Target to master retry. Indicates that the expansion bus has retried or disconnected the current cycle. If either or both of the <code>lmo_xfer</code> signals are asserted with this signal, data transfer occurs with the termination.

Name	Type	Polarity	Description
lmo_xferhi	Output	High	Target to master high doubleword (4 bytes) of data transfer. Indicates the data is being transferred on the current clock to or from the master. On master writes, the assertion of this signal indicates that the expansion bus target has transferred a higher doubleword of data (lmi_writedata[63..32]). On master reads, the assertion of this signal indicates that valid read data is being driven on lmo_readdata[63..32].
lmo_xferlo	Output	High	Target to master low doubleword (4 bytes) data transfer. Indicates that the data is being transferred on the current clock to or from the master device. On master writes, the assertion of this signal indicates that the expansion bus target has transferred a lower doubleword of data (lmi_writedata[31..0]). On master reads, the assertion of this signal indicates that valid read data is being driven on lmo_readdata[31..0].

*Local Master Control Signals Interfacing Only with PCI-X Protocol*

**Table 12** lists additional local side interface inputs and outputs, describing control logic information for the transaction. These signals interface with PCI-X only.

Name	Type	Polarity	Description
lmi_busno[7..0]	Input	—	Master to target bus number. Indicates the bus number of the originating bus.
lmi_deviceno[4..0]	Input	—	Master to target device number. Indicates the device number of the originating bus.
lmi_discadb	Input	High	Master to target disconnect at ADB. Indicates master is overriding the data phase count and wishes to terminate data transfer at the next ADB.
lmi_functionno[2..0]	Input	—	Master to target function number. Indicates the function number of the originating device.
lmi_scexception	Input	High	Indicates the current transaction is a split completion exception.
lmi_scmmessage	Input	High	Indicates the current transaction is a split completion message.
lmi_tbc[11..0]	Input	—	Master to target byte count. Indicates the number of bytes to be transferred for this transaction.



**Table 1 2 .Local Master Control Signals Interfacing Only with PCI-X Protocol (Part 2 of 2)**

<code>lmi_tbcmodified</code>	Input	High	Master to target transaction byte count modified. Indicates the master has modified the byte count of the current transaction.
<code>lmi_tagno[4..0]</code>	Input	High	Master to target tag number. Indicates the tag number of the current cycle.
<code>lmo_split</code>	Output	High	Target to master split response. Indicates the target on the expansion bus drove a split response for this cycle.

### Local Master Error Reporting Signals

**Table 13** lists the local side interface inputs and outputs, describing error-reporting information for the transaction.

**Table 1 3 .Local Master Error Reporting Signals**

Name	Type	Polarity	Description
<code>lmi_perr</code>	Input	High	Master to target parity error. Indicates the current data driven on <code>lmi_writedata</code> encountered an ECC/parity error or other corruption on the originating bus. Valid only when <code>lmi_writedata</code> is driven on a write cycle. The <code>lmi_perr</code> signal is driven concurrently with the data that contains the ECC/parity error or corruption information, and must follow <code>lmi_writedata</code> wait-state rules.
<code>lmi_serr</code>	Input	High	Master to target system error. Instructs the expansion bus target to assert the system error signal ( <code>serrn</code> ) on the expansion bus.
<code>lmo_masterabort</code>	Output	High	Target to master, master abort signal. Indicates that a target did not respond to the cycle on the target expansion bus.
<code>lmo_perr</code>	Output	High	Target to master parity error. Driven two clocks after the affected <code>lmo_xfer</code> , the <code>lmo_perr</code> signal indicates that the data driven or received on the expansion bus encountered a parity error.
<code>lmo_targetabort</code>	Output	High	Target to master on target abort signal. Indicates that a target on the target expansion bus issued a target-abort for the cycle.

## MegaCore Overview

---



*Notes:*



# Specifications

## Contents

August 2000

PCI Bus Commands.....	37
Target Mode Operation.....	38
Target Read Transactions .....	38
Target Write Transactions .....	64
Master Mode Operation.....	92
Addressing.....	92
Master Read Transactions.....	94
Master Write Transactions.....	115
Split Transactions.....	133
Master Device Receives Split Response.....	133
Target Device Issues Split Response .....	135
Master Issues Split Completion .....	137
Target Device Receives Split Completion .....	139
Decode & Configuration.....	141
Design Files.....	142
Signal Descriptions .....	143
Functional Blocks .....	145
Design Considerations .....	146



*Notes:*

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

## PCI Bus Commands

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

*Table 1. PCI Bus Command Support Summary*

cben[3:0] Value	PCI-X Commands	PCI-2.2 Commands	Master	Target
0000	Interrupt acknowledge	Interrupt acknowledge	Yes	Yes
0001	Special cycle	Special cycle	Yes, Note (1)	Yes, Note (1)
0010	I/O read	I/O read	Yes	Yes
0011	I/O write	I/O write	Yes	Yes
0100	Reserved	Reserved	Ignored	Ignored
0101	Reserved	Reserved	Ignored	Ignored
0110	Memory read	Memory read	Yes	Yes
0111	Memory write	Memory write	Yes	Yes
1000	Alias to memory read block	Reserved	Yes, Note (2)	Yes, Note (2)
1001	Alias to memory write block	Reserved	Yes, Note (2)	Yes, Note (2)
1010	Configuration read	Configuration read	Yes	Yes
1011	Configuration write	Configuration write	Yes	Yes
1100	Split completion	Memory read multiple	Yes	Yes
1101	Dual address cycle (DAC)	Dual address cycle (DAC)	Yes	Yes
1110	Memory read block	Memory read line	Yes	Yes
1111	Memory write block	Memory write and invalidate	Yes	Yes

**Notes:**

- (1) During a special cycle, the `pci_x` function does not detect and drive `serr_n` on data parity error; however, the function does forward special cycle information to the target local side interface.
- (2) These commands are supported for PCI-X cycles, but ignored during PCI-2.2 cycles.

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

The `pci_x` responds to standard memory read/write, cache memory read/write, I/O read/write, and configuration read/write commands. The bus commands are discussed in greater detail in “Target Mode Operation” and “Master Mode Operation.”

## Target Mode Operation

This section describes all supported target transactions for the `pci_x` function and includes waveform diagrams showing typical PCI(X) cycles in target mode. As a target device, the `pci_x` supports all types of PCI-X and PCI-2.2 command types, except for special cycle commands. The `pci_x` does not detect and drive `serr_n` on data parity error during a special cycle. However, the `pci_x` does forward special cycle information to the local target bus.

The data width of the local target interface is 64-bit; however, the interface communicates with either 64-bit or 32-bit PCI-X master devices. Therefore, local target devices can perform all operations, except where PCI specifications require single data phases, with QWORD granularity.



The `pci_x` supports decode speed C for PCI-X cycles and slow decode speed for PCI-2.2 cycles.

### Target Read Transactions

The `pci_x` function performs both PCI-X and PCI-2.2 target read transactions. There are a few major differences between the PCI-X and PCI-2.2 target read transactions; however, from the perspective of the local target bus, behavior of the `pci_x` during the transactions is similar.

PCI-X transactions are different than PCI-2.2 transactions because they include an attribute phase, block cycle support, and split transaction support:

- The attribute phase follows the address phase and provides more transaction information up front, e.g., transaction byte count of the request.
- Block cycle (128-byte, address-aligned boundary) support is new with PCI-X protocol and enhances the robustness of transactions, i.e., more data can be transferred per clock cycle.
- Split transactions enable other peripheral devices to perform transactions with the PCI-X bus during an active transaction, i.e., with PCI-2.2 protocol no other bus transactions can occur until the current transaction is complete. See “Split Transactions” on page 133 for more information.

### PCI-X Target Read Transactions

Because the PCI-X bus can operate as a 64-bit or 32-bit bus and the local target bus is 64-bit, the `pci_x` performs some data and byte enable manipulation to compensate for the data-width mismatch.

The following are some general operating rules for PCI-X target read transactions:

- The PCI-2.2 delayed transaction protocol is not allowed with PCI-X protocol; thus, there is no issue of local targets responding with a local target retry and storing transaction information. Local targets that cannot provide immediate data *must* instead provide a split response (i.e., assert `lti_Split`), and then store the relevant transaction information.
- For 64-bit or 32-bit memory read block cycles where the local target has immediate data, the `pci_x` will accept read data from the local target bus in 64-bit increments. While the PCI-X master does not drive any byte enables for a memory read block transaction (i.e., the `pci_cbe_n[7:0]` bus is reserved drive high), the megacore will generate valid byte enables on the local target `lto_BE_n` bus lines on a per data phase basis.
- For 32-bit single data phase PCI-X cycles (e.g., DWORD memory, I/O, and configuration cycles) where the local target device has immediate data, the `pci_x` will pick up the data on either the upper or lower 32 bits of the 64-bit local data bus (i.e., `lti_ReadData[63:0]`). The value of bit 2 in the address determines whether the data is picked up in the upper or lower boundary of the data bus.
- For all immediate read cycles (i.e., non split transactions), parity error information (i.e., `lti_Perr`) for a particular data phase can be driven by the local target along with the assertion of `lti_Rdy`. At this point, the `pci_x` intentionally drives the data with bad parity on the PCI-X bus.
- The address presented to the local bus is the same as the PCI-X address. Therefore, when the `pci_x` is operating with PCI-X protocol, the address specified on `lto_Addr` is byte-aligned, except for configurations transactions, which are DWORD aligned.
- When a local target issues a split response during a PCI-X local read request, the local target is responsible for:
  - Storing the relevant transaction information
  - Subsequently issuing the split completion cycle on the local master bus.

However throughout the assertion of `lto_Cyc`, the `pci_x` will maintain a stable value for the transaction information, including address, command, device number, bus number, and tag number values.

In PCI-X protocol, the `pci_x` supports two types of 64-bit target 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-X master asserts `pci_frame_n` and `pci_req64n` signals and drives the address and command on `pci_ad[31:0]` and `pci_cben[3:0]`, correspondingly. Asserting the `pci_req64n` signal indicates to the target device that the master device is requesting a 64-bit data transaction.
2. One clock later is the attribute phase. The attribute phase contains additional information about the current transaction, such as the byte count of the transaction.
3. Turn-around cycles on the `pci_ad[63:0]` bus occur during the clock immediately following the attribute phase. During the turn-around cycles, the PCI-X master tri-states the `pci_ad[63:0]` bus. This process is necessary because the PCI-X agent driving the `pci_ad[63:0]` bus changes during read cycles.
4. If the address of the transaction matches one of the base address registers, the `pci_x` function turns on the drivers for the `pci_ad[63:0]`, `pci_devsel_n`, `pci_ack64_n`, `pci_trdy_n`, and `pci_stop_n` signals. The drivers for `pci_par` and `pci_par64` are turned on in the following clock.
5. The `pci_x` drives and asserts `pci_devseln` and `pci_ack64n` to indicate to the master device that it is accepting the 64-bit transaction.
6. One or more data phases follow next, depending on the type of read transaction.

### PCI-X 64-Bit Single-Cycle Target Memory Read Transaction

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



Figure 1. PCI-X 64-Bit Single-Cycle Target Memory Read Transaction

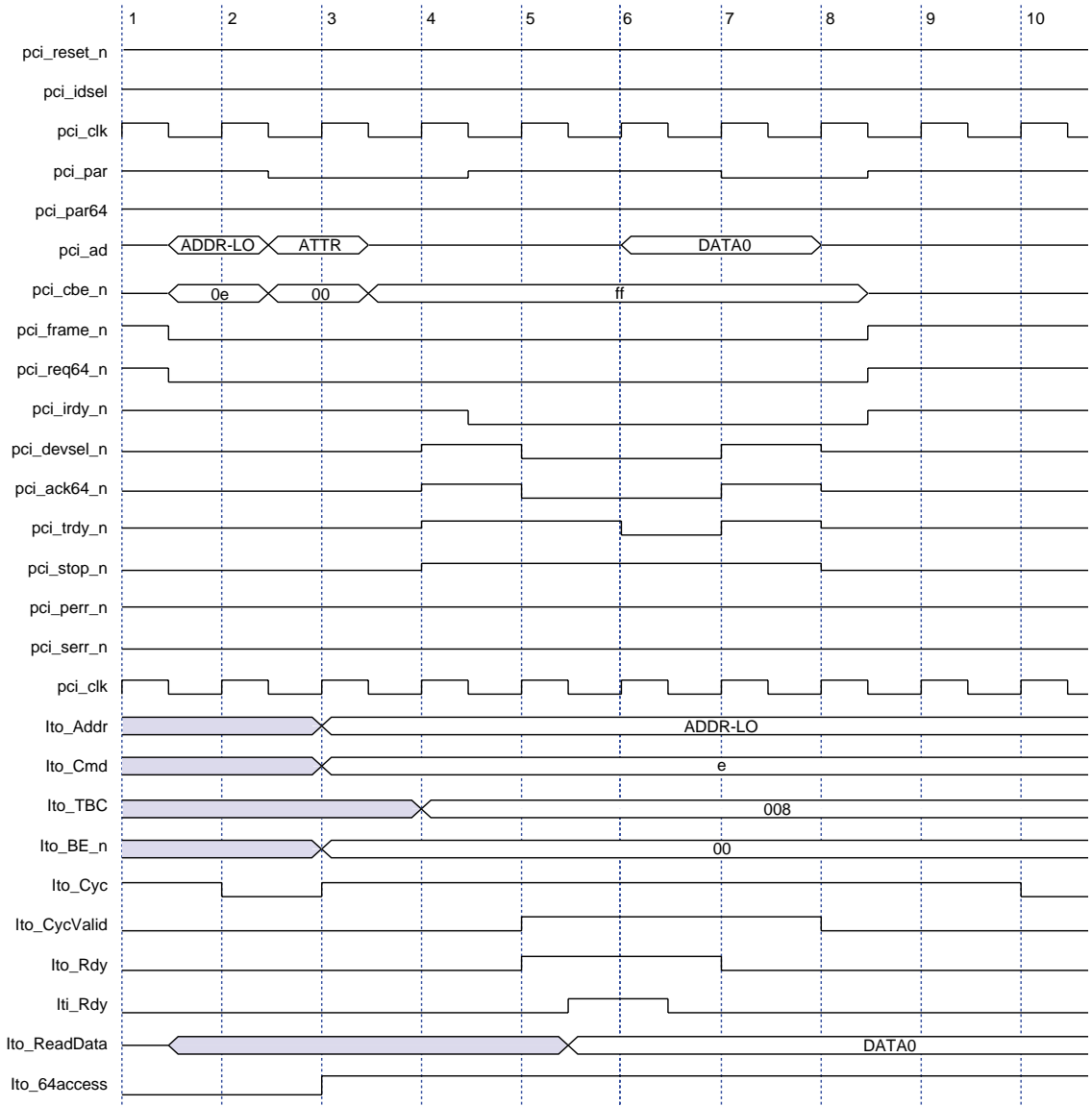


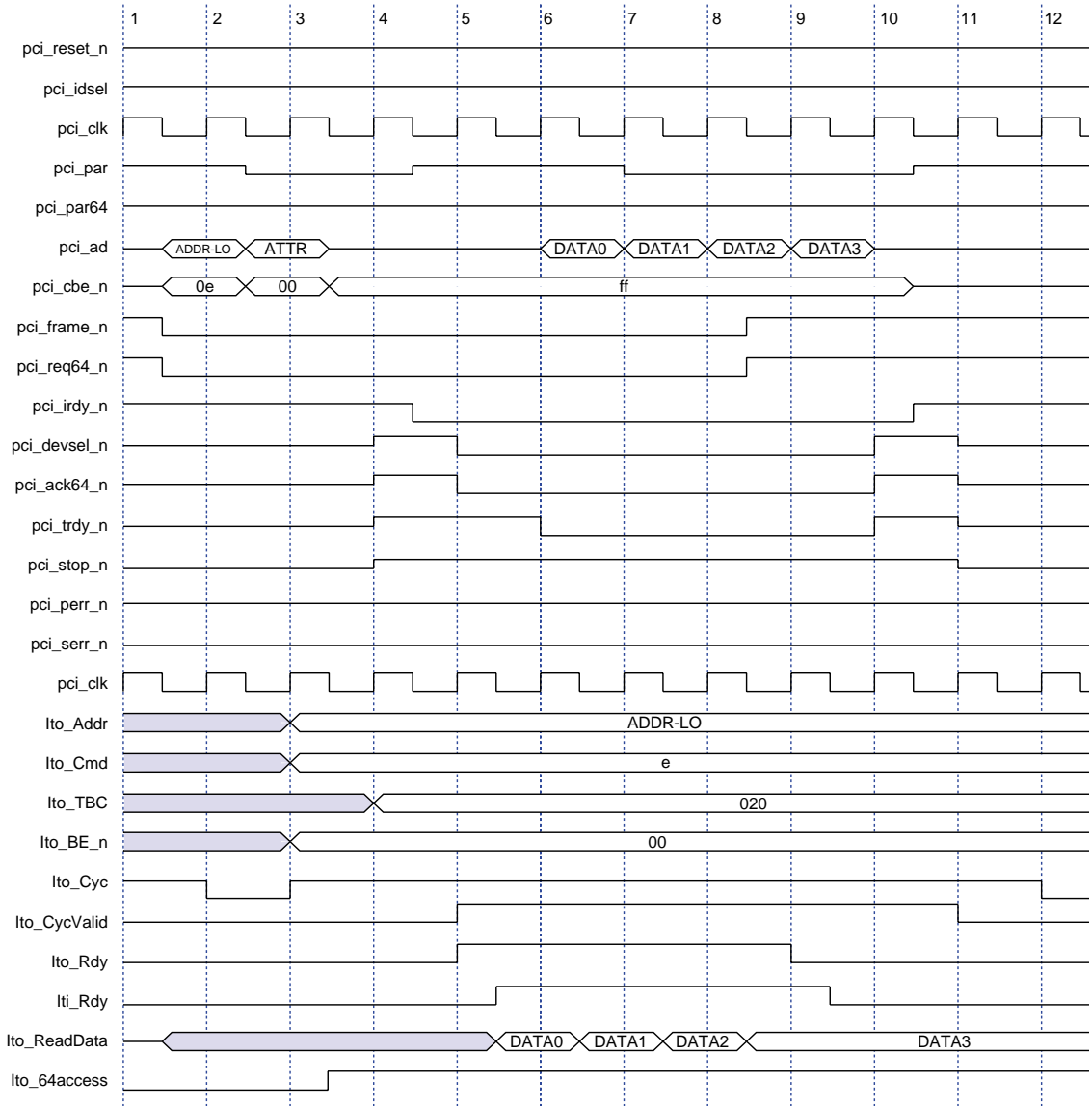
Table 2 shows the sequence of events for a PCI-X 64-bit single-cycle target memory read transaction.

Table 2. PCI-X 64-Bit Single-Cycle Target Read Transaction	
Clock Cycle	Event
1	PCI-X side: The address phase. A PCI-X master asserts <code>pci_frame_n</code> to indicate the beginning of a transaction. At the same time, <code>pci_req64_n</code> is asserted to indicate that a 64-bit transaction is being requested.
2	PCI-X side: The attribute phase. Additional information about the transaction is provided on <code>pci_ad[31:0]</code> and <code>pci_cben[3:0]</code> . In this transaction, the total byte count transferred is eight. Local side: The <code>pci_x</code> latches the address and command signals and decodes the address, verifying if the address falls within one of the BARs.
3	PCI-X side: Turn-around cycle on the <code>pci_ad[63:0]</code> bus. Local side: The <code>lto_Cyc</code> is asserted to indicate that a PCI-X transaction is occurring. The <code>pci_x</code> drives the transaction address on the <code>lto_Addr[31:0]</code> bus and the command address on the <code>lto_Cmd[3:0]</code> bus. The <code>pci_x</code> also asserts <code>lto_64access</code> to indicate to the local side that the current transaction request is 64-bit
4	PCI-X side: The PCI-X bus master asserts <code>pci_irdyn</code> to indicate that it is ready to accept data.
5	PCI-X side: After decoding the transaction address, the <code>pci_x</code> asserts <code>pci_devsel_n</code> and <code>pci_ack64_n</code> to claim the 64-bit transaction. Local side: The <code>lto_CycValid</code> signal is asserted to indicate that the <code>pci_x</code> has claimed the transaction. The <code>lto_Rdy</code> signal is also asserted to indicate that the PCI-X master is ready to accept data. With the assertion of the <code>lto_CycValid</code> signal, and because the local target is ready to provide data, the <code>lti_Rdy</code> signal is also asserted. The local side read data is registered in this clock cycle.
6	PCI-X side: With the assertion of <code>lti_Rdy</code> in clock 5, the <code>pci_x</code> asserts <code>pci_trdy_n</code> . Data transfer occurs in this clock cycle. Local side: Because the transaction byte count is eight, only one 64-bit data phase is required to expire the byte count. Therefore, <code>lti_Rdy</code> is deasserted and the local target completes the data transfers.
7	PCI-X side: The <code>pci_x</code> deasserts the <code>pci_trdy_n</code> , <code>pci_devsel_n</code> , and <code>pci_ack64_n</code> signals because the byte count has been satisfied. Local side: The <code>lto_Rdy</code> signal is deasserted to indicate that the PCI-X master is not ready to accept more data.
8	Local side: The <code>lto_CycValid</code> signal is deasserted to indicate to the local side that the current transaction has ended.

### PCI-X 64-Bit Burst Target Memory Read

Figure 2 shows a 64-bit burst target memory read transaction. The sequence of events for a burst read transaction is the same as a single-cycle read transaction; however, during a burst read transaction, more than one data transfer occurs. Figure 2 shows a 64-bit zero wait-state burst read transaction with four data phases. Four Qwords are transferred from the local side in clocks 5 through 8 and are then transferred to the PCI-X side in clocks 6 through 9.

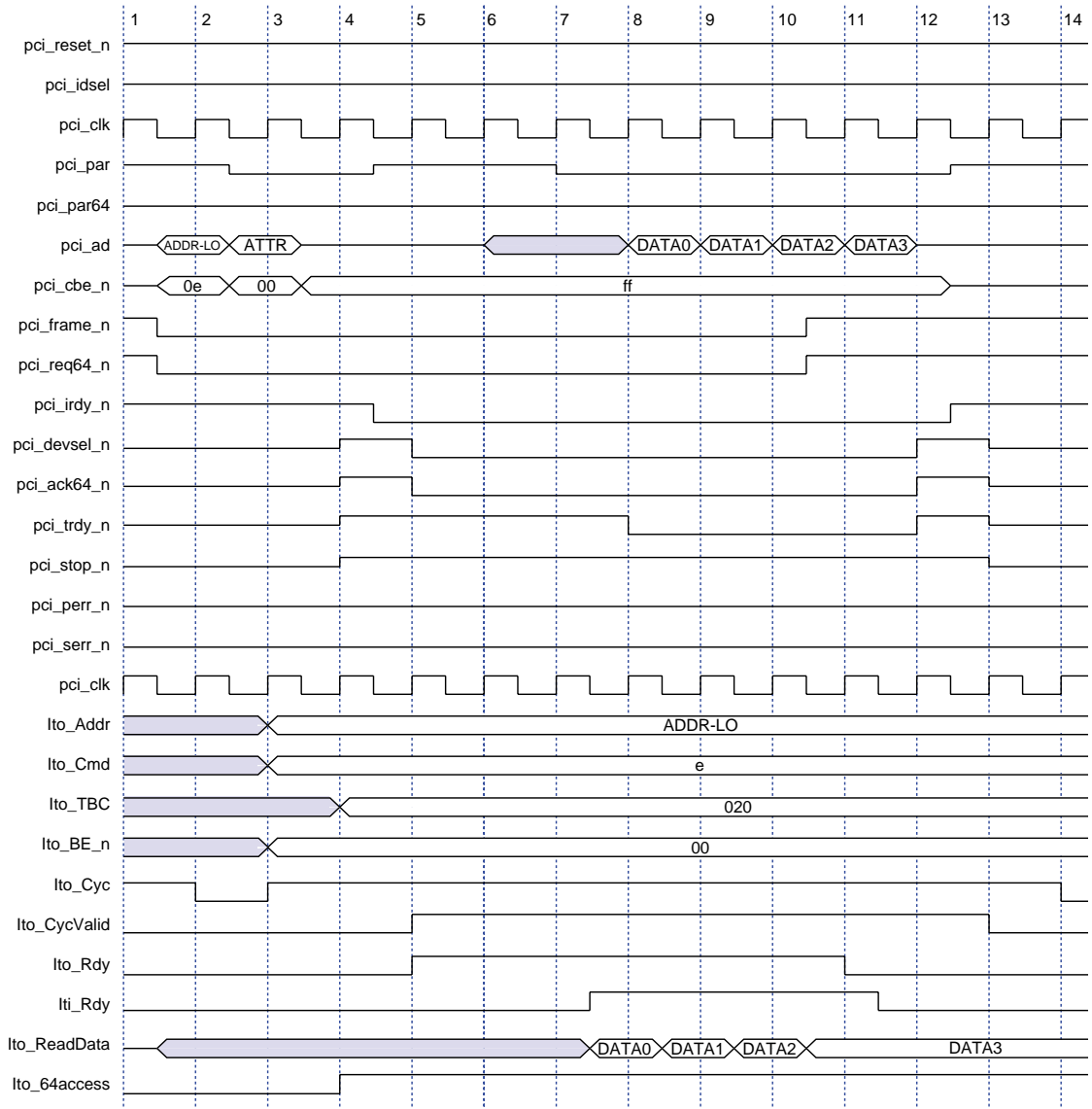
Figure 2. PCI-X 64-Bit Burst Target Memory Read Transaction



**PCI-X 64-Bit Burst Target Memory Read Transaction with Local Side Wait States**

Figure 3 shows the same transaction as Figure 2 except with the local target asserting initial wait states. In Figure 3, the local target is not ready to send data as soon as lto\_CycValid is asserted in clock 5. Thus, the local target asserts two initial wait states and does not assert lti\_Rdy until clock 7.

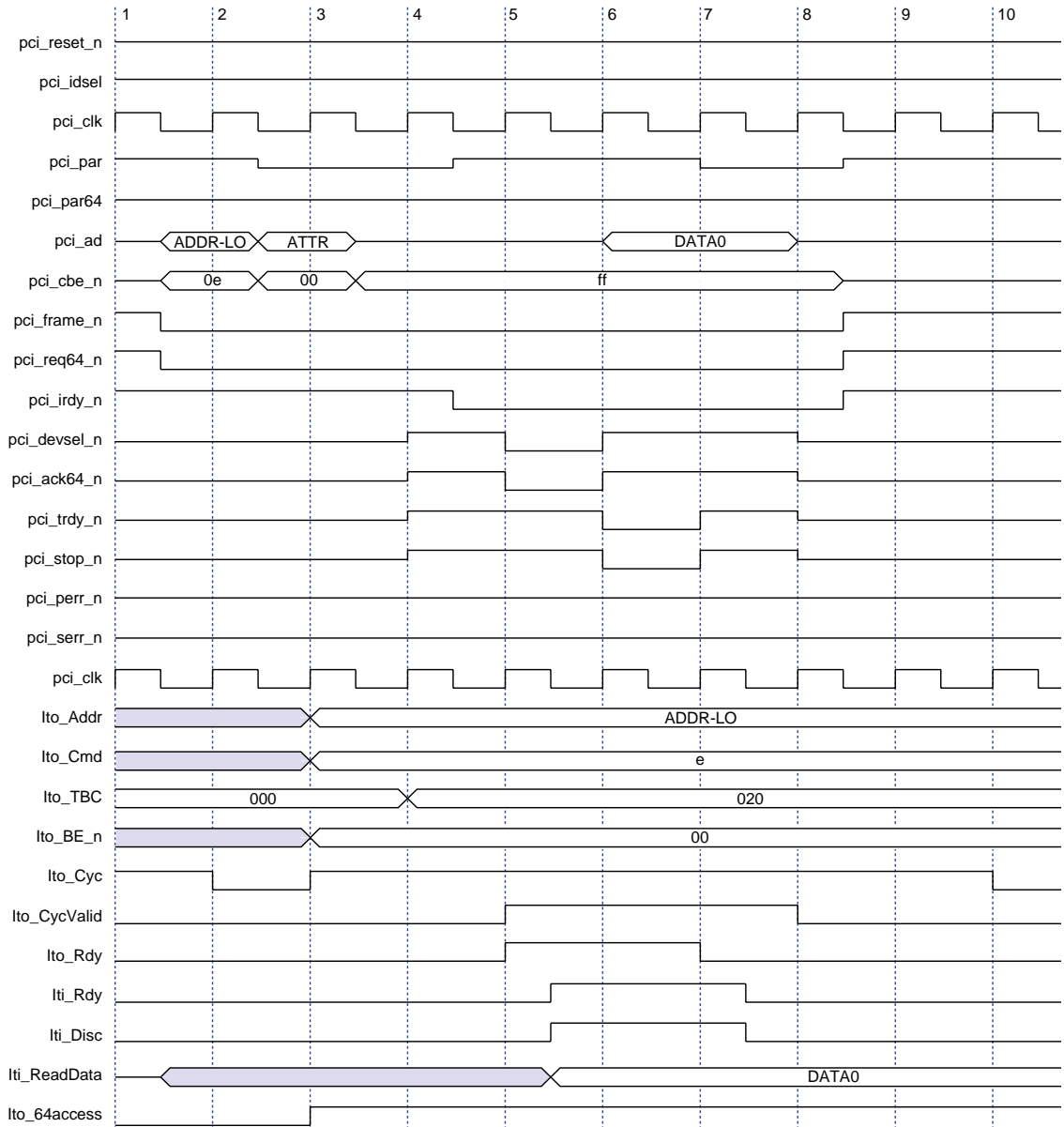
Figure 3. PCI-X 64-Bit Target Memory Read with Local Side Wait States



### PCI-X 64-Bit Target Memory Read with Single Data Phase Disconnect

Figure 4 is identical to Figure 1, with the local side indicating a single data phase disconnect. In Figure 4, the PCI-X master is requesting to transfer four QWORDS with `lto_TBC` signals set to 20 hexadecimal. However, the local target can only transfer 1 QWORD and therefore, asserts `lti_Disc` along with `lti_Rdy` in clock 5 to indicate a single data phase disconnect.

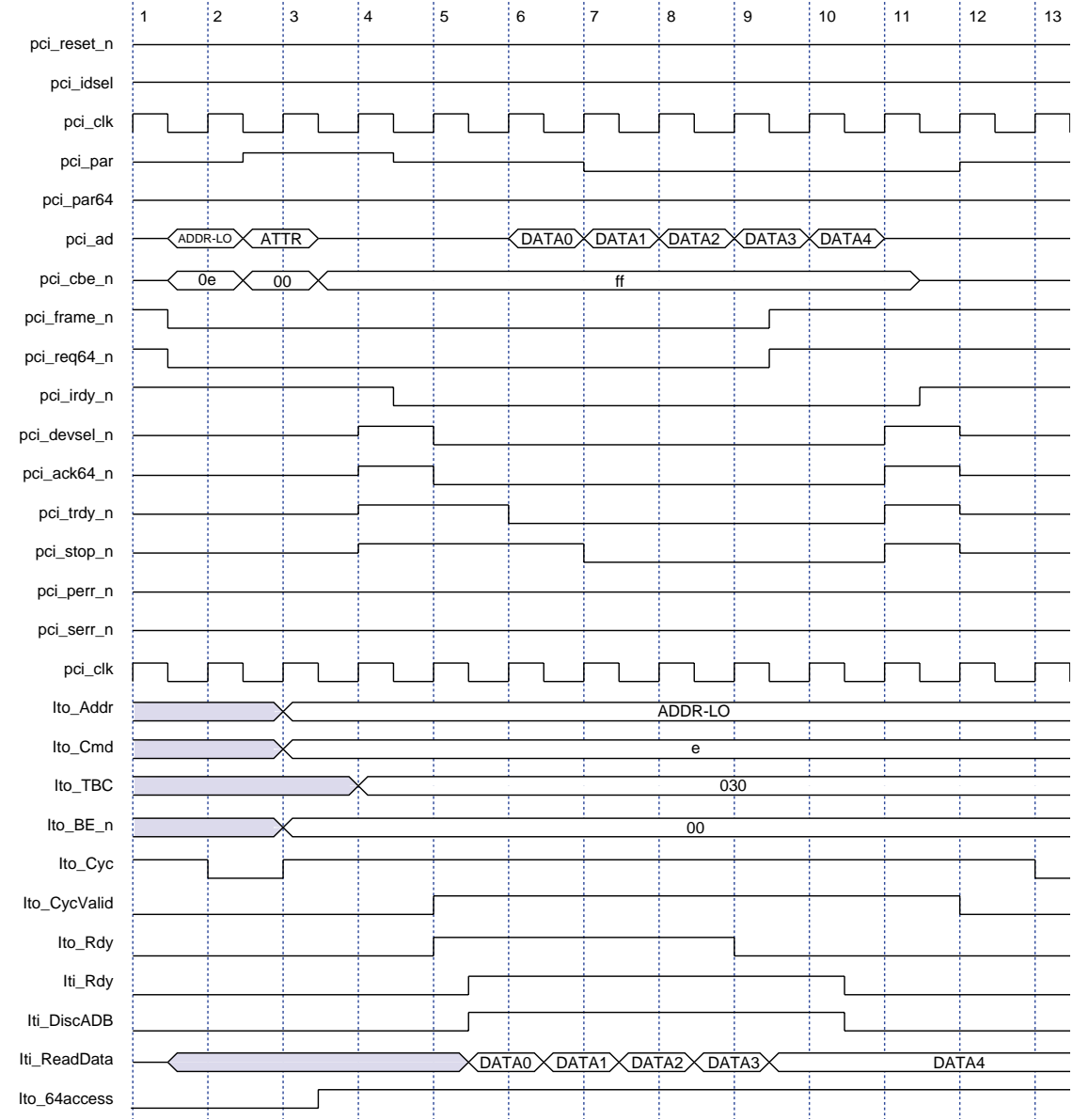
Figure 4. PCI-X 64-Bit Target Memory Read with Single Data Phase Disconnect



**PCI-X 64-Bit Target Memory Read Transaction with Disconnect at ADB**

Figure 5 is identical to Figure 2, with the local target requesting to disconnect at the ADB by asserting `lti_DiscADB`. In Figure 5, the starting address is 5 Qwords from the ADB, and the PCI-X master device is requesting to transfer 6 Qwords (`lto_TBC=12'h12'h030`). The local target asserts `lti_DiscADB` at the same clock cycle it asserts `lti_Rdy` in clock 5. As a result, the `pci_x` function transfers 1 Qword with the assertion of `pci_trdy_n` in clock 6, and in the following 4 clock cycles, the `pci_x` function acknowledges the request to disconnect at the ADB by asserting `pci_trdy_n` and `pci_stop_n`.

Figure 5. PCI-X 64-Bit Target Memory Read Transaction with Disconnect at ADB



In PCI-X protocol, the `pci_x` responds to three types of 32-bit target read transactions:

- Memory read
- I/O read
- Configuration read



For 32-bit single data phase PCI-X cycles (e.g., DWORD memory, I/O, and configuration cycles) where the local target device has immediate data, the `pci_x` will pick up the data on either the upper or lower 32 bits of the 64-bit local data bus (i.e., `lti_ReadData[63:0]`). The value of bit 2 in the address determines whether the data is picked up in the upper or lower boundary of the data bus. The `pci_x` will also adjust byte enables, disabling byte enables for the 32-bit half that is not requested.

### PCI-X 32-bit Target Memory Read Transactions

Memory transactions are either single-cycle or burst. For memory transactions, the `pci_x` function always assumes a 64-bit local side. For a 32-bit PCI bus, the `pci_x` function automatically reads 64-bit data on the local side and transfers the data to the PCI-X master device, one DWORD at a time.

Figure 6 shows a 32-bit single-cycle target memory read transaction. The sequence of events in Figure 6 is identical to Figure 1, except for the following:

- During the address phase (clock 1), the PCI-X master does not assert `pci_req64_n`. The PCI-X master uses the memory read DWORD command (`pci_cbe_n[3:0] = 4'h6`) to specifically request a single-cycle 32-bit transaction.
- The `pci_x` function does not assert `pci_ack64_n` when it asserts `pci_devsel_n`.
- The `lto_64access` signal is not asserted to indicate (to the local side) that the current transaction is 32-bits.

Figure 6 shows that with the assertion of `lty_Rdy`, the local side transfers a full Qword in clock 5. The `pci_x` function, however, transfers only the upper Dword (`lty_ReadData[63:32]`) to the PCI-X master because the starting address is at a high Dword boundary, i.e., `pci_ad[2] = 1'b1`. This is indicated by `lto_BE_n[7:0] = 8'h0F`, where a value “0” indicates valid byte enables and a value “F” indicates invalid byte enables.

Figure 6. PCI-32 Bit Single-Cycle Target Memory Read Transaction

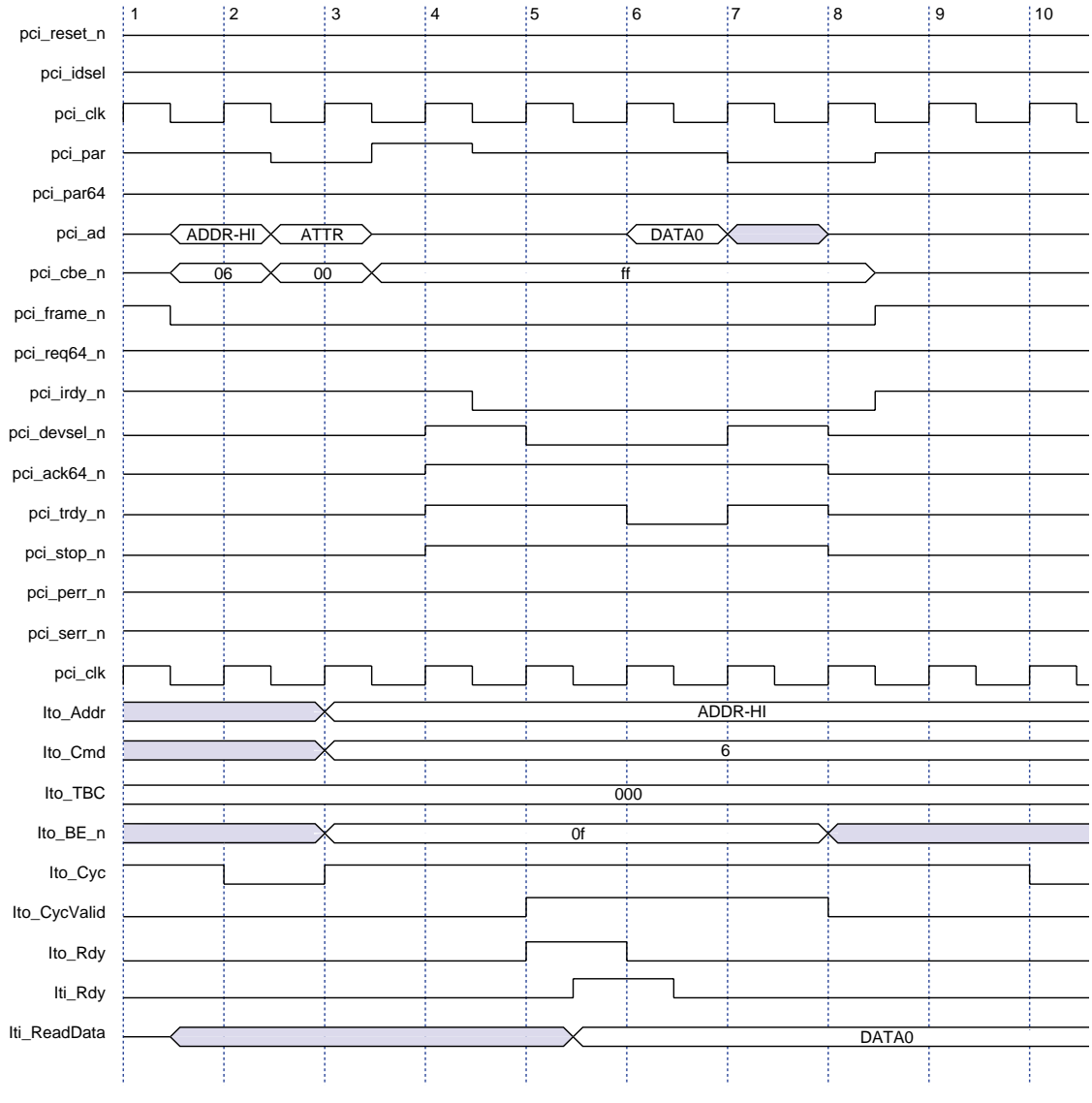


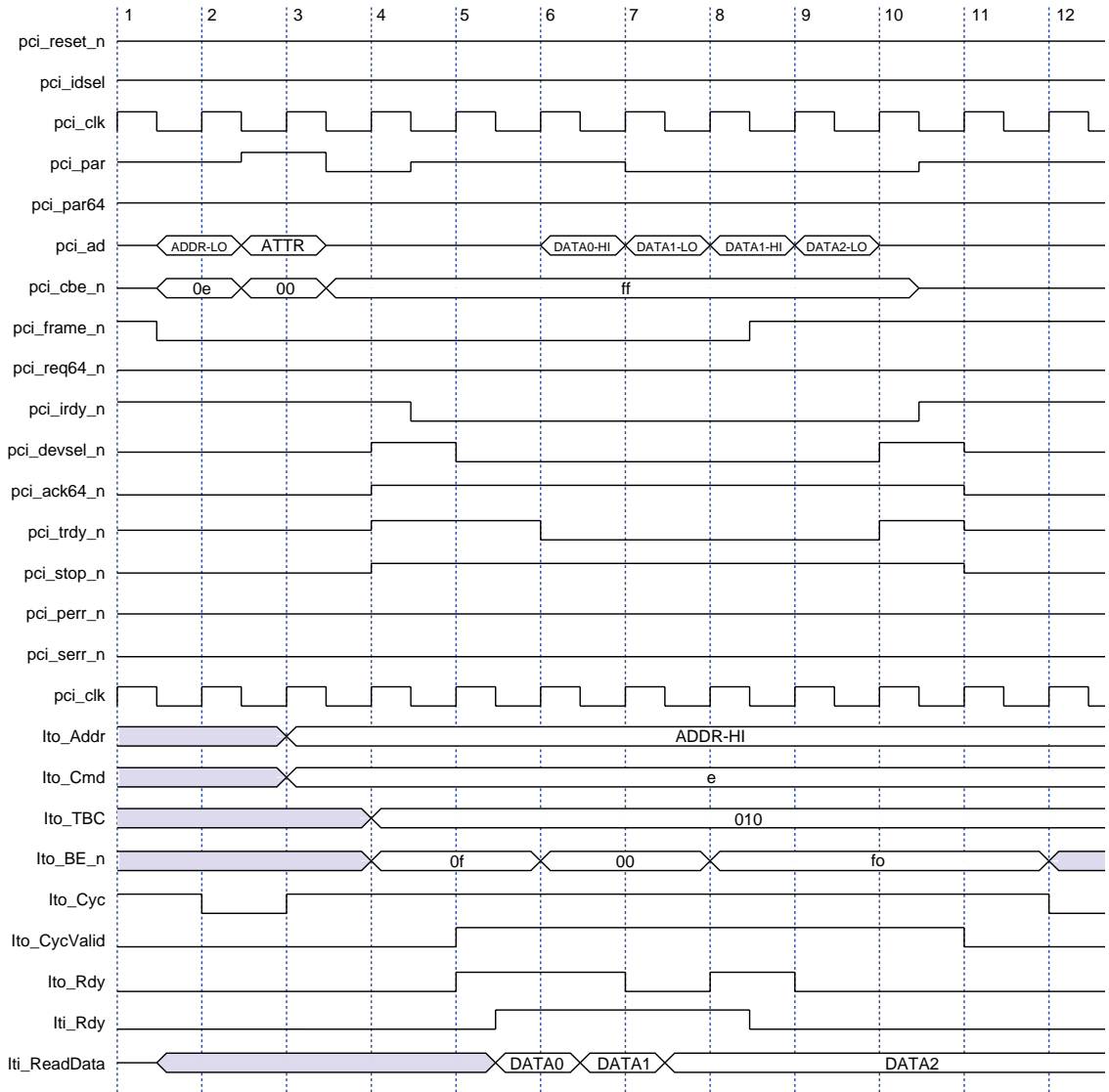
Figure 7 shows a 32-bit burst target memory read transaction. The sequence of events in Figure 7 is identical to Figure 6, except for the following:

- During the address phase (clock 1), the PCI-X master intends to do a 32-bit burst memory read by not asserting `pci_req64_n` and driving the memory read block command (`pci_cbe_n[3:0] = 4'hE`).

- The `pci_x` function transfers more than one data phase.

Figure 7 shows that the PCI-X master is requesting to transfer 16 bytes (`lto_TBC = 12'h010`). Because Figure 7 represents a 32-bit cycle, the transaction will require four data phases to complete the byte count. On clock 5, the `pci_x` function registers only the upper Dword of `DATA0` (indicated by `lto_BE_n = 8'h0F`) and transfers it to the PCI-X master in clock 6. In the following clock cycles, the `pci_x` function registers three more Dwords and transfers the data to the PCI-X master to satisfy the byte count request.

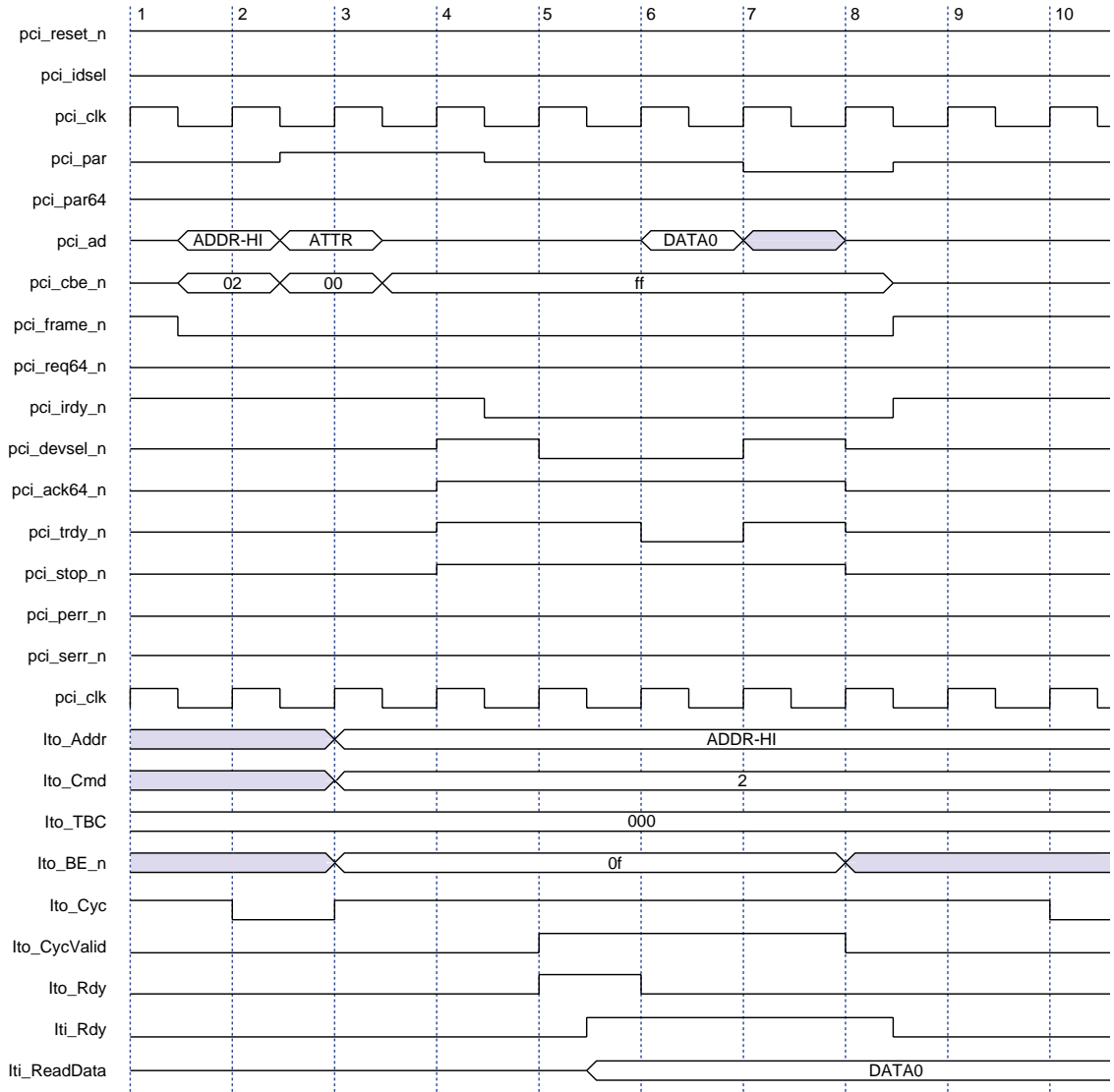
Figure 7. PCI-X 32-Bit Burst Target Memory Read Transaction



### PCI-X Target I/O Read Transaction

By definition, a PCI-X I/O transaction is 32-bit and single-cycle. [Figure 8](#) shows an I/O read transaction, where bit 2 of the address is 1'b1 (`pci_ad[2] = 1'b1`). The `pci_x` uses this information to determine where to register the data and drive the byte enables. Because the I/O read was to the high Dword, the data that appears on `lti_ReadData[63:32]` is driven on `pci_ad[31:0]`. In addition, the byte enables that appear on `pci_cbe_n[3:0]` also appear on `lto_BE_n[7:4]`. The `pci_x` intentionally disables the lower byte enables, i.e., `lto_BE_n[3:0] = 4'hF`.

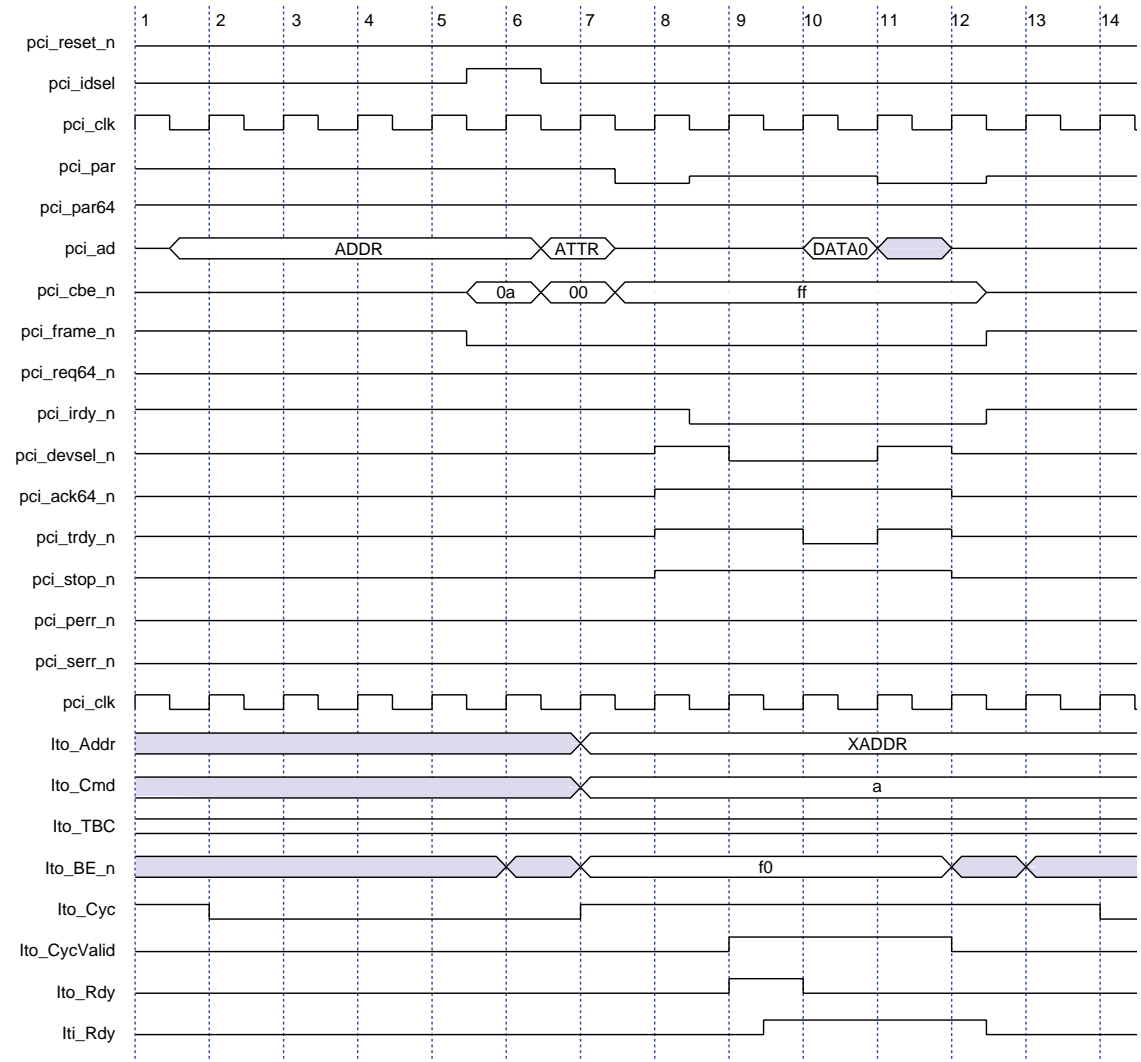
Figure 8. PCI-X I/O Read Transaction



### PCI-X Target Configuration Read Transaction

By definition, a PCI-X configuration transaction is 32-bit and single-cycle. **Figure 9** shows configuration read transaction, where bit 2 of the address is 1'b0 (`pci_ad[2] = 1'b0`). The `pci_x` uses this information to determine where to register the data and drive the byte enables. Because the I/O read was to the low Dword, the data that appears on `lti_ReadData[31:0]` is driven on `pci_ad[31:0]`. In addition, the byte enables that appear on `pci_cbe_n[3:0]` also appear on `lto_BE_n[3:0]`. The `pci_x` intentionally disables the upper byte enables, i.e., `lto_BE_n[7:4] = 4'hF`.

Figure 9. PCI-X Configuration Read Transaction



PCI-2.2 Target Read Transactions

As stated earlier, from the perspective of the local target bus the behavior of the `pci_x` during PCI-2.2 cycles is similar to the behavior during PCI-X cycles. Also, because the PCI bus can be 32-bit or 64-bit and the local target is always 64-bit, the `pci_x` does some data and byte-enable manipulation to accommodate for the data mismatch.

The following are some general operating rules for PCI-2.2 target read transactions:



- For memory burst cycles, whether there is a 32-bit or 64-bit PCI master device running the cycle, the `pci_x` will accept read data from the local target bus in 64-bit increments.
- For 32-bit single data phase PCI-2.2 cycles (e.g., memory, I/O, and configuration cycles), the `pci_x` will pick up the data on either the lower or upper 32 bits of the 64-bit local data bus (`lti_ReadData[63:0]`). The value of bit 2 in the address determines whether the data is picked up in the upper or lower boundary of the data bus. The MegaCore will also adjust the byte enables, disabling the byte enables for the 32-bit half that was not requested.
- For all read cycles, parity error information (`lti_Perr`) for a particular data phase can be driven by the local target along with the assertion of `lti_Rdy`. At this point, the `pci_x` intentionally drives the data with bad parity on the PCI bus.
- The address presented to the local bus is the same as the PCI-2.2 address. Therefore, when the `pci_x` is operating in PCI-2.2 protocol, the address presented for a memory read cycle is either Dword- or Qword-aligned, depending on the data width of the PCI master device. I/O cycles have byte-aligned addresses in PCI-2.2.
- PCI-2.2 delayed transaction support: The `pci_x` does not make any assumptions as to whether the local target is going to provide data immediately during reads (e.g., memory, I/O, or configuration cycles) or use the PCI-2.2 delayed transaction protocol to get the read data from the final destination. When a local target retries the `pci_x` during a read cycle, the `pci_x` does not know if any cycle has been enqueued or not. Local targets using the PCI-2.2 delayed transaction protocol are responsible for storing the relevant transaction information. The `pci_x` will however, throughout the assertion of `lto_Cyc`, maintain a stable value for address (`lto_Addr`), command (`lto_Cmd`), and 64-bit PCI master information (`lto_64access`) to assist in the storing of the transaction information.

In PCI-2.2 protocol, the `pci_x` support two types of 64-bit target 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:

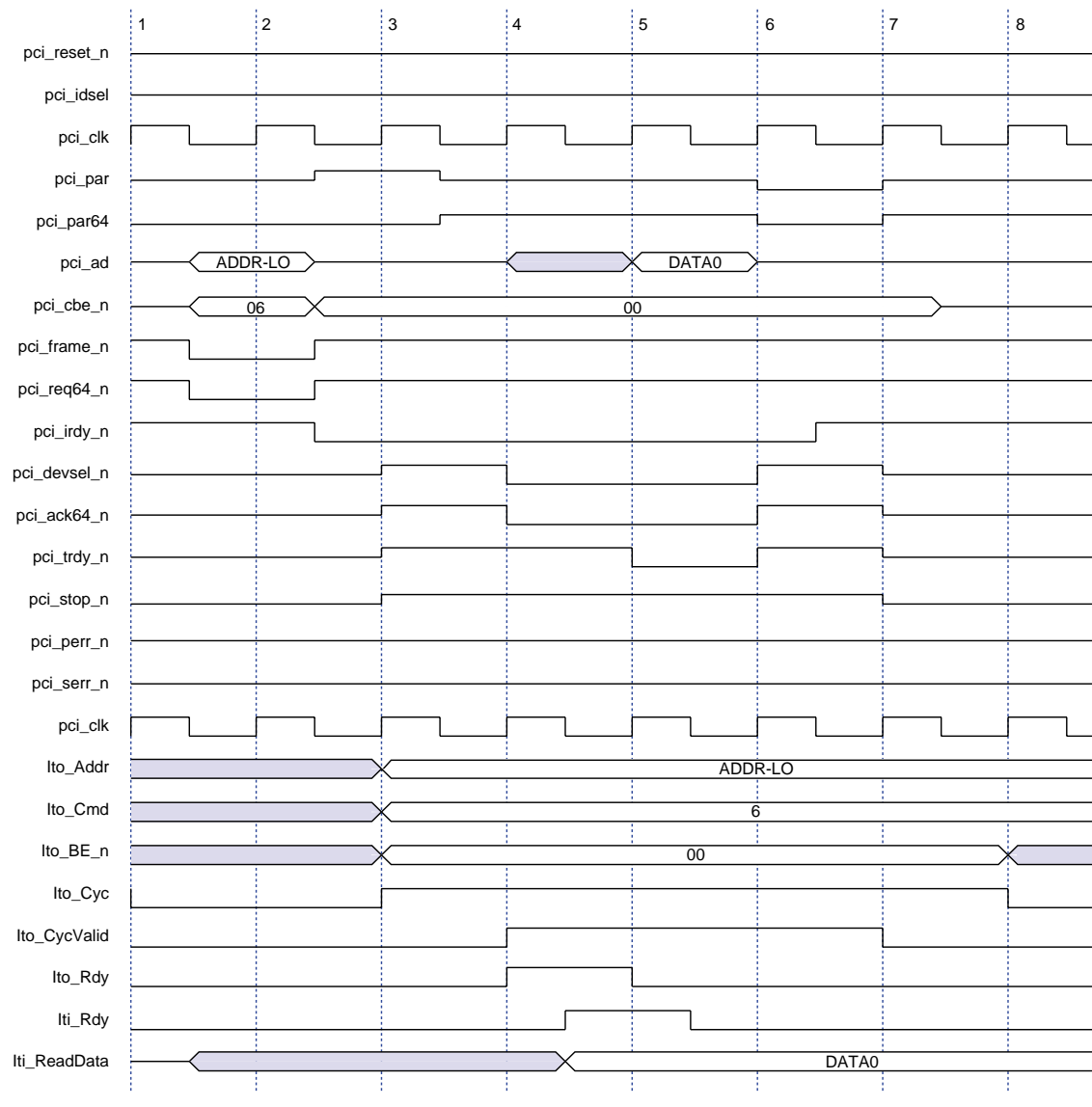
1. The address phase occurs when the PCI-X master asserts `pci_frame_n` and `pci_req64_n` signals and drives the address and command on `pci_ad[31:0]` and `pci_cbe_n[3:0]`, correspondingly. Asserting the `pci_req64_n` signal indicates to the target device that the master device is requesting a 64-bit data transaction.
2. Turn-around cycles on the `pci_ad[63:0]` bus occur during the clock immediately following the address phase. During the turn-around cycles, the PCI master tri-states the `pci_ad[63:0]` bus. This process is necessary because the PCI agent driving the `pci_ad[63:0]` bus changes during read cycles.
3. If the address of the transactions matches one of the base address registers, the `pci_x` function turns on the drivers for the `pci_ad[63:0]`, `pci_devsel_n`, `pci_ack64_n`, `pci_trdy_n`, and `pci_stop_n` signals. The drivers for `pci_par` and `pci_par64` are turned on in the following clock.
4. The `pci_x` function drives and asserts `pci_devsel_n` and `pci_ack64_n` 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.

### PCI-2.2 64-Bit Single-Cycle Target Memory Read Transaction

The sequence of events for a 64-bit PCI-2.2 single-cycle target memory read transaction (shown in Figure 10) is the same as a 64-bit PCI-X single-cycle target memory read transaction (shown in Figure 1), except for the following:

- There is no attribute phase (clock 2).
- If the `pci_x` function is the target of a transaction, it claims the transaction three clocks after the assertion of `pci_frame_n` by asserting `pci_devsel_n`. The `pci_x` function is a slow decode device in PCI-2.2 protocol and a decode C device in PCI-X protocol.

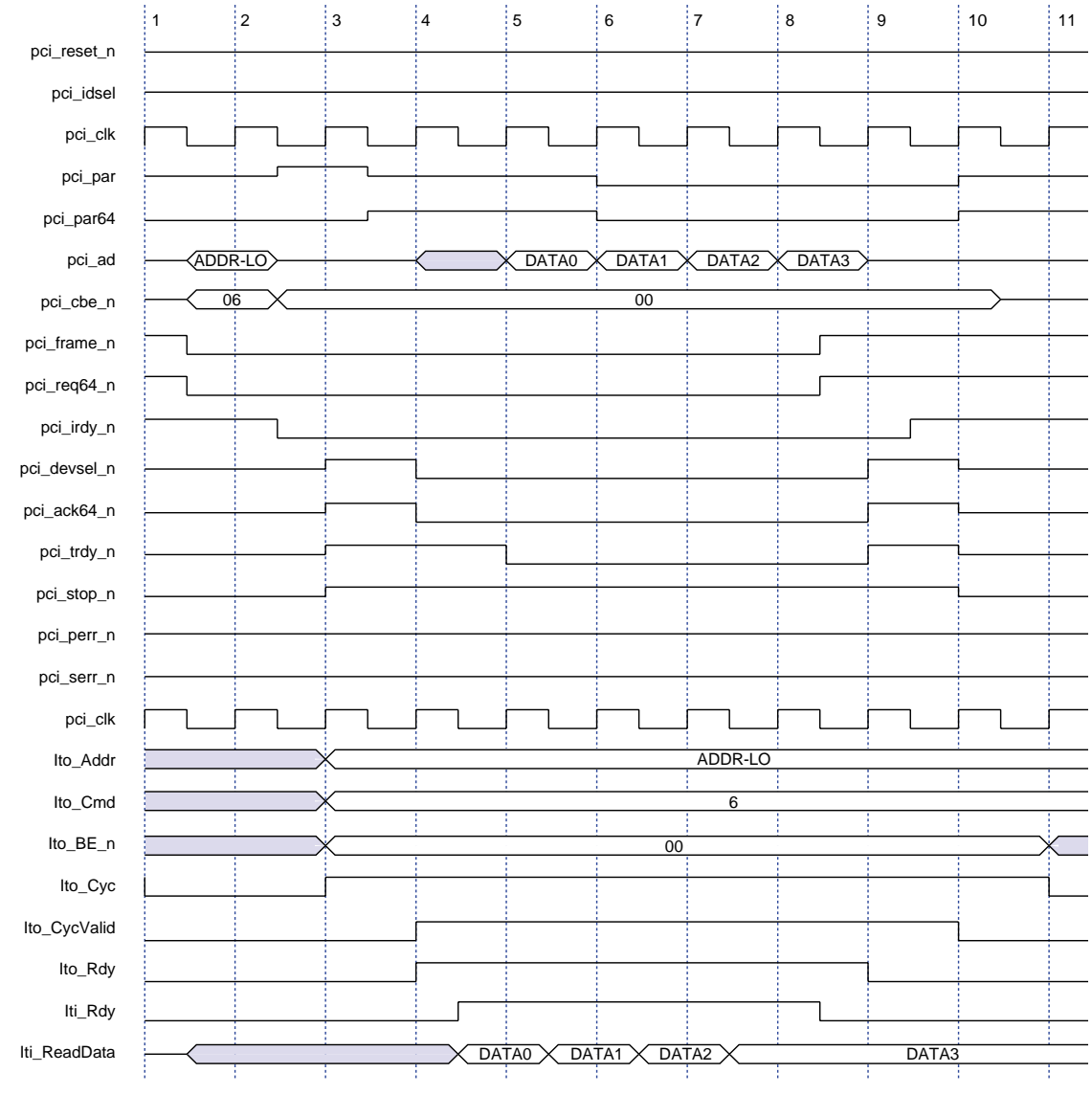
Figure 10. PCI 64-Bit Single-Cycle Target Memory Read Transaction



### PCI-2.2 64-Bit Burst Target Memory Read Transaction

Figure 11 shows a 64-bit burst target memory read transaction. It is similar to Figure 10, except that Figure 11 describes a four Qword data transfer transaction.

Figure 11. PCI-2.2 64-Bit Burst Target MemoryRead Transaction



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

- Memory read
- I/O read
- Configuration read

### PCI 32-bit Target Memory Read Transactions

Memory transactions are either single- or burst-cycle. For memory transactions, the `pci_x` always assumes a 64-bit local side. Assuming a 32-bit PCI bus, the `pci_x` function automatically reads 64-bit data on the local side and transfers the data to the PCI master, one DWORD at a time.

Figure 12 shows a 32-bit single-cycle target memory read transaction. The sequence of events in Figure 12 is identical to Figure 10, except for the following:

- During the address phase (clock 1), the PCI-X master does not assert `pci_req64_n` to request a 32-bit transaction.
- The `pci_x` function does not assert `pci_ack64_n` when it asserts `pci_devsel_n`.
- The local side signal, `lto_64access`, is not asserted to indicate that the current transaction is 32-bits.

Figure 12 shows that the local side transfers a full Qword in clock four with the assertion of `lti_Rdy`. The `pci_x` function, however, transfers only the upper Dword (`lti_ReadData[63:32]`) to the PCI master because the starting address is at a high Dword boundary, i.e., `pci_ad[2] = 1'b1`. This is indicated by `lto_BE_n[7:0] = 8'h0F`, where a value “0” indicates valid byte enables and a value “F” indicates invalid byte enables.

Figure 12. PCI 32-Bit Single-Cycle Target Memory Read Transaction

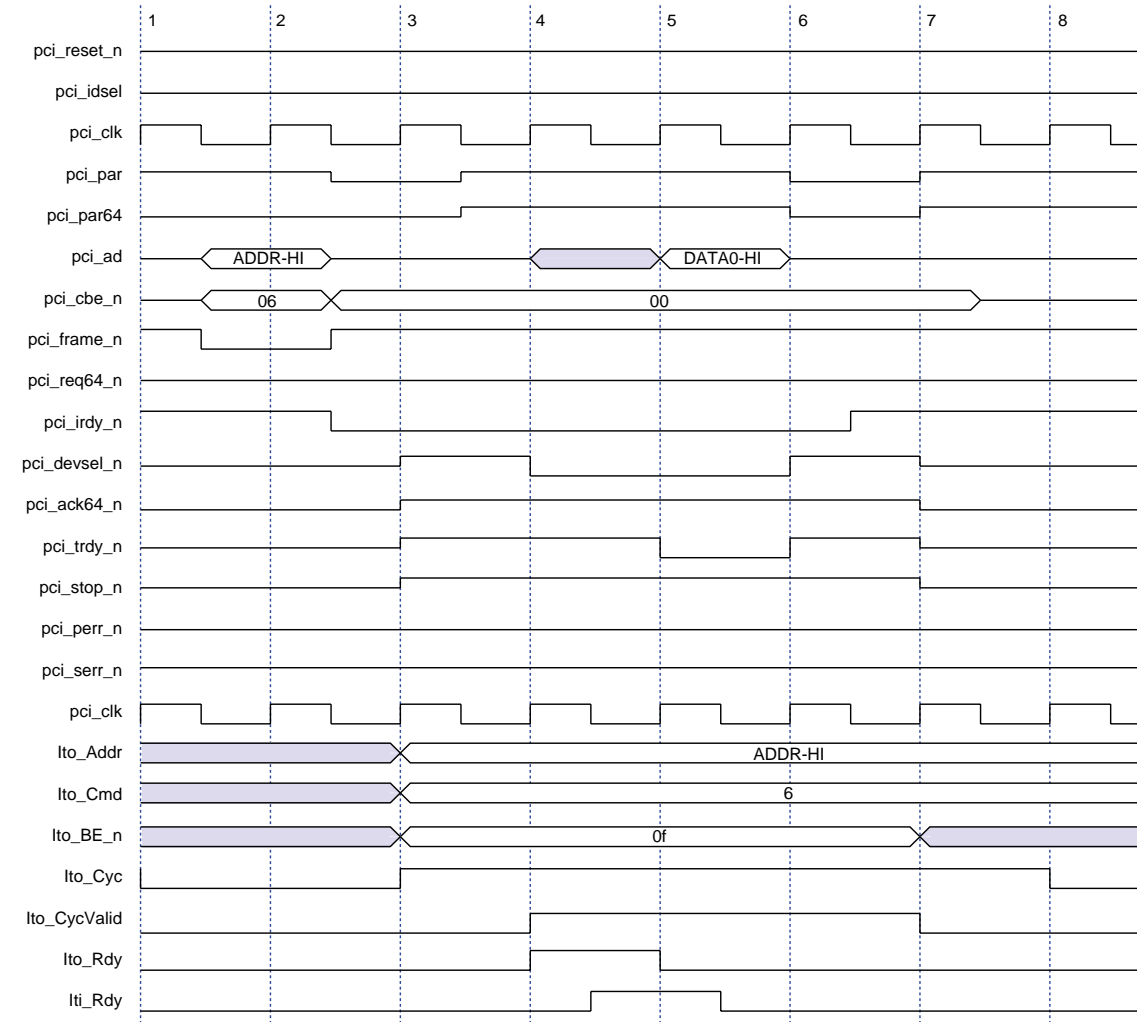
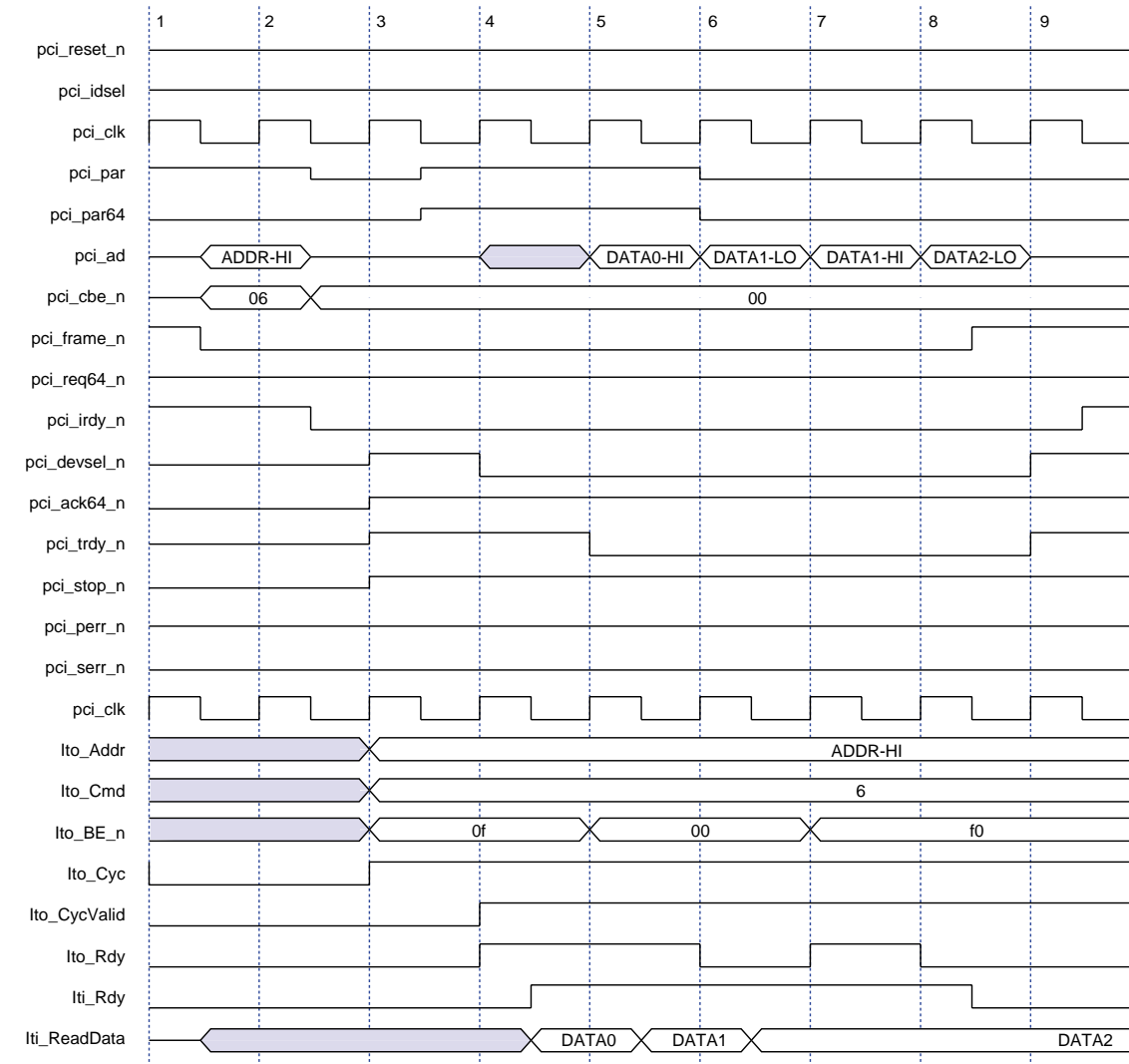


Figure 13 shows a 32-bit burst target memory read transaction. The sequence of events in Figure 13 is identical to Figure 12, except for the multiple data phases. In clock 4, the `pci_x` function registers only the upper Dword of DATA0 (indicated by `lto_BE_n[7:0] = 8'h0F`) and transfers it to the PCI master device in clock 5. In the following clock cycles, the `pci_x` function registers three more Dwords and transfers the data to the PCI master device.

Figure 13. PCI-2.2 32-Bit Burst Target Memory Read Transaction



PCI-2.2 target I/O and configuration read transactions are nearly identical to PCI-X transactions, i.e., PCI-X transactions have an attribute phase. Go to “PCI-X Target I/O Read Transaction” on page 53 and “PCI-X Target Configuration Read Transaction” on page 55 for more information.

## Target Write Transactions

The `pci_x` function performs both PCI-X and PCI-2.2 target write transactions. There are a few major differences between the PCI-X and PCI-2.2 target write transactions; however, from the perspective of the local target bus, behavior of the `pci_x` during the transactions is similar.

PCI-X transactions include an attribute phase, block cycle support, and split transaction support. See “Target Read Transactions” on page 38 for more information.

The following are some general operating rules for PCI-X and PCI-2.2 target write transactions:

- Because the PCI-X or PCI-2.2 bus can operate as a 64- or 32-bit bus and the local target bus is 64-bit, the `pci_x` performs some data and byte enable manipulation to compensate for the data-width mismatch.
- For 32-bit single data phase PCI-X or PCI-2.2 cycles (i.e., Dword memory, I/O, and configuration cycles), the `pci_x` function will place the data on the lower or upper 32-bits of the 64-bit local data bus (`lto_WriteData`). The value of bit 2 in the address determines whether the data is picked up in the upper or lower boundary of the data bus. The `pci_x` will also adjust the byte enables, disabling the byte enables for the 32-bit half that was not transferred.
- For all PCI-X and PCI-2.2 memory writes, parity information for a particular data phase appears two clocks after the assertion of `lto_Rdy`. The `pci_x` calculates expected parity and drives `lto_Perr` to 1'b1 to indicate that the data driven with `lto_Rdy` two clocks earlier has bad parity. Local targets can choose to use the parity information either on a per data phase basis or a per transaction basis.
- The `pci_x` function will continue to treat configuration and I/O writes differently from memory writes. For memory writes, `lto_Rdy` is asserted after data transfers have taken place on the PCI-X or PCI-2.2 bus. However for configuration and I/O writes, `lto_Rdy` is asserted and the data is driven on the local target bus before any data transfers have taken place on the PCI-X or PCI-2.2 bus. The `pci_x` first waits for data and parity information from the PCI-X or PCI-2.2 bus. When the `pci_x` asserts `lto_Rdy`, `lto_WriteData` shows the data being driven by the PCI-X or PCI-2.2 master device. The value of `lto_Perr` in the same clock informs the local target whether the data being written has good or bad parity. Local targets can use this feature to prevent overwriting critical registers with bad data.



### *PCI-X Target Write Transactions*

The following are some general operating rules for PCI-X target write transactions:

- For burst memory write cycles (i.e., memory write or memory write block), regardless of whether there is a 32-bit or 64-bit PCI-X master running the cycle, the `pci_x` will forward the write data to the local target bus in 64-bit increments. While the PCI-X master device does not drive any byte enables for a memory write block transaction (`pci_be_n` bus is reserved drive high), the `pci_x` will generate valid byte enables on the local target `lto_BE_n` lines on a per data phase basis.
- The address presented to the local bus is the same as the PCI-X address. Therefore, when the `pci_x` is operating in PCI-X protocol, the address specified on `lto_Addr` is byte aligned, except for configuration transactions, which are DWORD aligned (indicates the configuration transaction type).

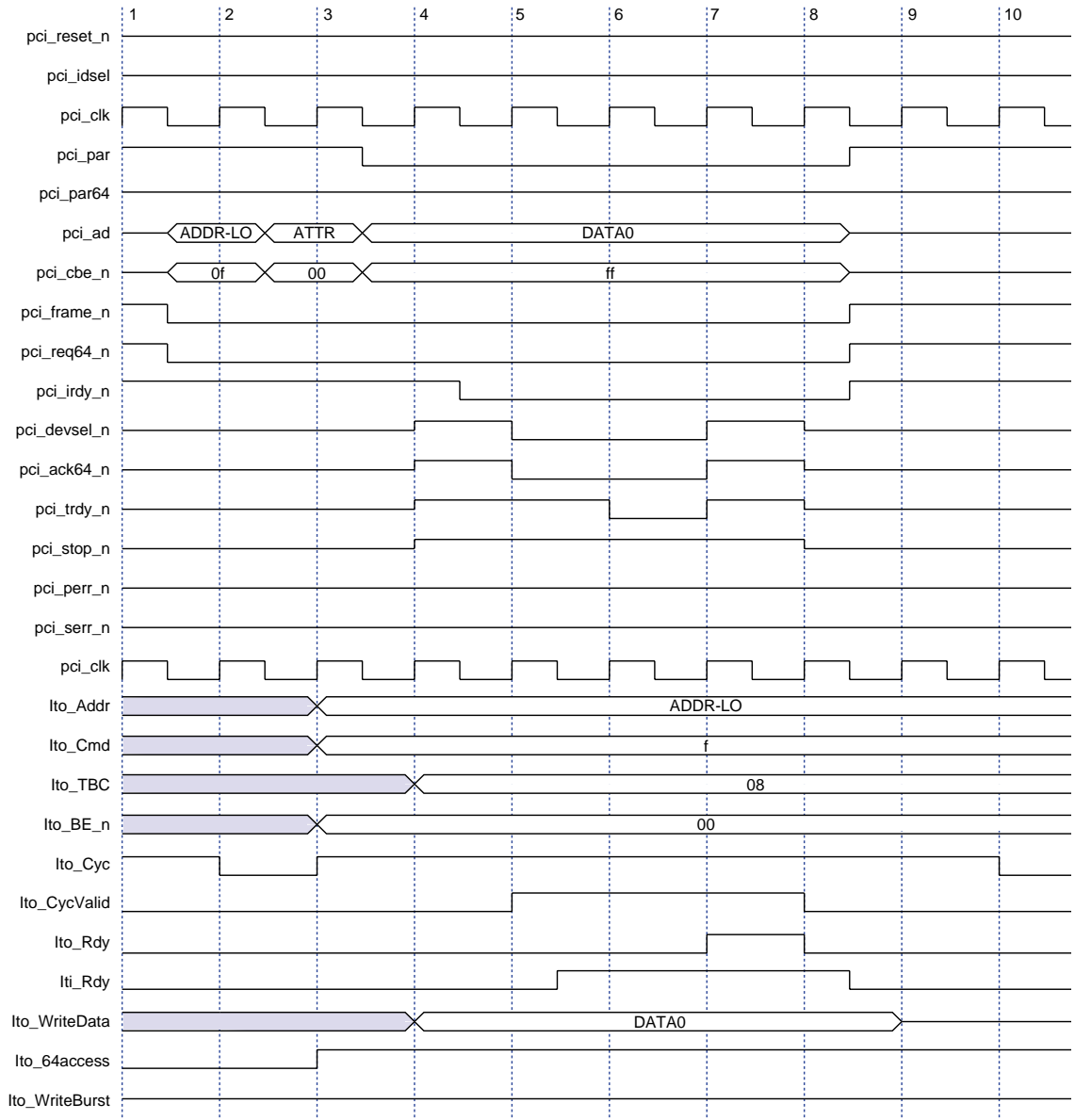
In PCI-X protocol, the `pci_x` function responds to two types of 64-bit target write transactions:

- Memory single-cycle write
- Memory burst write

#### **PCI-X 64-Bit Single-Cycle Target Memory Write Transaction**

Figure 14 shows the waveforms for a PCI-X 64-bit single-cycle target memory write transaction.

Figure 14. PCI-X 64-Bit Single-Cycle Target Memory Write Transaction



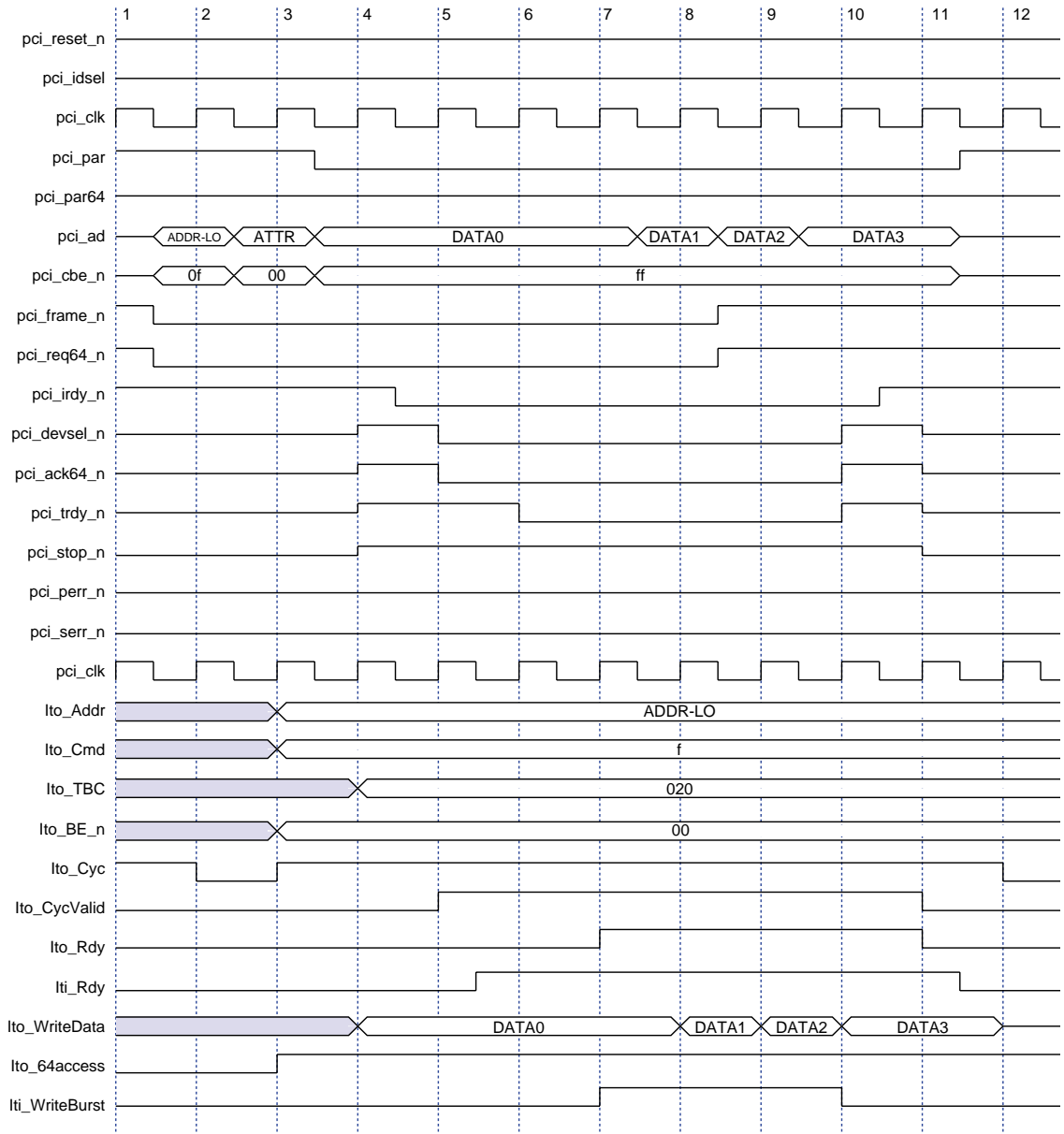
**Table 3** shows the sequence of events for a PCI-X 64-bit single-cycle target memory write transaction.

<i>Table 3 .Single-Cycle Target Memory Write Transaction</i>	
Clock Cycle	Event
1	PCI-X side: The address phase. A PCI-X master device asserts <code>pci_frame_n</code> to indicate the beginning of a transaction. The <code>pci_req64_n</code> is asserted to indicate that a 64-bit transaction is being requested.
2	PCI-X side: The attribute phase. Additional information about the transaction is provided on <code>pci_ad[31:0]</code> and <code>pci_cben[3:0]</code> . In this transaction, the total byte count transferred is eight. Local side: The <code>pci_x</code> latches the address and command data, and then decodes the address to verify that it falls within the BAR range.
3	PCI-X side: Turn-around cycle on the <code>ad[63:0]</code> bus. Local side: The <code>lto_Cyc</code> is asserted to indicate that a PCI-X transaction is occurring. The <code>pci_x</code> drives the transaction address on the <code>lto_Addr[31:0]</code> bus and the command data on the <code>lto_Cmd[3:0]</code> bus. The <code>pci_x</code> also asserts <code>lto_64access</code> to indicate (to the local side) that the current transaction request is 64-bit.
4	PCI-X side: The PCI-X master device asserts <code>pci_irdy_n</code> to indicate that it is ready to send data.
5	PCI-X side: After decoding the transaction address, the <code>pci_x</code> function asserts <code>pci_devsel_n</code> and <code>pci_ack64_n</code> to claim the 64-bit transaction. Local side: The <code>lto_CycValid</code> signal is asserted to indicate that the <code>pci_x</code> function has claimed the transaction. With the assertion of <code>lto_CycValid</code> , and because the local target is ready to provide data, <code>lti_Rdy</code> is also asserted.
6	PCI-X side: With the assertion of <code>lti_Rdy</code> in the clock 5, the <code>pci_x</code> function asserts <code>pci_trdy_n</code> . Data transfer occurs in this clock cycle.
7	PCI-X side: The <code>pci_x</code> function deasserts <code>pci_trdy_n</code> , <code>pci_devsel_n</code> , and <code>pci_ack64_n</code> because the byte count has been satisfied. Local side: The <code>lto_Rdy</code> signal is asserted to indicate that the <code>lto_WriteData[63:0]</code> data is valid.
8	Local side: The <code>lto_CycValid</code> signal is deasserted to indicate to the local side that the current transaction has ended. Also, the local target deasserts <code>lti_Rdy</code> because the transaction has ended.

### PCI-X 64-Bit Burst Target Memory Write Transaction

**Figure 15** shows a 64-bit burst target memory write transaction. The sequence of events for a burst write transaction is the same as a single-cycle write transaction; however, during a burst write transaction more than one data transfer occurs. **Figure 15** shows a 64-bit zero wait-state burst write transaction with four data phases. Four Qwords are transferred from the PCI-X side in clocks 6 through 9 and are then transferred to the local side in clocks 7 through 10. In addition, the assertion of `lto_WriteBurst` indicates that another data transfer should occur in the following clock cycle.

Figure 15. PCI-X 64-Bit Burst Target Memory Write Transaction

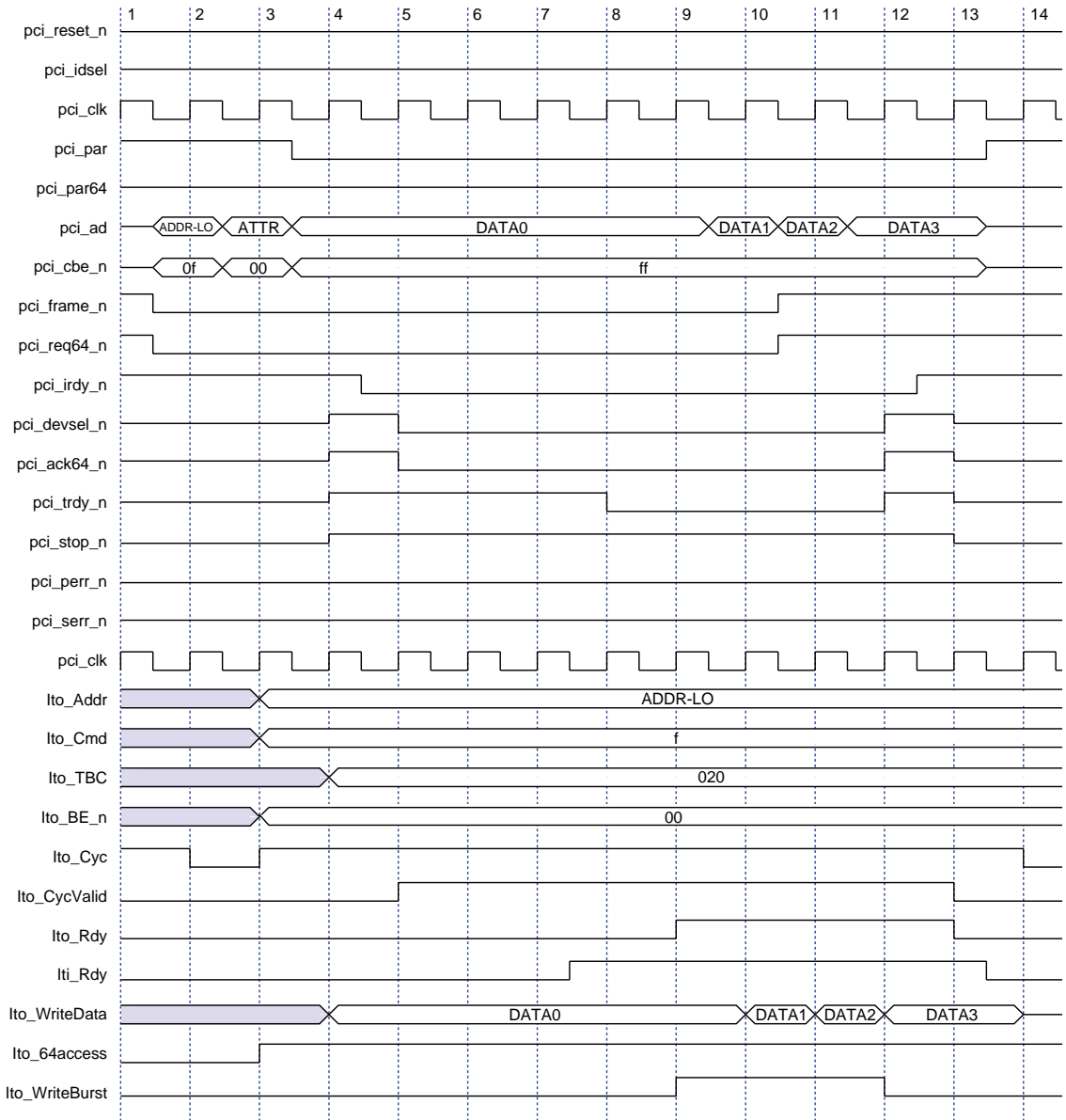


---

### PCI-X 64-Bit Burst Target Memory Write Transaction with Wait States

Figure 16 is identical to Figure 15, except with the local target asserting initial wait-states. In Figure 16, the local target is not ready to accept data as soon as `lto_CycValid` is asserted in clock 5. Thus, the local target asserts two initial wait-states and does not assert `lti_Rdy` until clock 7.

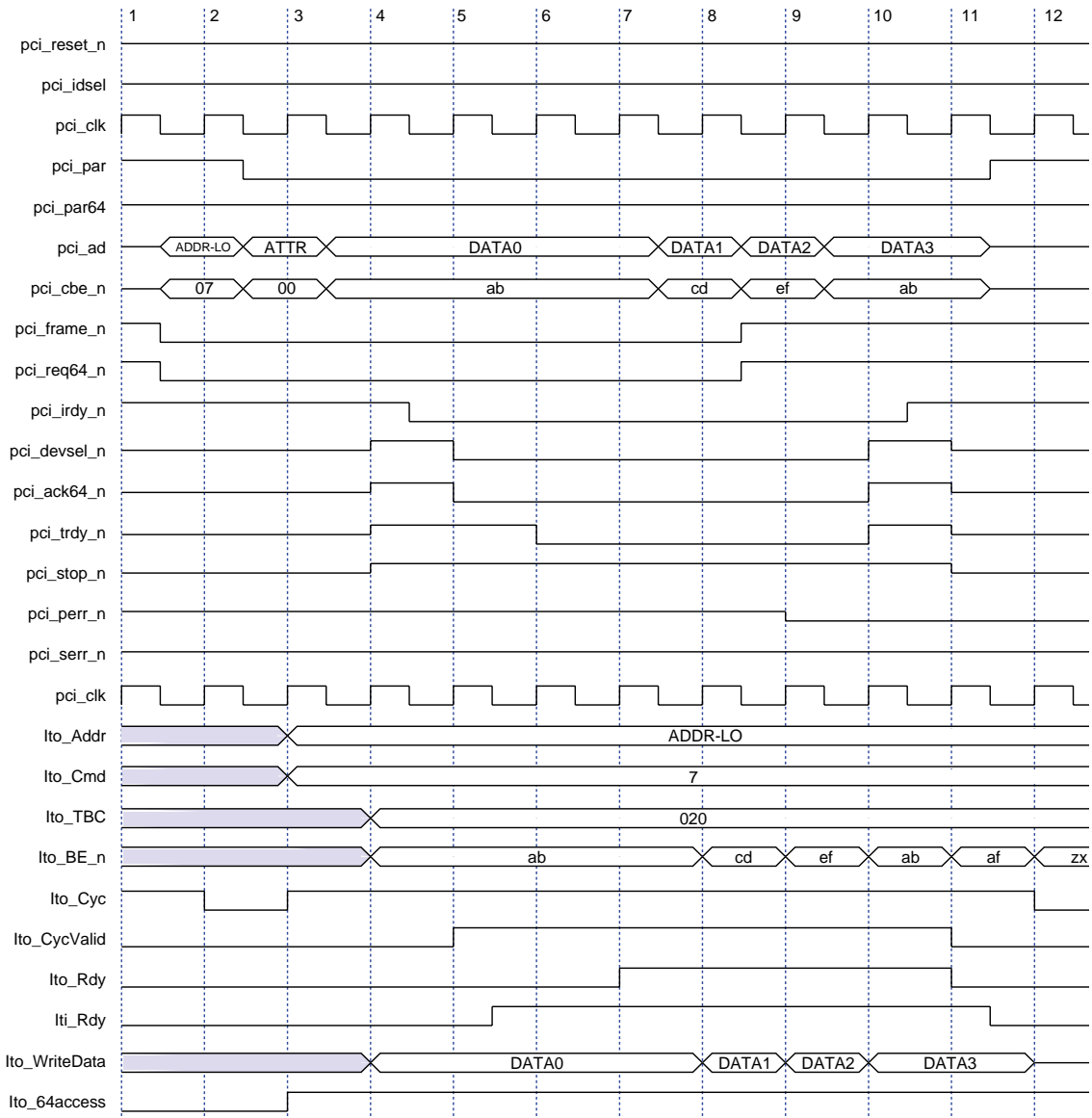
Figure 16. PCI-X 64-Bit Burst Target Memory Write Transaction with Local Side Wait-States



### PCI-X 64-Bit Burst Target Memory Write Transaction with Byte Enables (PCI-X Command 7)

Figure 17 is the same as Figure 15, except that in Figure 17 the PCI-X command is 7, which by definition allows the PCI-X master device to set byte enables for each data phase.

Figure 17. PCI-X 64-Bit Burst Target Memory Write Transaction with Byte Enables (PCI-X Command 7)

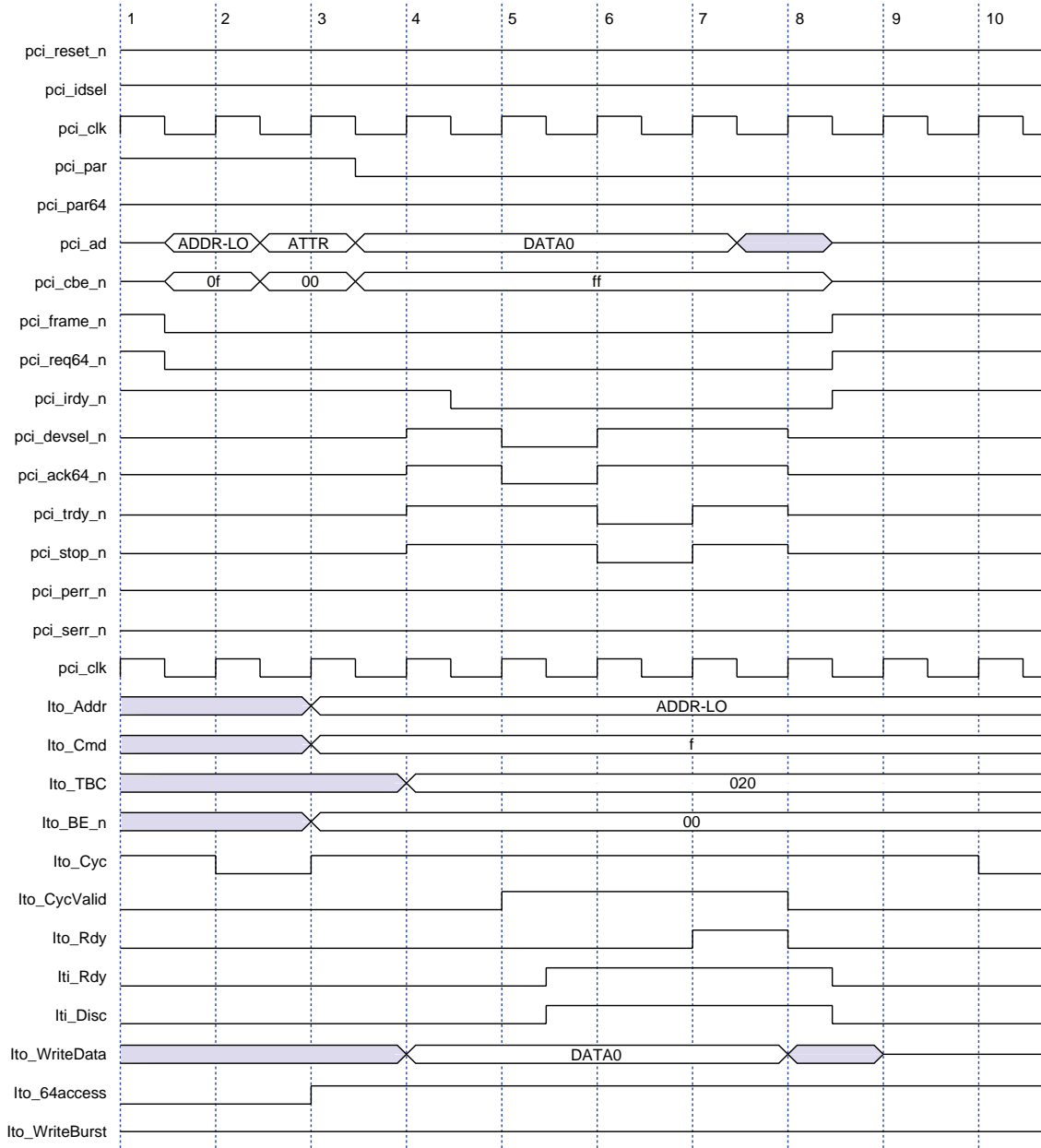


### PCI-X 64-Bit Target Memory Write Transaction with Single Data Phase Disconnect

Figure 18 shows the same transaction as in Figure 14, except with the local side indicating a single data phase disconnect. In Figure 18, the PCI-X master device is requesting to transfer four Qwords with `lto_TBC` set to 20 hexadecimal. However, the local target can only accept one Qword and therefore, asserts `lti_Disc` along with `lti_Rdy` in clock 5 to indicate a single data phase disconnect.



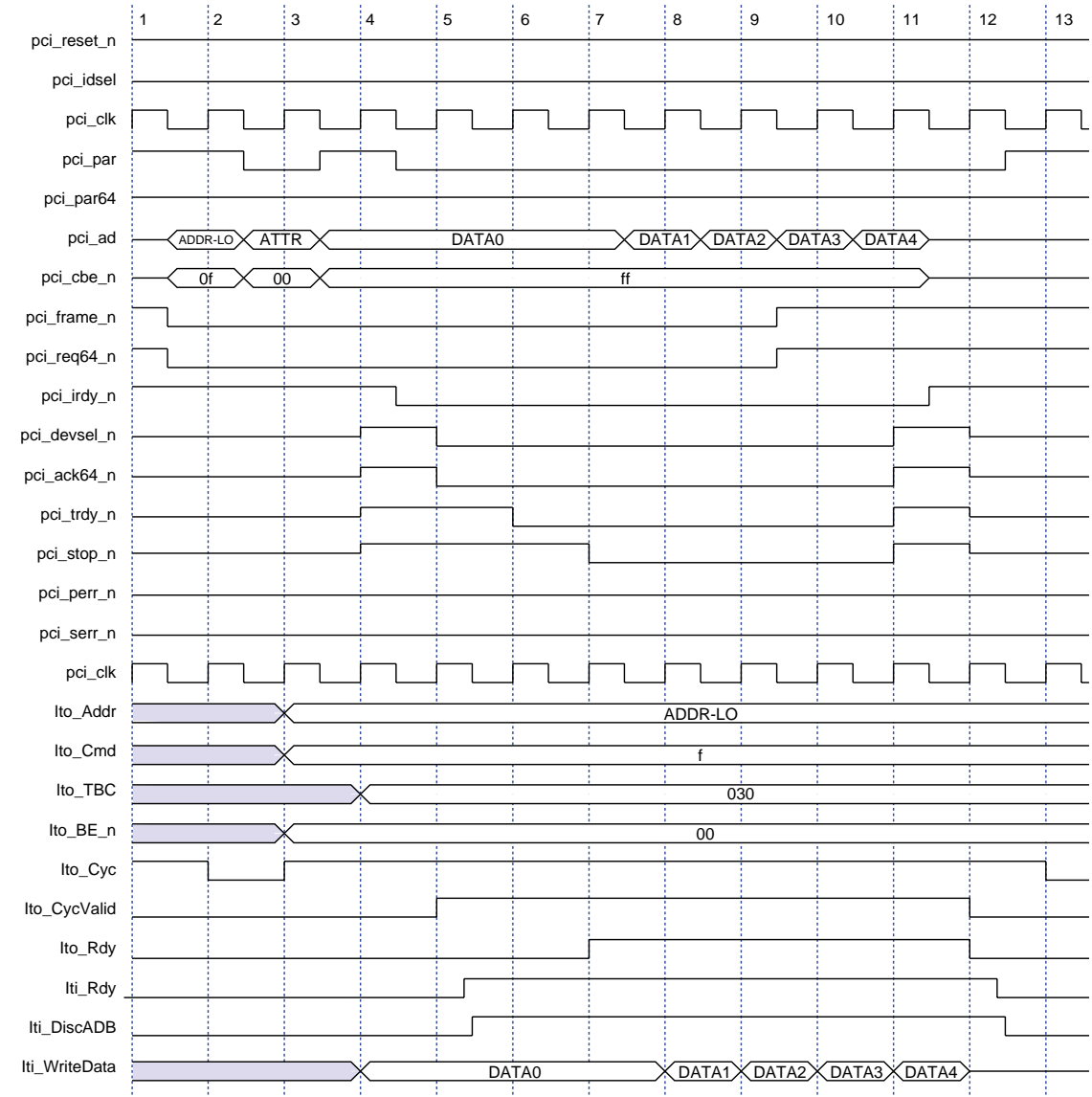
Figure 18. PCI-X 64-Bit Target Memory Write Transaction with Single Data Phase Disconnect



### PCI-X 64-Bit Target Memory Write Transaction with Disconnect at ADB

Figure 19 shows the same transaction as Figure 15 with the local target requesting to disconnect at the ADB by asserting `lti_DiscADB`. In Figure 19, the starting address is five Qwords from the ADB, and the PCI-X master is requesting to transfer six Qwords (`lto_TBC = 12'h030`). The local target asserts `lti_DiscADB` on the same clock cycle as `lti_Rdy`, i.e., clock 5. As a result, the `pci_x` function transfers one Qword with the assertion of `pci_trdy_n` in clock 6, and in the following 4 clock cycles, the `pci_x` function acknowledges the request to disconnect at the ADB by asserting `pci_trdy_n` and `pci_stop_n`.

Figure 19. PCI-X 64-Bit Target Memory Write Transaction with Disconnect at ADB



In PCI-X protocol, the `pci_x` function responds to three types of 32-bit target write transactions:

- Memory write
- I/O write
- Configuration write

### PCI-X 32-bit Target Memory Write Transactions

Memory transactions are either single-cycle or burst. For memory transactions, the `pci_x` function always assumes a 64-bit local side. The `pci_x` function reads one DWORD at a time if the PCI bus is 32-bits wide and then automatically transfers 64-bit data to the local side.

Figure 20 shows a 32-bit single-cycle target memory write transaction. The sequence of events in Figure 20 is identical to Figure 14, except for the following:

- During the address phase (clock 1), the PCI-X master does not assert `pci_req64_n` to indicate a 32-bit transaction.
- The `pci_x` function does not assert `pci_ack64_n` when it asserts `pci_devsel_n`.
- The local side signal, `lto_64access`, is not asserted to indicate (to the local side) that the current transaction is 32-bits.

Figure 20 shows that the `pci_x` registers a Dword from the PCI-X master device in clock 6, and because the starting address is a high Dword boundary (`pci_ad[2] = 1'b1`), the `pci_x` function transfers the Dword to the local side on `lto_WriteData[63:32]`. This is indicated by `lto_BE_n[7:0] = 8'h0F`, where a value "0" indicates valid byte enables and a value "F" indicates invalid byte enables.

Figure 20. PCI-X 32-Bit Single-Cycle Target Memory Write Transaction

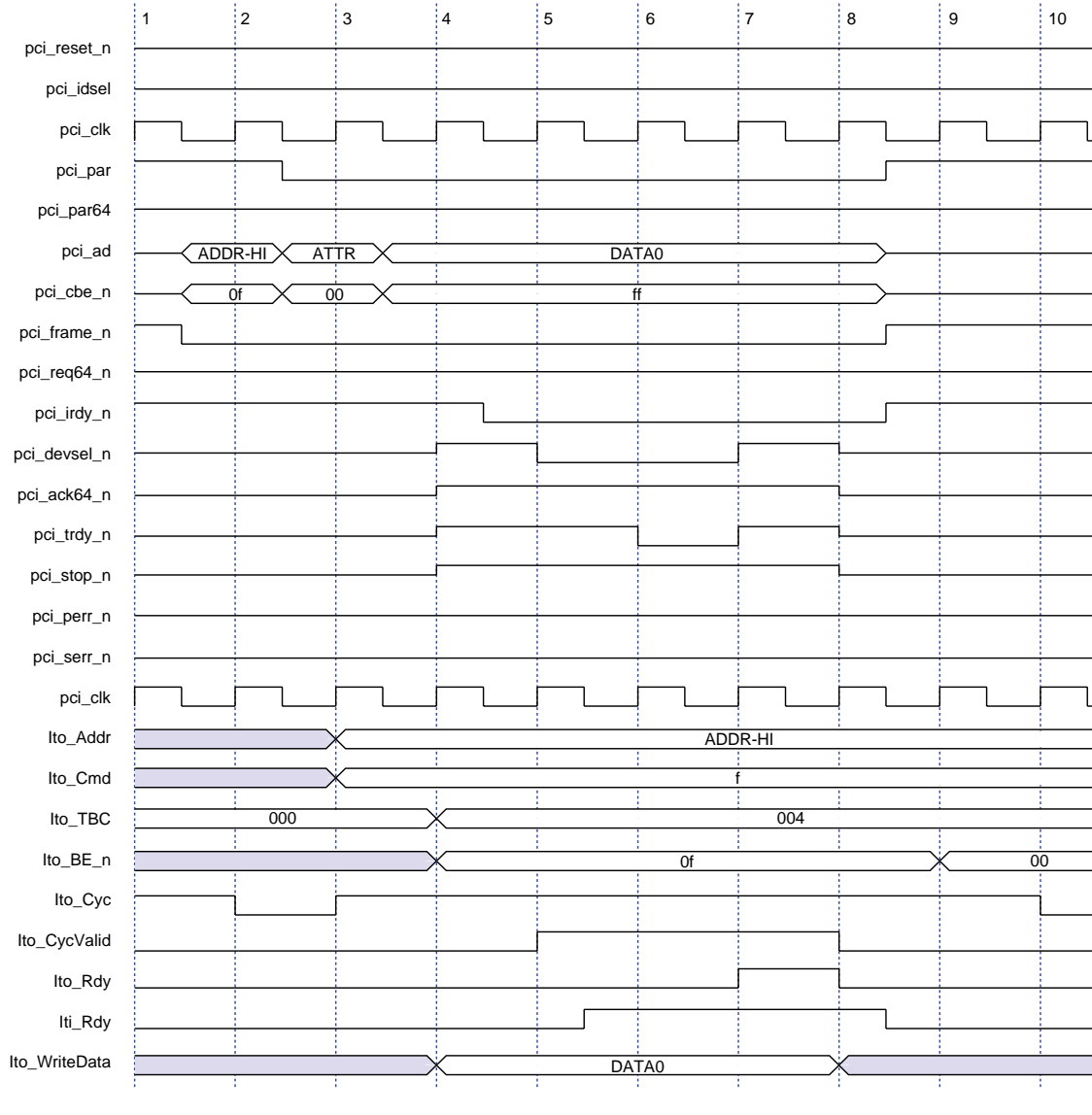
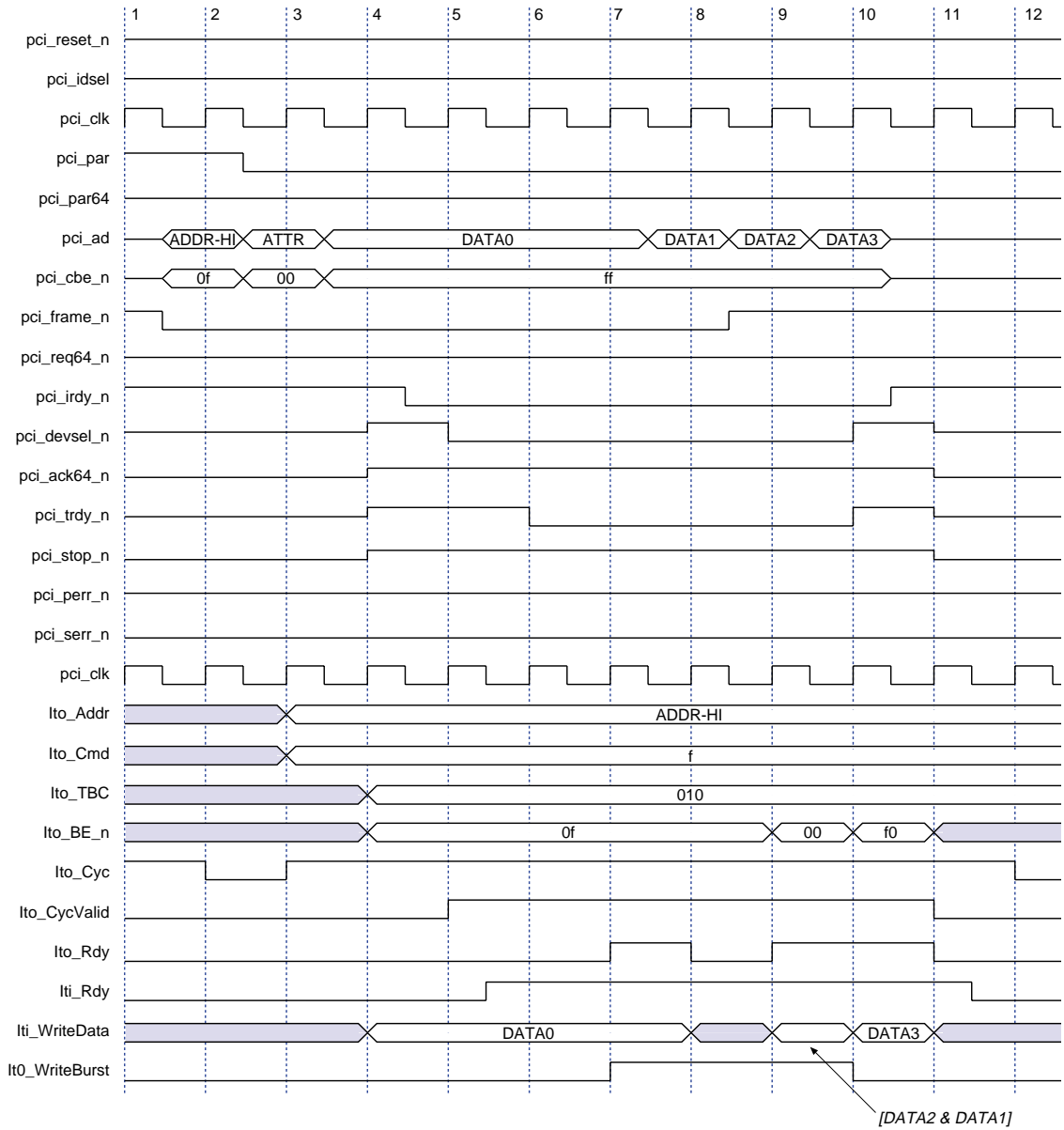


Figure 21 shows a 32-bit burst target memory write transaction. The sequence of events in Figure 21 is exactly the same as in Figure 20, except more than one data phase occurs.

Figure 21 shows that the PCI-X master requests to transfer 16 bytes (`lto_TBC = 12'h010`). Because this is a 32-bit transaction, four data phases are required to complete the byte count. In clock 6, the `pci_x` registers a Dword. In the following clock cycle (clock 7), the `pci_x` function transfers the Dword to the local side on `lto_WriteData[63:32]`. Because the starting address is a high Dword boundary, the data transfer is indicated to the local side with the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'h0F`. In addition, the `pci_x` registers another Dword in clock 7. However, this time the `pci_x` does not transfer the Dword to the local side in the following cycle. Instead, the `pci_x` captures another Dword from the PCI-X master device in clock 8, and then transfers a Qword on `lto_WriteData[63:0]`, which is indicated to the local side with the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'h00` in clock 9. The last Dword is transferred to the local side in clock 10 on `lto_WriteData[31:0]`, which is indicated to the local side with the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'hF0`.

Figure 21. PCI-X 32-Bit Burst Target Memory Write Transaction

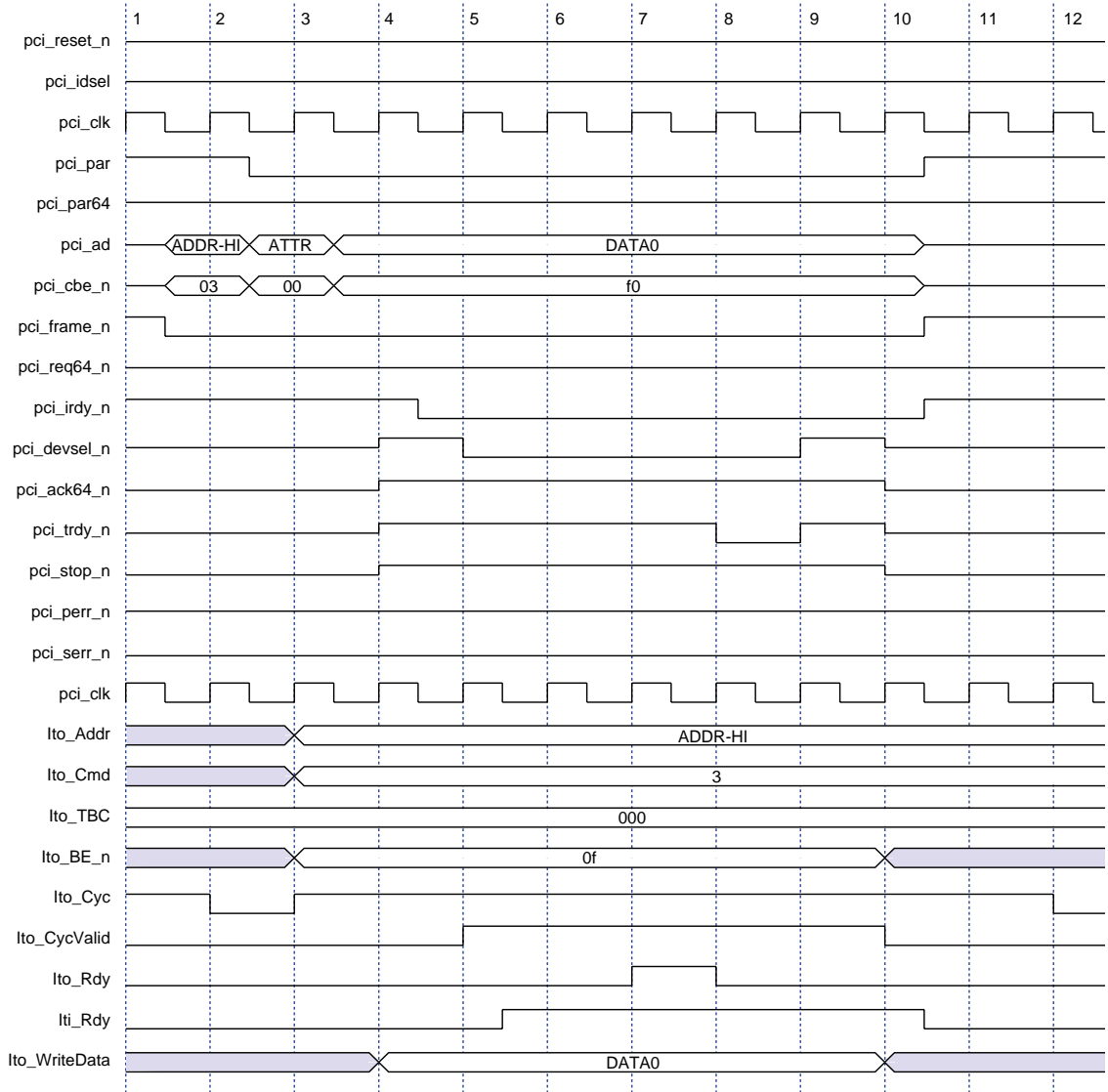


### PCI-X Target I/O Write Transaction

By definition, a PCI-X I/O transaction is 32-bit and single-cycle. [Figure 22](#) shows an I/O write transaction, where bit 2 of the address is 1'b1 (`pci_ad[2] = 1'b1`). The `pci_x` uses the address information to determine where to register the data and the drive the byte enables. Because the I/O data is driven to the upper Dword boundary, the data that appears on `pci_ad[31:0]` also appears on `lto_WriteData[63:32]`. In addition, the byte enables that appear on `pci_cbe_n[3:0]` also appear on `lto_BE_n[7:4]`. The `pci_x` intentionally disables the lower byte enables, i.e., `lto_BE_n[3:0] = 4'hF`.



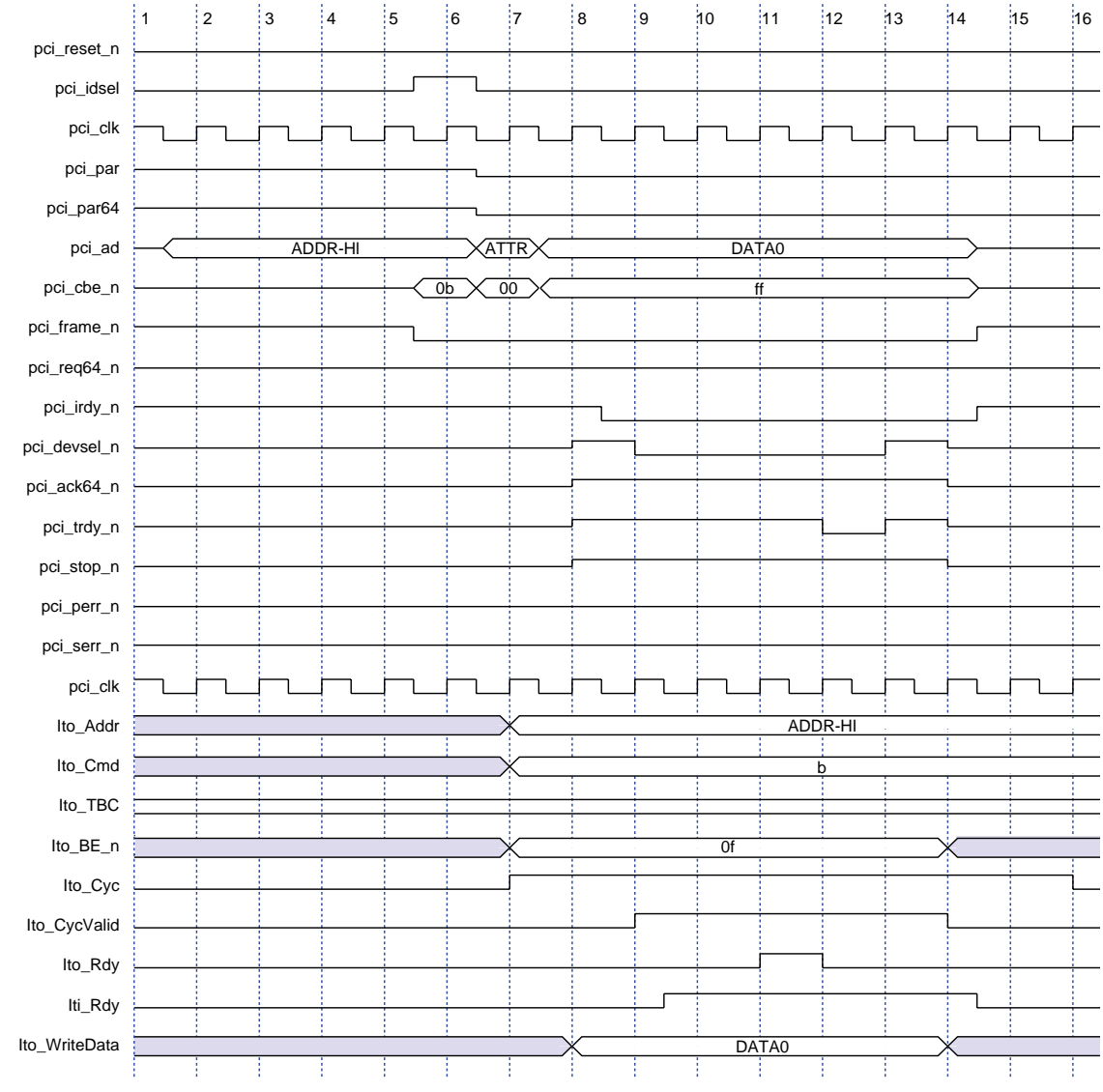
Figure 22. PCI-X I/O Write Transaction



### PCI-X Target Configuration Write Transaction

By definition, a PCI-X configuration transaction is 32-bit and single-cycle. **Figure 23** shows a configuration write transaction, where bit 2 of the address is 1'b1 (`pci_ad[2] = 1'b1`). The `pci_x` uses the address information to determine where to register the data and drive the byte enables. Because the I/O write data was driven to the high Dword boundary, the data that appears on `pci_ad[31:0]` is driven on `lto_WriteData[63:32]`. In addition, the byte enables that appear on `pci_cbe_n[3:0]`, the clock after the address phase, also appear on `lto_BE_n[7:4]`. The `pci_x` intentionally disables the lower byte enables, i.e., `lto_BE_n[3:0] = 4'hF`.

Figure 23. PCI-X Configuration Write Transaction



### PCI-2.2 Target Write Transactions

As stated earlier, from the perspective of the local target bus the behavior of the `pci_x` during PCI-2.2 cycles is similar to the behavior during PCI-X cycles. Also, because the PCI bus can be 32-bit or 64-bit and the local target is always 64-bit, the `pci_x` does some data and byte-enable manipulation to accommodate for the data mismatch.

The following are some general operating rules for PCI-2.2 target write transactions:

- For burst memory write cycles, regardless of whether there is a 32-bit or 64-bit PCI-2.2 master device running the cycle, the `pci_x` will forward the write data to the local target bus in 64-bit increments.
- The address presented to the local bus is the same as the PCI-2.2 address. Therefore, when the `pci_x` is operating in PCI-2.2 protocol, the address presented for a memory write cycle is either Dword- or Qword-aligned, depending on whether the PCI master device is 32-bit or 64-bit. I/O cycles have byte-aligned addresses in PCI-2.2.
- PCI-2.2 Delayed Transaction Support: The `pci_x` does not make any assumptions as to whether the local target is going to accept data immediately during configuration or I/O writes or use the PCI-2.2 Delayed Transaction protocol to forward the write data to the final destination. When a local target retries the `pci_x` during an I/O or configuration write cycle, the `pci_x` does not know if any cycle has been queued or not. Local targets using the PCI-2.2 Delayed Transaction protocol are responsible for storing the relevant transaction information. However the `pci_x` will, throughout the assertion of `lto_Cyc`, maintain stable values for the address (`lto_Addr`), command (`lto_Cmd`), and 64-bit PCI master information (`lto_64access`) in order to assist in storing the transaction information.
- For configuration and I/O write cycles, the `pci_x` will drive data and parity information on the local bus even if it has not received a response from the local target. Local targets that are using the PCI-2.2 Delayed Transaction protocol can then store the relevant data and parity information.

In PCI-2.2 protocol, the `pci_x` supports two types of 64-bit write 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 `pci_frame_n` and `pci_req64_n` signals and drives the address and command on the `pci_ad[31:0]` and `pci_cbe_n[3:0]` buses, correspondingly. Asserting the `pci_req64_n` signal indicates to the target device that the master device is requesting a 64-bit data transaction.

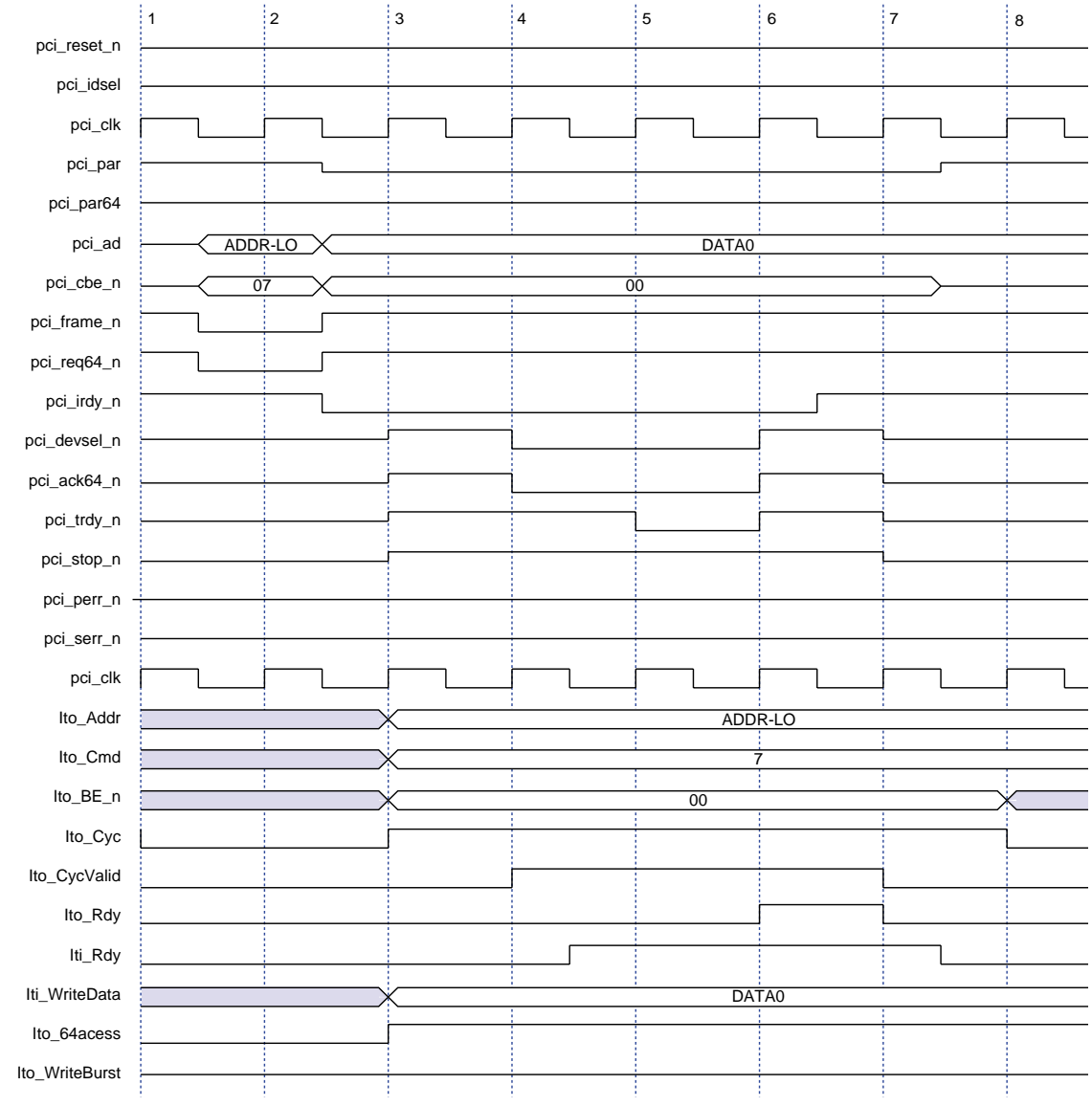
2. If the address of the transactions matches one of the BARs, the `pci_x` function turns on the drivers for the `pci_ad[63:0]`, `pci_devsel_n`, `pci_ack64_n`, `pci_trdy_n`, and `pci_stop_n` signals. The drivers for `pci_par` and `pci_par64` are turned on in the following clock cycle.
3. The `pci_x` function drives and asserts `pci_devsel_n` and `pci_ack64_n` 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.

### PCI-2.2 64-Bit Single-Cycle Target Memory Write Transaction

Figure 24 shows a 64-bit single-cycle target memory write transaction. The sequence of events in Figure 24 is identical to Figure 14, except for the following:

- There is no attribute phase (clock cycle 2)
- If the `pci_x` is the target of a transaction, it will claim the transaction three clocks after the assertion of `pci_frame_n` by asserting `pci_devsel_n`. The `pci_x` is a slow decode device in PCI-2.2 and decode C device in PCI-X.

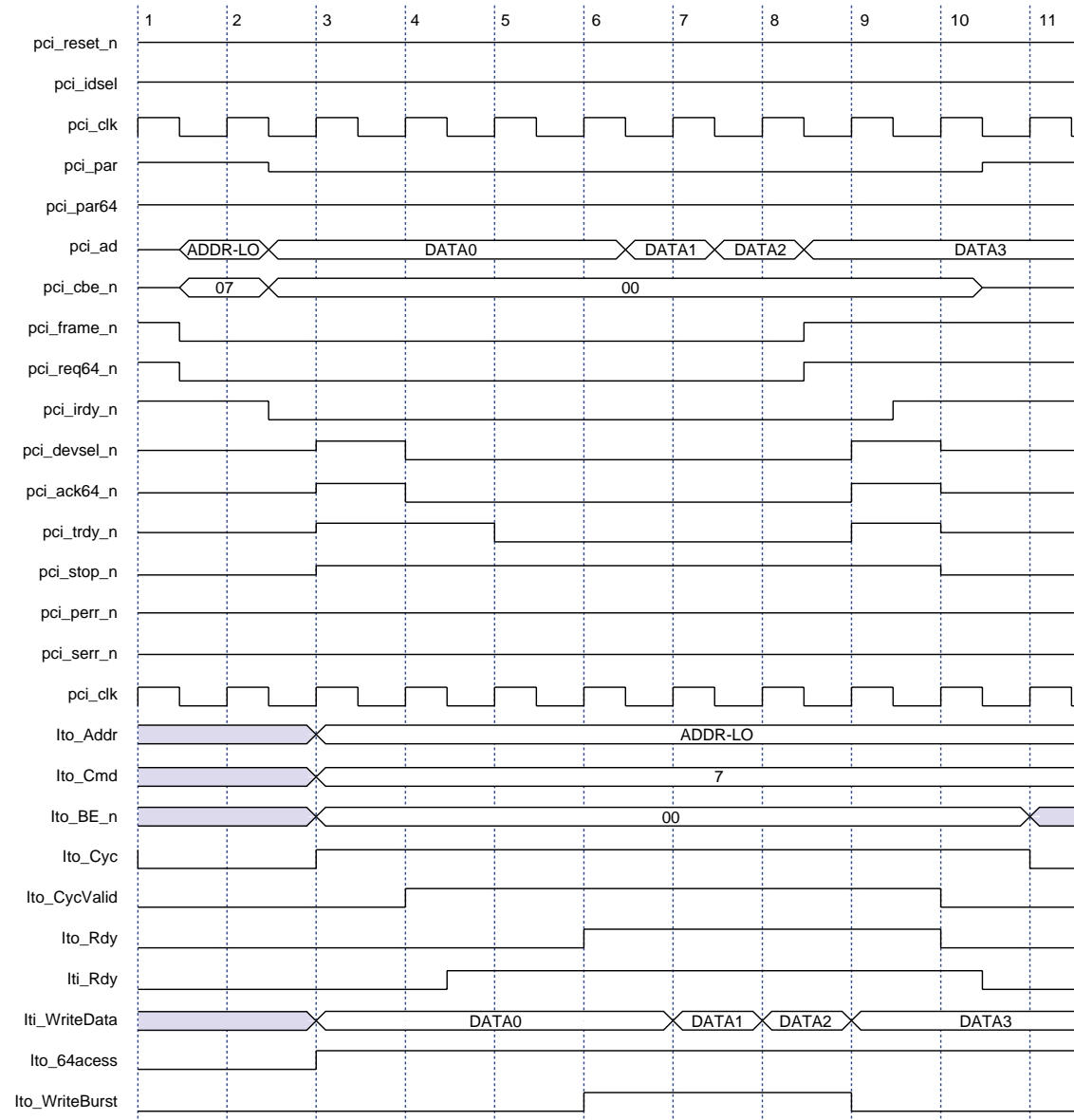
Figure 24. PCI-2.2 64-Bit Single-Cycle Target Memory Write Transaction



PCI-2.2 64-Bit Burst Target Memory Write Transaction

Figure 25 shows a 64-bit burst target memory write transaction. Figure 25 is similar to Figure 24, except that the former transaction is a four Qword data transfer.

Figure 25. PCI-2.2 64-Bit Burst Target Memory Write Transaction



In PCI-2.2 protocol, the `pci_x` function responds to three types of 32-bit target write transactions:

- Memory read
- I/O read
- Configuration read

### PCI-2.2 32-bit Target Memory Write Transactions

Memory transactions are either single- or burst-cycle. For memory transactions, the `pci_x` function always assumes a 64-bit local side. The `pci_x` function registers one Dword at a time if the PCI bus is 32-bit and then automatically transfers 64-bit data increments to the local side.

Figure 26 shows a 32-bit single-cycle target memory write transaction. The sequence of events in Figure 26 is identical to Figure 24, except for the following:

- During the address phase (clock 1), the PCI master does not assert `pci_req64_n` to request a 32-bit transaction.
- The `pci_x` does not assert `pci_ack64_n` when it asserts `pci_devsel_n`.
- The local side signal, `lto_64access`, is not asserted to indicate (to the local side) that the current transaction is 32-bits.

Because the starting address is a high Dword boundary, Figure 26 shows that the `pci_x` function registers a Dword in clock 5 and then in clock 6 transfers the Dword to the local side on the `lto_WriteData[63:32]` bus (indicated by the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'h0F`).



Figure 26. PCI-2.2 32-Bit Single-Cycle Target Memory Write Transaction

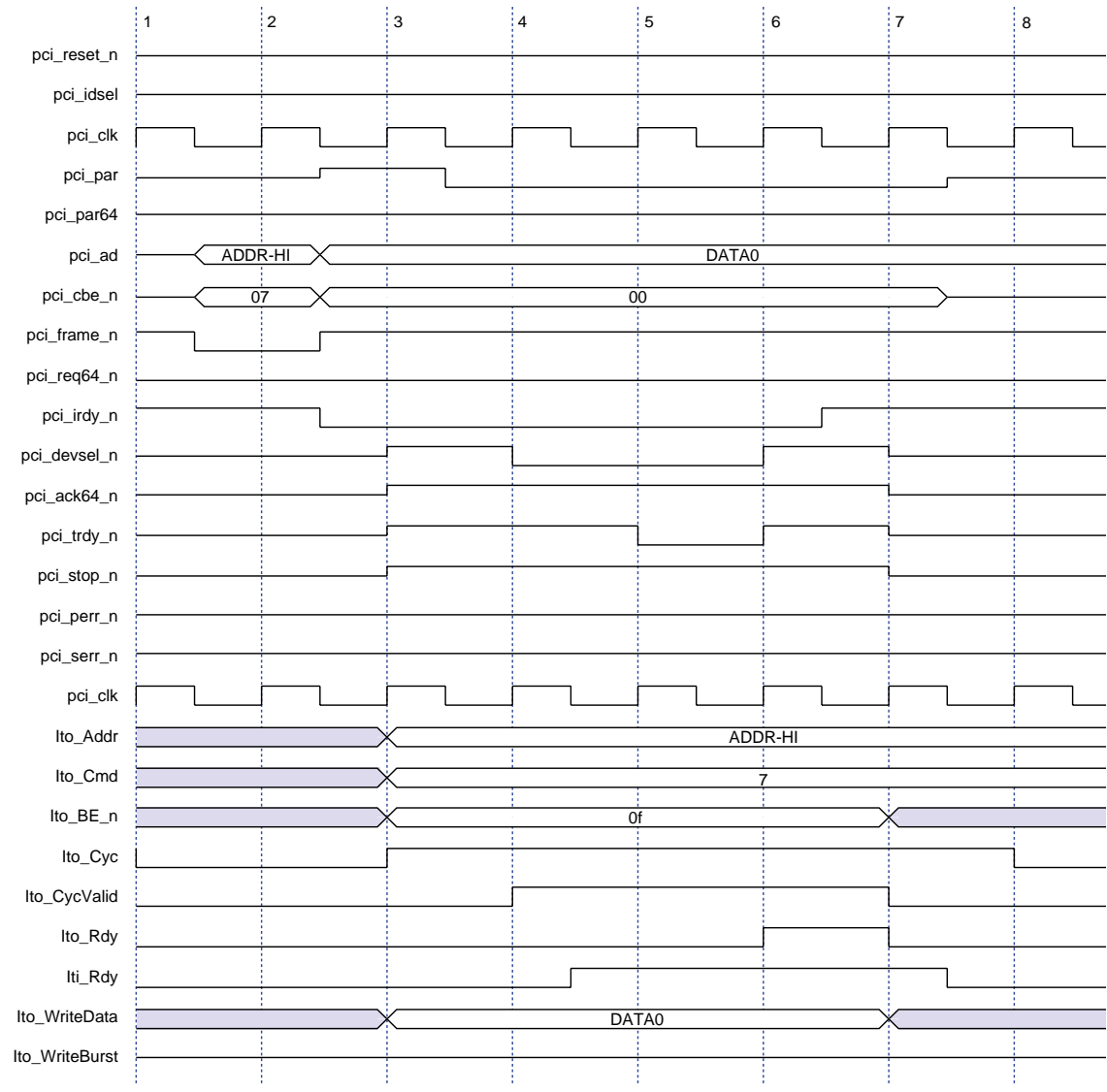
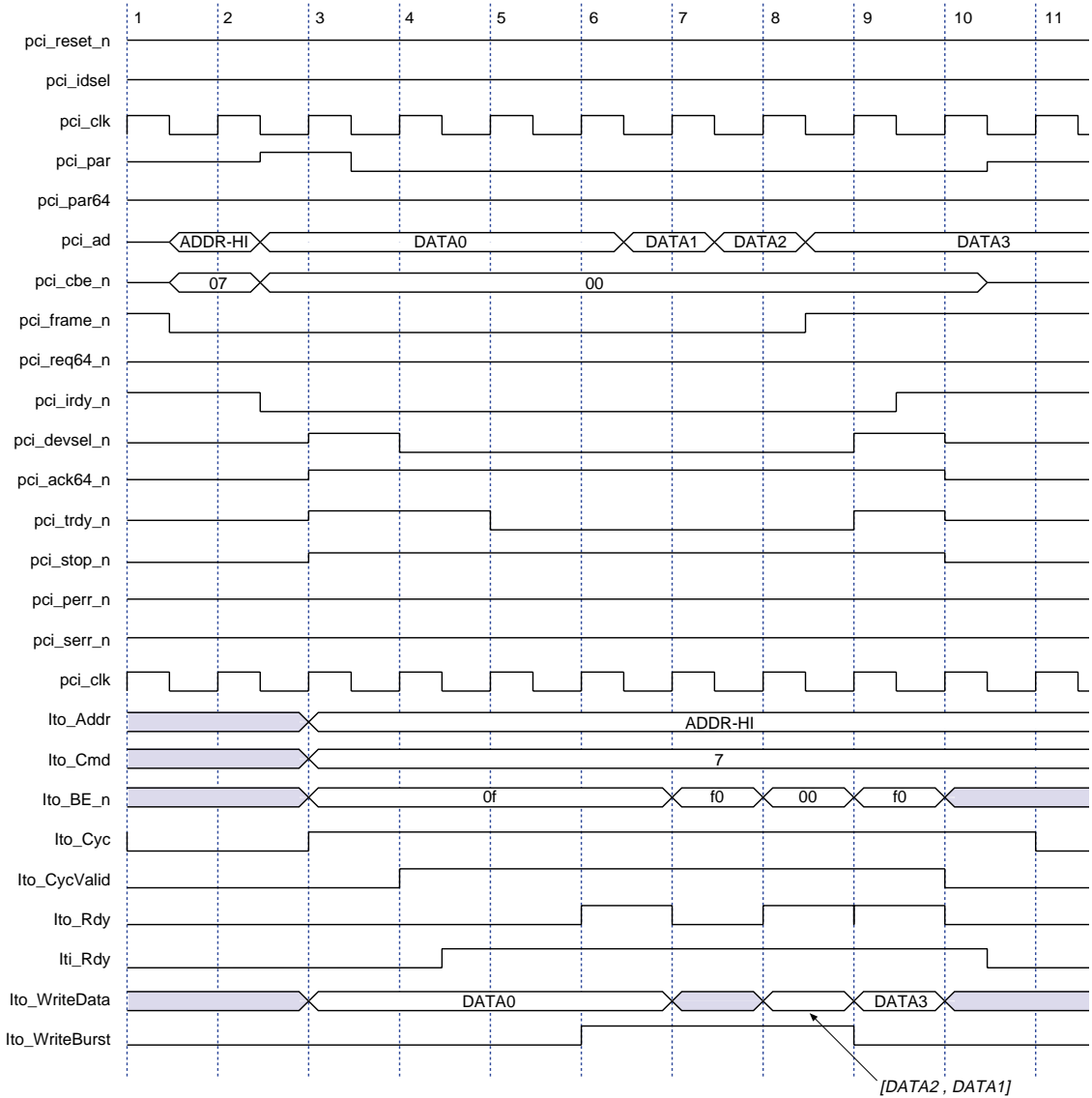


Figure 27 shows a 32-bit burst target memory write transaction. The sequence of events in Figure 27 is identical to Figure 26, except for the multiple data phases. In clock 5, the `pci_x` function registers a Dword. Because the starting address is a high Dword boundary, the `pci_x` transfers the Dword to the local side on the `lto_WriteData[63:32]` bus (indicated to the local side with the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'h0F`) in clock 6. Also in clock 6, the `pci_x` function registers another Dword.

However, the `pci_x` does not transfer the Dword to the local side in clock 7. Instead, the `pci_x` captures another Dword from the PCI-2.2 master device in clock 7, and then transfers a Qword on the `lto_WriteData[63:0]` bus (indicated to the local side with the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'h00`) in clock 8. The last Dword is transferred to the local side in clock 9 on the `lto_WriteData[31:0]` bus (indicated to the local side with the assertion of `lto_Rdy` and `lto_BE_n[7:0] = 8'hF0`).

Figure 27. PCI-32 Bit Burst Target Memory Write Transaction



PCI-2.2 target I/O and configuration write transactions are nearly identical to PCI-X transactions, i.e., PCI-X transactions have an attribute phase. Go to “PCI-X Target I/O Write Transaction” on page 80 and “PCI-X Target Configuration Write Transaction” on page 82 for more information.

## Master Mode Operation

This section describes all supported master transactions for the `pci_x` function and includes waveform diagrams showing typical PCI(X) cycles in master mode. As a master device, the `pci_x` supports all types of PCI-X and PCI-2.2 command types, including special cycle commands.

Because the local master interface does not completely isolate the local master device from potential data-width mismatches, it is necessary for local master devices to be prepared to initiate both Qword and Dword transactions.

Still, the `pci_x` will initiate most transactions with 64-bit data width (Qword), the exceptions are as follows:

- PCI-X protocol: For DWORD commands cycles (e.g., DWORD memory, I/O, and configuration).
- PCI-2.2 protocol: For transactions with a high Dword starting address, `pci_ad[2] = 1'b1` (e.g., I/O, configuration, and memory transactions).
- For master read cycles, the `pci_x` provides the local master with transfer status (`lmo_xferHi/lmo_xferLo`) on a per Dword (32-bit) basis. Local masters use the transfer status information as an indication of when the `pci_x` is providing valid data on the local bus. The transfer status signals are also used to help the local master increment its address. Thus, the master can re-initiate a terminated transaction if it is terminated before the master is able to completely receive all the read data requested.
- For master write cycles, the `pci_x` also provides the local master with transfer status (`lmo_xferHi/lmo_xferLo`) on a per Dword (32-bit) basis. Local masters use the transfer status as an indication of whether the `pci_x` has successfully transferred the data on the PCI(X) bus. The transfer status signals are also used:
  - To determine when the local master must provide the next data phase.
  - To help the local master increment its address so that the local master can re-initiate a terminated transaction if it is terminated before the local master is able to completely send all the data originally intended to write.

### Addressing

Addressing requirements are different for PCI-X and PCI-2.2 protocols. In PCI-X protocol there are less stringent starting address requirements—beyond the byte-alignment requirement. By contrast, in PCI-2.2 protocol the addressing requirements are more stringent. Also, 64-bit addressing is fully supported in PCI-X, whereas in PCI-2.2, 64-bit addressing was an optional feature.

### *PCI-X Protocol Addressing*

In PCI-X protocol, although all addressing must be byte-aligned (including memory transactions) the address can have either Qword or Dword-granularity. The `pci_x` will initiate all burst-type transactions as 64-bit transactions, and all DWORD-type transactions as single data phase 32-bit transactions on the PCI-X bus—regardless of the value of `lmi_DPhaseCnt`.

### *PCI-2.2 Protocol Addressing*

For PCI-2.2 protocol, the `pci_x` requires that local the master issue all memory requests with addresses that have Qword granularity, i.e., `lmi_Addr[2:0] = 3'b000`. Exceptions to this requirement are noted in “In PCI-2.2 Protocol When The Local Master Device Must Initiate a Cycle With High Dword Address”.

For all memory transactions that have an `lmi_DPhaseCnt` greater than one and if `lmi_Addr[2] = 0`, the `pci_x` will initiate the transactions as 64-bit. However, for all memory transactions that have an `lmi_DPhaseCnt` greater than one and if `lmi_Addr[2] = 1` (i.e., a high Dword address), the `pci_x` will initiate the transactions as 32-bit. If the local master wants to transfer a large amount of data, the second approach is not very efficient. Also, the `pci_x` uses bit 2 of the address to accommodate a special case of PCI target termination (See “In PCI-2.2 Protocol When The Local Master Device Must Initiate a Cycle With High Dword Address”).

All other cycle types can be issued with addresses that have byte granularity.

### **In PCI-2.2 Protocol When The Local Master Device Must Initiate a Cycle With High Dword Address**

Under normal operating conditions, the `pci_x` always initiates memory transactions as 64-bit when the `lmi_DPhaseCnt` is greater than one. However, there is a potential livelock condition, (See PCI SIG’s *PCI Local Bus Specification, Rev. 2.2*), where a 32-bit PCI non-bursting target can infinitely retry a 64-bit PCI master device. To avoid this problem, there is a condition on the local master bus, following which the local master must initiate a cycle with a high Dword address (`lmi_Addr[2] = 1'b1`). The algorithm is described below:

If the local master initiates a transaction of the following type:

- `lmi_Cmd` is a memory command

- `lmi_DPhaseCnt` is greater than one
- Address is Qword aligned (i.e., `lmi_Addr[2] = 1'b0`)
- `lmi_BE_[7:0] = 8'hxx`, all 8 bits are whatever the local master specifies

Then the `pci_x` will initiate a 64-bit transaction.

If the `pci_x` terminates the cycle by asserting `lmo_XferLo`, `lmo_Retry`, and `lmo_CycDone`, then the local master **must** initiate the retried cycle with the following parameters:

- `lmi_Cmd` is a memory command
- `i2tiDPhaseCnt` is greater than one
- The address should be a high Dword address, i.e., the address is now Dword-aligned and `lmi_Addr[2] = 1'b1`
- Byte enables should be `lmi_BE_[7:0] = 8'hxF`, where the lower 4 bits are 4'hF but the upper 4 bits are whatever the local master specifies

At this point, the `pci_x` will:

- Initiate a 32-bit transaction, (i.e., not assert `pci_req64_n`),
- Assert `lmo_XferLo` and `lmo_XferHi`. However, the first `lmo_XferLo` must be ignored by the local master consistent with `lmi_BE_[7:0]` being 8'hxF—i.e., the lower 4 byte enables are not enabled.

### *64-bit Addressing*

The `pci_x` supports full 64-bit addressing for all local to PCI(X) cycles, which is a new feature in PCI-X. If `l1ti_Addr[63:0]` has an address greater than 4 GB, the `pci_x` will initiate a DAC in both PCI-2.2 and PCI-X protocols regardless of the type of transaction being initiated. The local master device is responsible for making sure that the address being driven meets the PCI(X) requirements, i.e., only memory cycles are allowed to address greater than 4 GB in both the PCI-2.2 and the PCI-X protocols.



The `pci_x` will not filter addresses because filtering may mask operating problems in the local master device.


### Master Read Transactions

The `pci_x` function performs both PCI-X and PCI-2.2 master read transactions. There are a few major differences between the PCI-X and PCI-2.2 master read transactions; however, from the perspective of the local master bus, behavior of the `pci_x` during the transactions is similar.

PCI-X transactions include an attribute phase, block cycle support, and split transaction support. See “Target Read Transactions” on page 38 for more information.

### *PCI-X Master Read Transactions*

The following are some general operating rules for PCI-X master read transactions:

- To isolate the local master device from potential data-width mismatch with local target devices, the `pci_x` must toggle data. The `pci_x` uses toggle signals (`lmo_XferHi` and `lmo_XferLo`) to qualify whether the data is valid on the upper or lower Dword. For example, the `pci_x` will assert dummy `lmo_XferHi` and `lmo_XferLo` signals if a particular Dword is not transferred because the intended target device is 32-bit and the master device initiating the transaction is 64-bit.
  - For master read block transactions: Because the deassertion of `pci_frame_n` is not an effective transaction termination indicator when master devices have three data phases—or fewer—to transfer, the PCI-X specification requires that master devices modify the byte count to accurately reflect the data phases by:
    - Modifying the value in `lmi_TBC` while initiating memory read block cycles that start three, two, or one Qword(s) from the first ADB.
    - Setting the local master to begin the memory read block cycle when `lmi_DiscADB` is asserted and when the cycle begins three, two or one Qword(s) away from the first ADB. If this option is used, the `pci_x` will generate the byte count up to the first ADB and then drive the byte count—instead of `lmi_TBC`—in the byte count field during the attribute phase.
-  For PCI-X memory read block transactions where the cycle starts greater than three Qwords from the first ADB and the local master asserts `lmi_DiscADB`, the `pci_x` will not need to modify the byte count because in this scenario `pci_frame_n` can be used as a transaction termination indicator.
- If a memory read block request starts three Qwords—or fewer—from the ADB, and `lmi_DiscADB` is asserted, the `pci_x` will generate the byte count up to the first ADB when initiating the cycle on the PCI-X bus. Thus, if the target on the PCI-X bus splits the cycle, only the data bytes from the starting address to the ADB will be returned during the split-completion transaction.

The local master is responsible for issuing the remainder of the transaction (if it wants to). Typically, memory read block cycles

should not be initiated if there is insufficient buffer space for storing the requested read data. However, master devices implementing this protocol should be aware of this corner condition.

In PCI-X protocol, the `pci_x` supports two types of 64-bit read transactions:

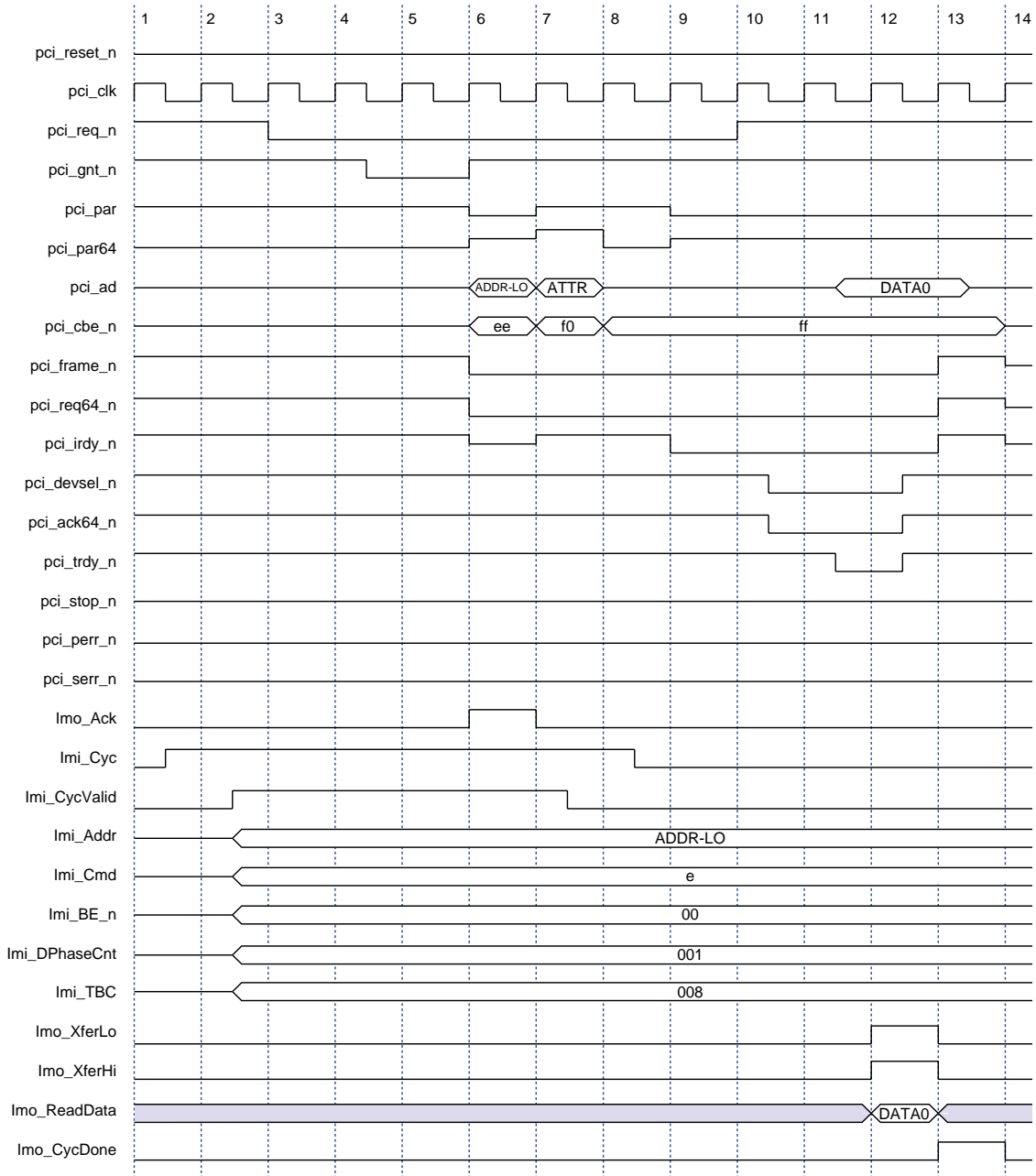
- Memory single-cycle read
- Memory burst read

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

Figure 28 shows the waveform for a PCI-X 64-bit single-cycle master memory read transaction.



Figure 28. PCI-X 64-Bit Single-Cycle Master Memory Read Transaction



**Table 4** shows the sequence of events for a PCI-X 64-bit single-cycle master memory read transaction

<i>Table 4 .PCI-X 64-Bit Single-Cycle Master Memory Read Transaction (Part 1 of 2)</i>	
Clock Cycle	Event
1	Local side: The local master asserts <code>lmi_Cyc</code> to request ownership of the PCI-X bus.
2	Local side: The local master asserts <code>lmi_CycValid</code> to indicate valid local address ( <code>lmi_Addr[63:0]</code> ), command ( <code>lmi_Cmd[3:0]</code> ), byte enables ( <code>lmi_BE_n[7:0]</code> ), data phase count ( <code>lmi_DPhaseCnt[9:0]</code> ), and total byte count ( <code>lmi_TBC[11:0]</code> ). In addition (not shown in the waveforms), the local master provides bus number ( <code>lmi_BusNo[7:0]</code> ), device number ( <code>lmi_DeviceNo[4:0]</code> ), function number ( <code>lmi_FunctionNo[2:0]</code> ), and tag number ( <code>lmi_TagNo[4:0]</code> ) information.
3	PCI-X side: With the assertion of <code>lmi_Cyc</code> in clock cycle 1, the <code>pci_x</code> function asserts <code>pci_req_n</code> to request ownership of the PCI bus. Local side: The local master continues to drive the same information as in clock 2.
4	PCI-X side: The arbiter asserts <code>pci_gnt_n</code> to indicate bus ownership has been granted. However, the arbiter may not always assert <code>pci_gnt_n</code> one clock after receiving <code>pci_req_n</code> .
5	PCI-X side: The arbiter continues to assert <code>pci_gnt_n</code> . Local side: The local master continues to drive the same information as in clock 2.
6	PCI-X side: The address phase. The <code>pci_x</code> asserts <code>pci_frame_n</code> to indicate the beginning of the transaction. The <code>pci_req64_n</code> is also asserted to indicate that a 64-bit transaction is being requested. Local side: The <code>pci_x</code> function drives <code>lmo_Ack</code> to indicate to the local side that the PCI-X bus has been granted to the local master.
7	PCI-X side: The attribute phase. The additional information provided by the local master in clock 2 is driven on <code>pci_ad[31:0]</code> and <code>pci_cbe_n[3:0]</code> . In this transaction, the total byte count to be transferred is eight, which is equivalent to one Qword. Local side: Because the attribute phase has taken place on the PCI-X side, the local master deasserts <code>lmi_CycValid</code> .
8	PCI-X side: Turn-around cycle on the <code>pci_ad[63:0]</code> bus. Local side: The local master deasserts <code>lmi_Cyc</code> (immediately after this transaction) to indicate that it has no intention of requesting ownership of the bus.
9	PCI-X side: Because this is a master read, the target provides the data on <code>pci_ad[63:0]</code> . The <code>pci_x</code> function asserts <code>pci_irdy_n</code> to indicate that it is ready to accept data.
10	PCI-X side: The target claims the 64-bit transaction request by asserting <code>pci_devsel_n</code> and <code>pci_ack64_n</code> . In this particular case, the target is a decode C device.

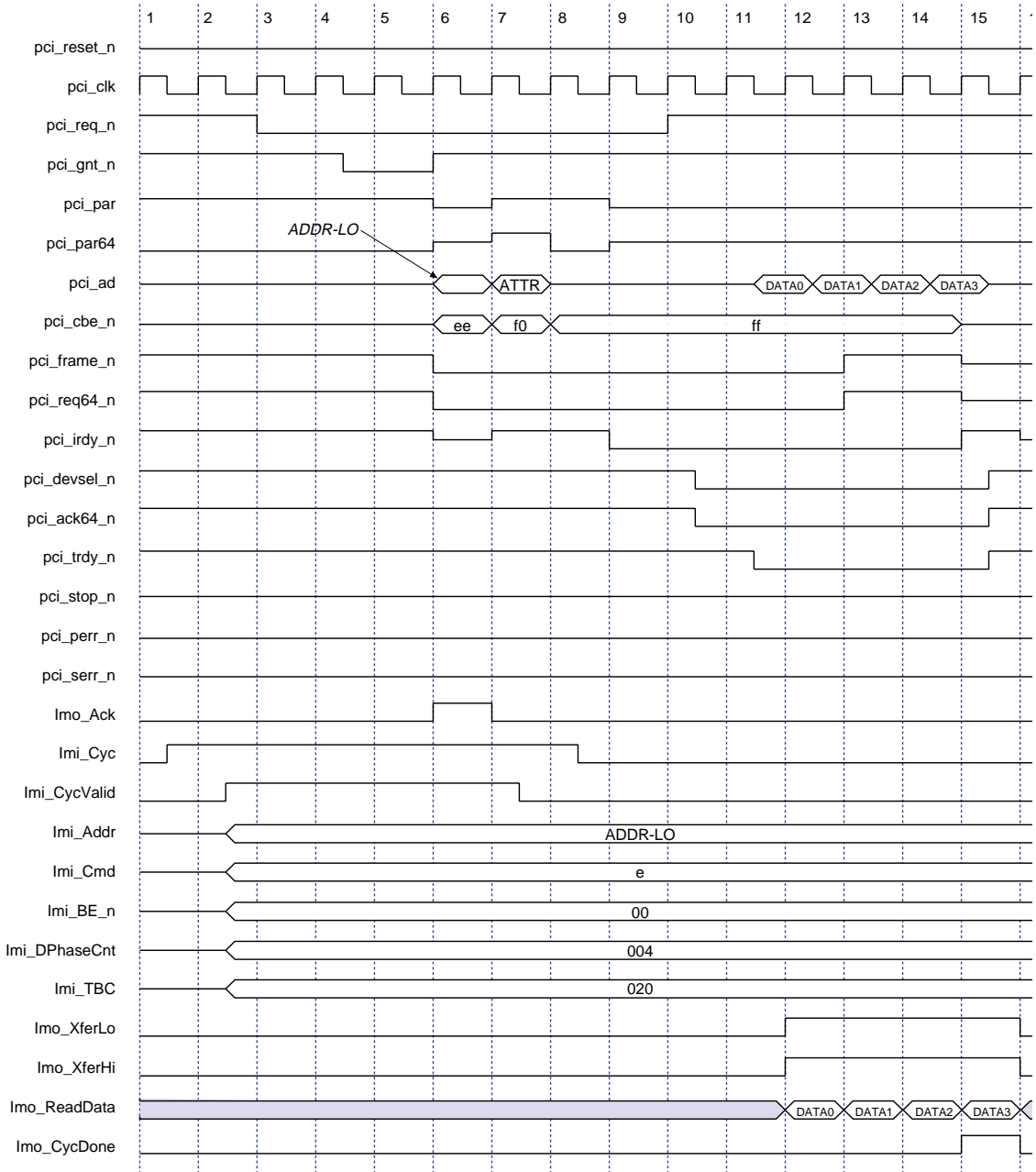
**Table 4 .PCI-X 64-Bit Single-Cycle Master Memory Read Transaction (Part 2 of 2)**

11	PCI-X side: The target asserts <code>pci_trdy_n</code> to indicate that it ready to send data. Data transfer occurs on the PCI-X bus in this clock cycle.
12	PCI-X side: The target deasserts <code>pci_devsel_n</code> , <code>pci_ack64_n</code> , and <code>pci_trdy_n</code> . Local side: The <code>pci_x</code> function transfers the data registered in clock cycle 11 to the local side on <code>lmo_ReadData[63:0]</code> (indicated by the assertion <code>lmo_XferLo</code> and <code>lmo_XferHi</code> ).
13	PCI-X side: The <code>pci_x</code> deasserts <code>pci_frame_n</code> , <code>pci_req64_n</code> , and <code>pci_irdy_n</code> . Local side: The <code>pci_x</code> asserts <code>lmo_CycDone</code> to indicate that the transaction is complete.

### PCI-X 64-Bit Burst Master Memory Read Transaction

Figure 29 shows a PCI-X 64-bit burst master memory read cycle. The sequence of events in Figure 29 is identical to Figure 28, except that more data is transferred. Figure 29 shows a 64-bit zero wait state burst read transaction with four data phases. Four Qwords are transferred from the PCI-X side in clocks 11 through 14 and are then transferred to the local side in clocks 12 through 15.

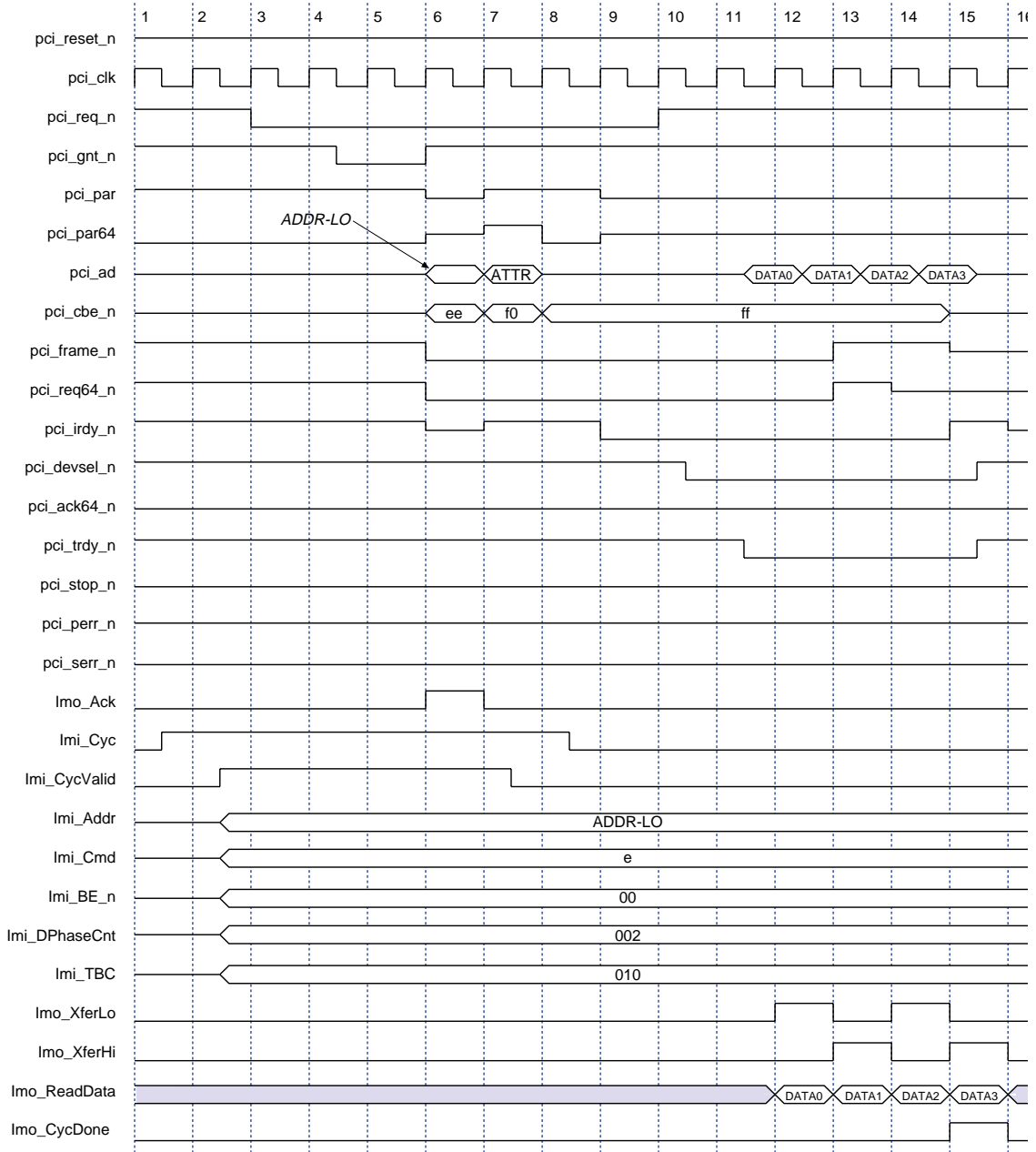
Figure 29. PCI-X 64-Bit Burst Master Memory Read Transaction



### PCI-X 64-Bit Burst Master Memory Read Transaction with a 32-Bit Target

Figure 30 shows the local master device requesting a 64-bit transaction by asserting `pci_req64_n` in clock 6. However, the PCI-X target is only 32-bit (indicated by the non-active state of `pci_ack64_n` in clock 10). Therefore, the `pci_x` function registers four Dwords in clocks 11 through 14, and transfers the data (to the local side) one Dword at a time in clocks 12 through 15. Also, because the starting address is a low Dword boundary (`pci_ad[2] = 1'b0`), the first Dword transferred to the local side is on `lmo_ReadData[31:0]` (indicated with the assertion of `lmo_XferLo`), and the second Dword is transferred on `lmo_ReadData[63:32]` (indicated by the assertion of `lmo_XferHi`). The toggling between `lmo_XferLo` and `lmo_XferHi` indicates that only Dword transfers are occurring on the local side.

Figure 30. PCI-X 64-Bit Burst Master Memory Read Transaction with a 32-Bit Target



In PCI-X protocol, the `pci_x` function can initiate three types of 32-bit master read transactions:

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



Because the `pci_x` is a 64-bit local master device, it does not initiate 32-bit burst master read transactions (using PCI-X protocol).

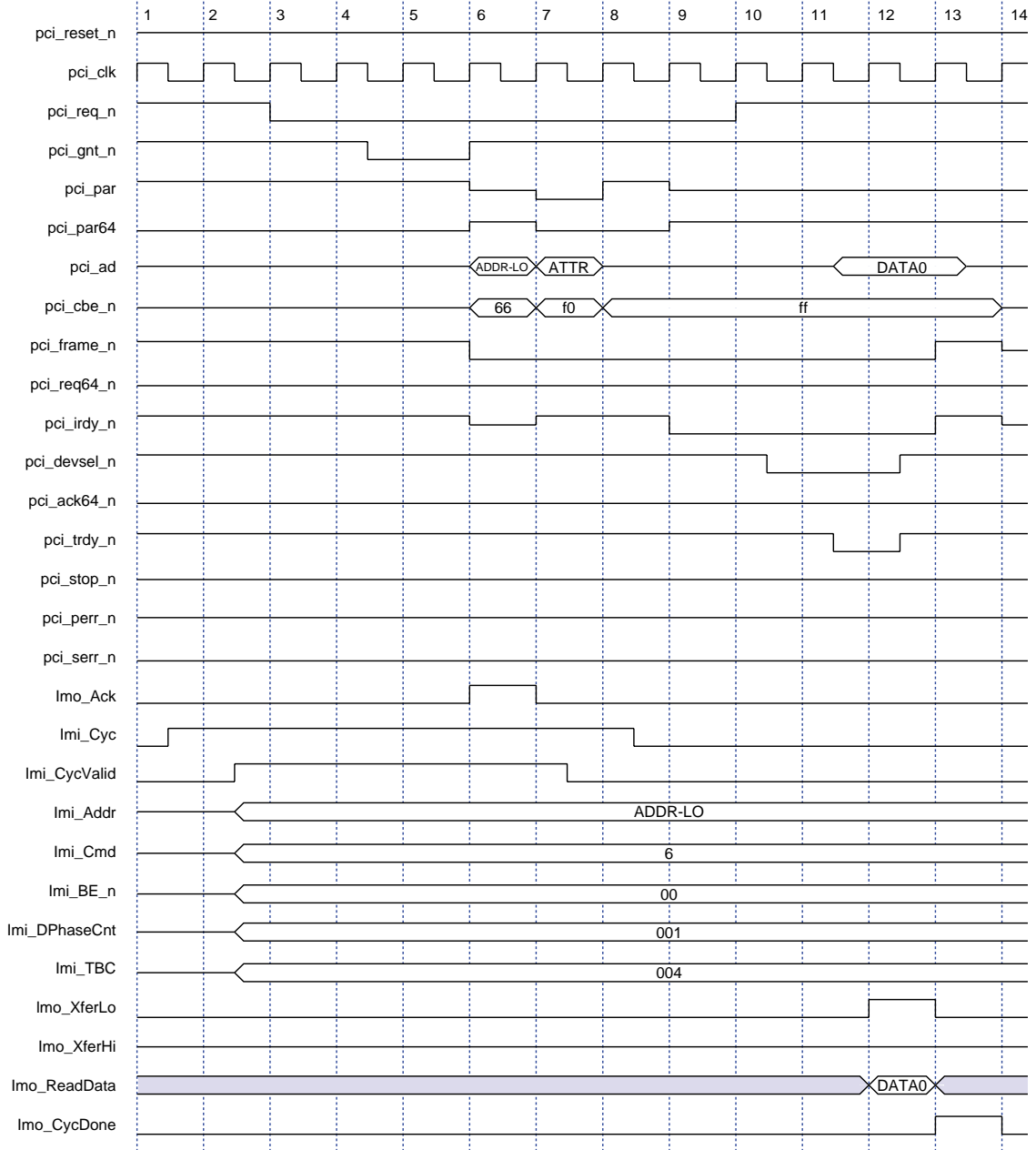
### PCI-X 32-bit Single-Cycle Master Memory Read Transactions

Figure 31 shows a 32-bit single-cycle master memory read transaction. The sequence of events in Figure 31 is identical to Figure 28, except for the following:

- During the address phase (clock 6), the `pci_x` function does not assert `pci_req64_n`. The local master uses the memory read DWORD command (`lmi_Cmd[3:0] = 4'h6`) to specifically request a single-cycle, 32-bit transaction.
- The PCI-X target does not assert `pci_ack64_n` when it asserts `pci_devsel_n`.

Figure 31 shows that the `pci_x` function registers a Dword in clock 11. Also, because the starting address is a low Dword boundary, the `pci_x` function transfers the Dword to the local side in the following clock cycle (clock 12) on `lmo_ReadData[31:0]` (indicated by the assertion of `lmo_XferLo`).

Figure 31. PCI-X 32-Bit Single-Cycle Master Memory Read Transaction

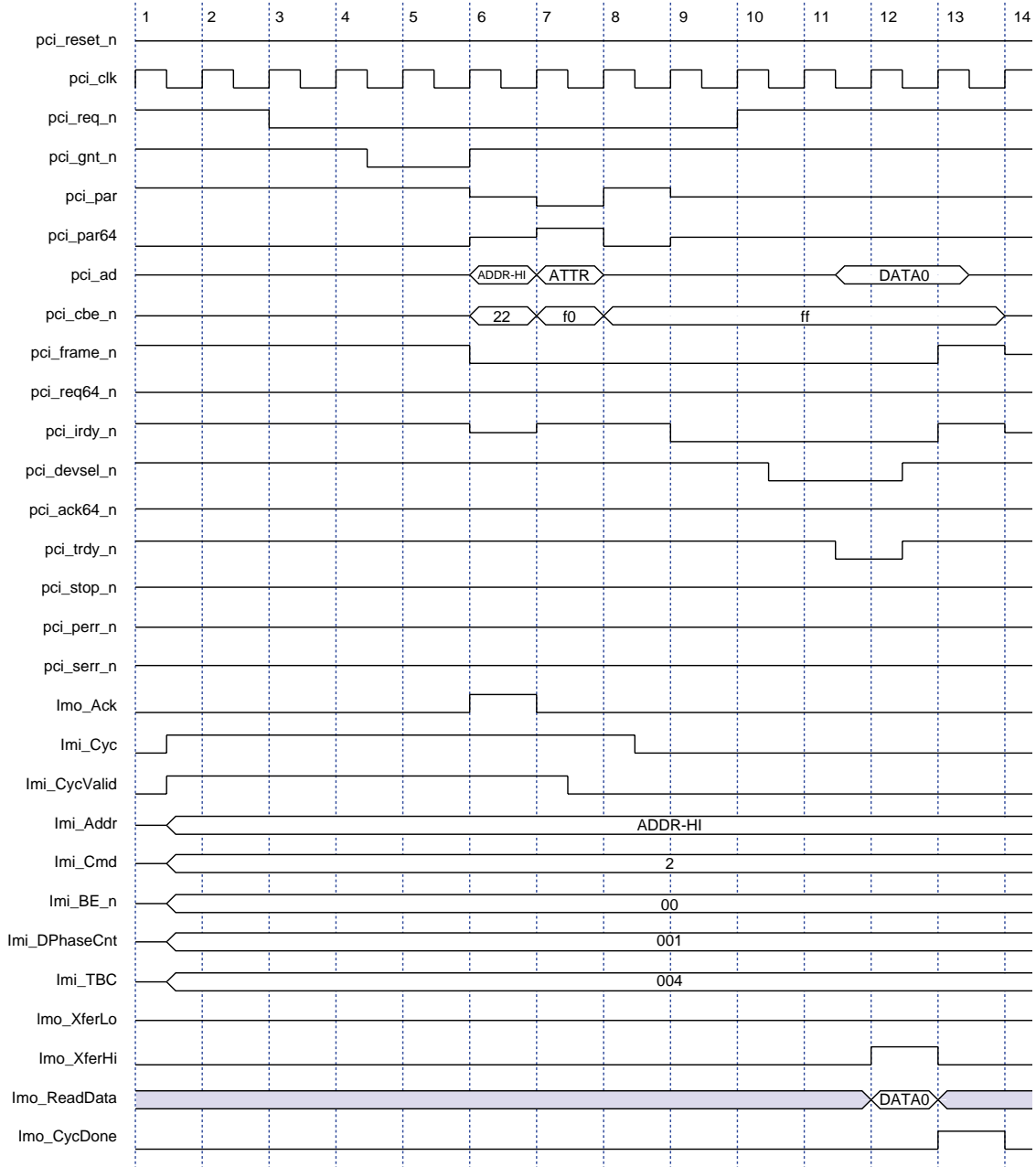




### PCI-X Master I/O Read Transactions

By definition, a PCI-X I/O transaction is 32-bit and single-cycle. [Figure 32](#) shows a master I/O read transaction. The sequence of events in [Figure 32](#) is similar to [Figure 31](#). [Figure 32](#) shows that the `pci_x` function registers a Dword in clock 11, and because the starting address is a high Dword boundary, the `pci_x` function transfers the Dword to the local side on `lmo_ReadData[63:32]` in clock 12 (indicated by the assertion of `lmo_XferHi`).

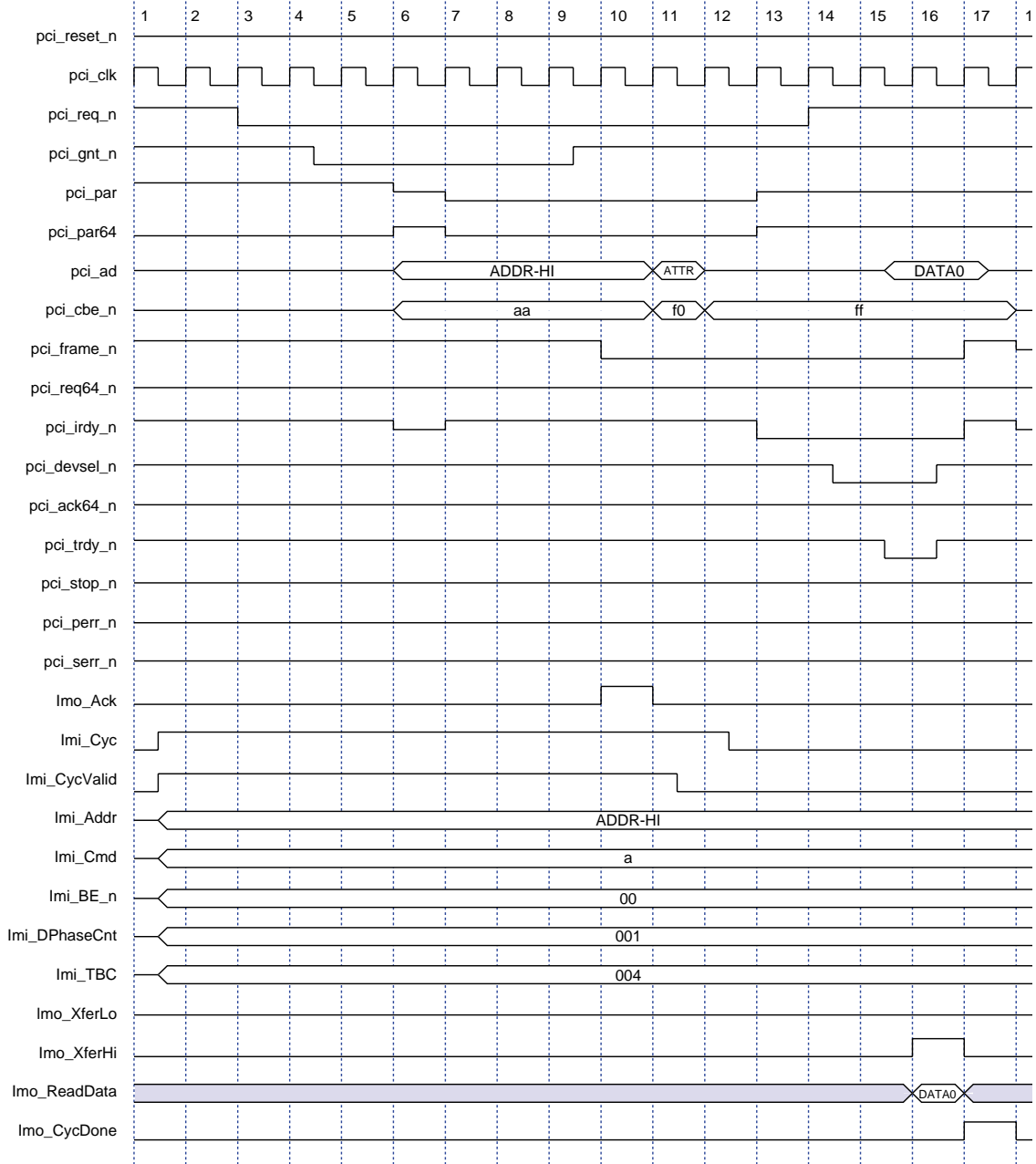
Figure 32. PCI-X Master I/O Read Transaction



### PCI-X Master Configuration Read Transactions

By definition, a PCI-X configuration transaction is 32-bit and single-cycle. Figure 33 shows a master configuration read transaction. The sequence of events in Figure 33 is similar to Figure 31, except that the `pci_x` function performs address stepping by driving the address bus (`pci_ad[63:0]`) four clock cycles before the PCI-X address phase. Figure 33 also shows that the `pci_x` function registers a Dword in clock 15, and because the starting address is a high Dword boundary, the `pci_x` function transfers the Dword to the local side in clock 16 on `lmo_ReadData[63:32]` (indicated by the assertion of `lmo_XferHi`).

Figure 33. PCI-X Master Configuration Read Transactions



### *PCI-2.2 Master Read Transactions*

From the perspective of the local master bus, the behavior of the `pci_x` during PCI-2.2 cycles is similar to the behavior during PCI-X cycles, except for the following:

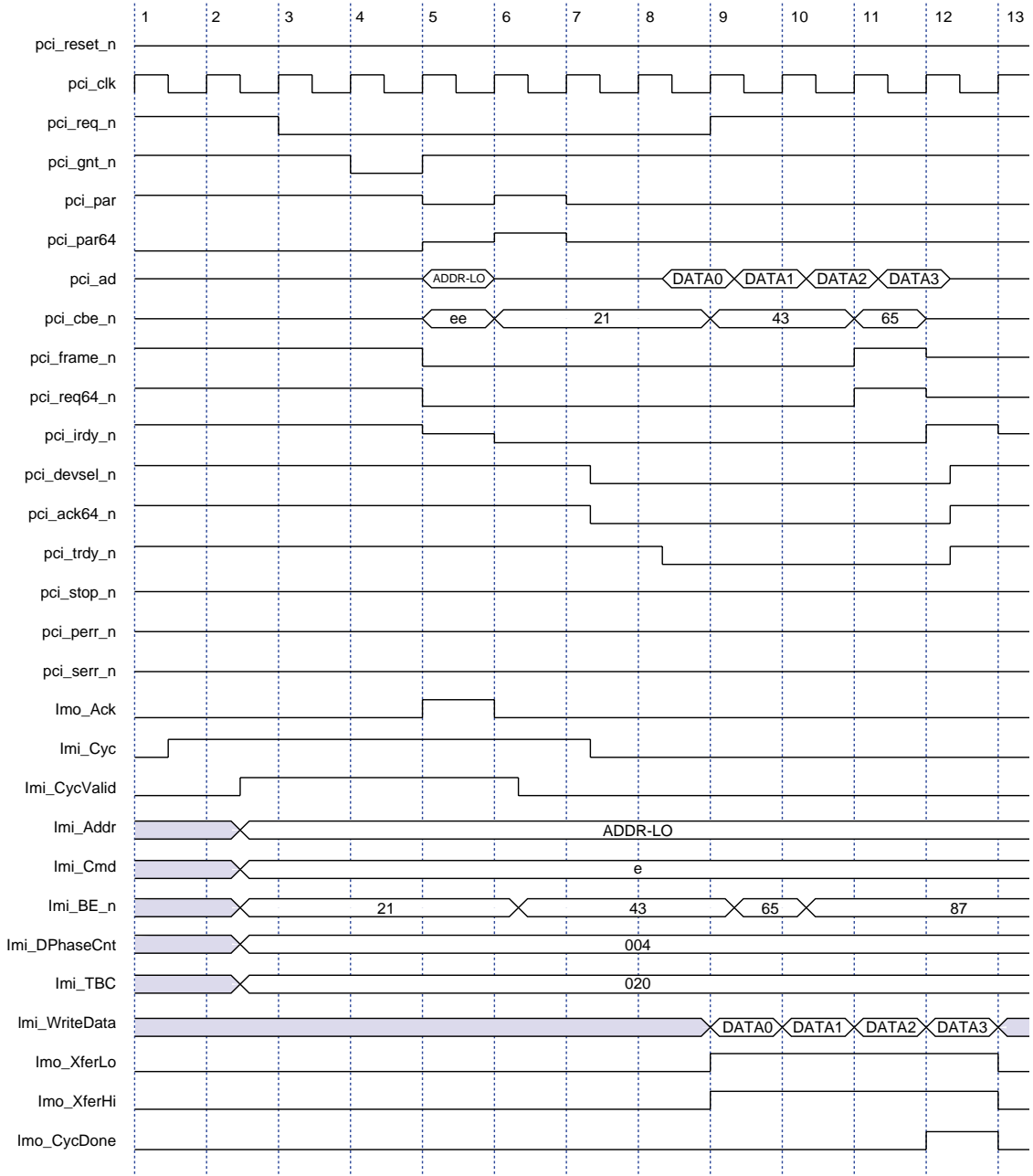
- There is no attribute phase.
- The `pci_x` does not initiate 64-bit single-cycle master read transactions.
- The `pci_x` can initiate a 32-bit burst master memory read transaction if the starting address is a high Dword boundary. However, if the starting address is a low Dword boundary, the `pci_x` function initiates a 64-bit burst transaction.

In PCI-2.2 protocol, the `pci_x` initiates only one type of 64-bit read transactions: Memory burst read.

### **PCI 64-Bit Burst Master Memory Read Transaction**

Figure 34 shows a 64-bit zero wait-state burst master memory read transaction with four data phases. Four Qwords are transferred from the PCI side in clocks 8 through 11, and are then transferred to the local side in clocks 9 through 12.

Figure 34. PCI-2.2 64-Bit Burst Master Memory Read Transaction



In PCI-2.2 protocol, the `pci_x` responds to three types of 32-bit master read transactions:

- Memory read
- I/O read
- Configuration read



The `pci_x` function does support single-cycle and burst memory transactions, but does not support bursting of configuration or I/O cycles.

### PCI-2.2 32-bit Master Memory Read Transactions

Memory transactions are either single-cycle or burst. In PCI-2.2 protocol, the `pci_x` function can initiate a 32-bit single-cycle, depending on `lmi_BE_n[7:0]`. To initiate a 32-bit single-cycle with a low Dword starting address, set `lmi_BE_n[7:0] = 8'Hf0`. To initiate a 32-bit single-cycle with a high Dword starting address, set `lmi_BE_n[7:0] = 8'h0F`. In PCI-2.2 protocol, the `pci_x` function only initiates 32-bit burst master memory read transactions if the starting address is a high Dword boundary.

**Figure 35** shows a 32-bit single-cycle master memory read transaction. In **Figure 35**, the local master initiates a high Dword starting address transaction. In addition, the local master sets `lmi_BE_n = 8'h0F` to indicate a 32-bit single-cycle, where the Dword will be transferred to the local side on `lmo_ReadData[63:32]`. The `pci_x` function drives a dummy `lmo_XferLo` signal.

Figure 35. PCI-2.2 32-Bit Master Memory Read Transaction

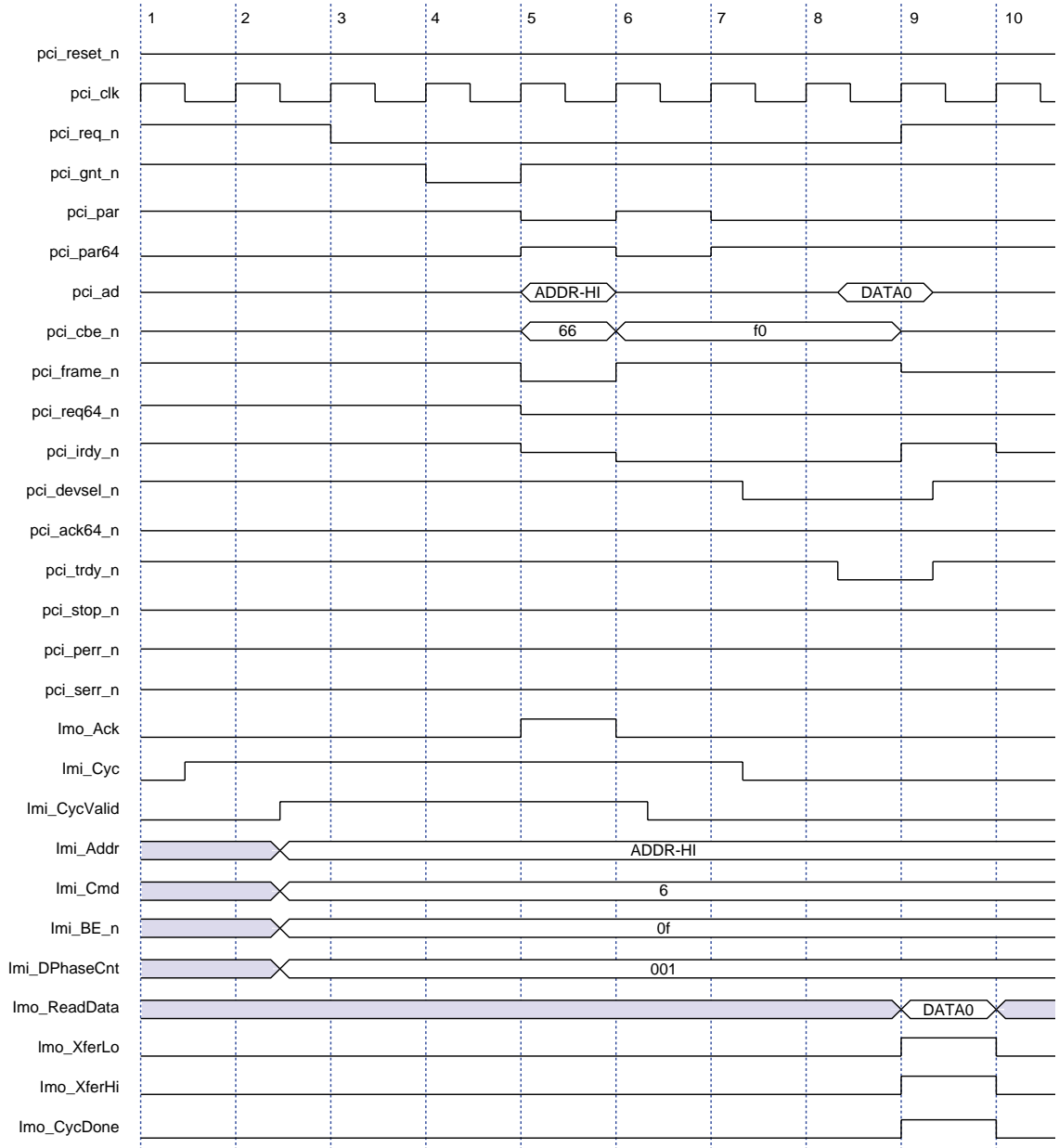
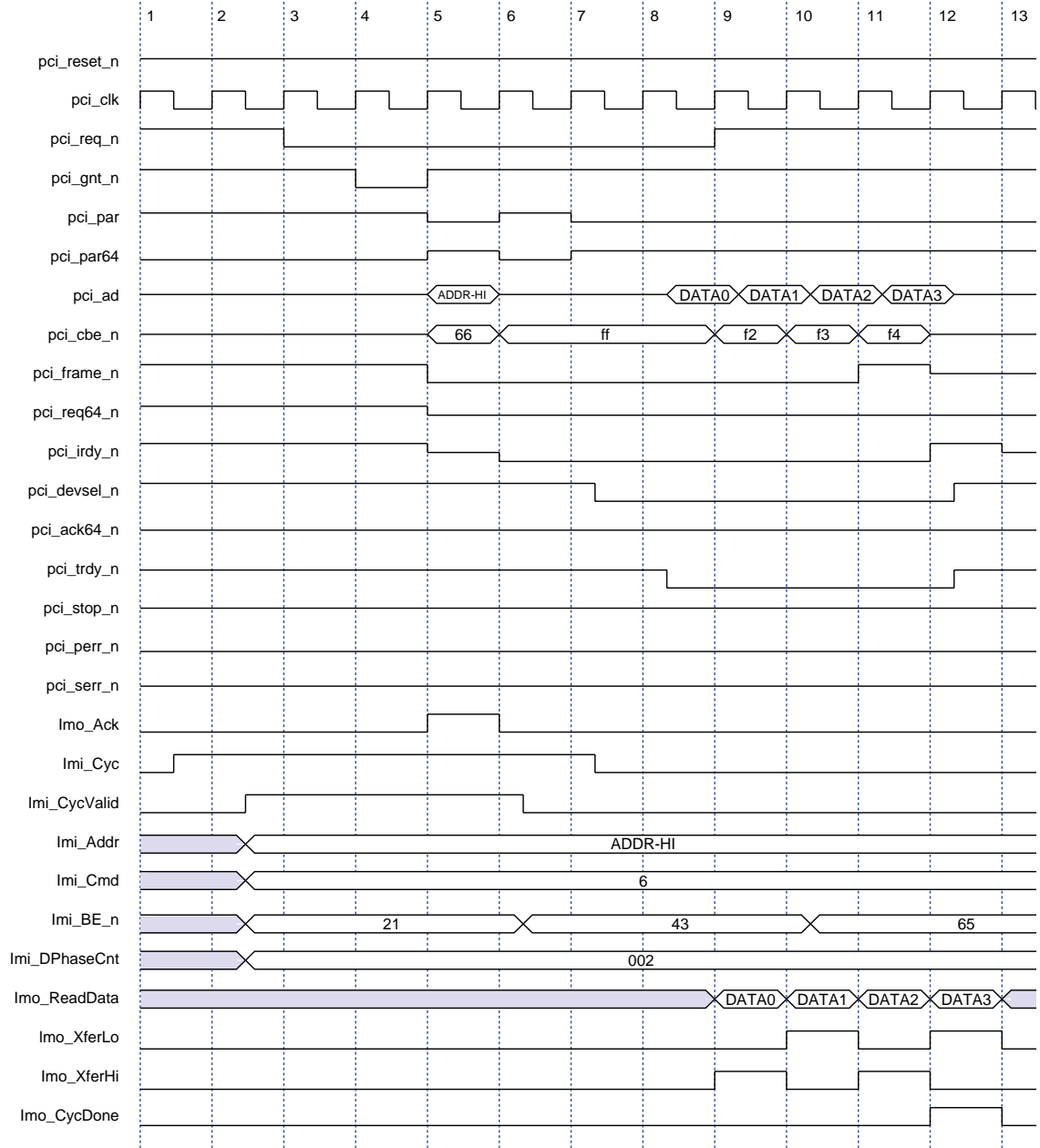




Figure 36 shows a 32-bit burst master memory read transaction. The sequence of events in Figure 36 is identical to Figure 35, except for the multiple data phases. The `pci_x` function initiates a 32-bit burst memory read transaction because the starting address is a high Dword boundary. The `pci_x` function registers four Dwords in clock cycles 10 through 13, and transfers them to the local side one Dword at a time in clock cycles 11 through 14. Because the starting address is a high Dword boundary (`pci_ad[2] = 1'b1`), the first Dword transferred to the local side is on `lmo_ReadData[63:32]` (indicated with the assertion of `lmo_XferHi`), and the second Dword is transferred on `lmo_ReadData[31:0]` (indicated by the assertion of `lmo_XferLo`). The toggling between `lmo_XferLo` and `lmo_XferHi` indicates that only Dword transfers are occurring on the local side.

Figure 36. PCI 32-Bit Burst Master Memory Read Transaction



## Master Write Transactions

The `pci_x` function performs both PCI-X and PCI-2.2 master write transactions. There are a few major differences between the PCI-X and PCI-2.2 master write transactions; however, from the perspective of the local master bus, behavior of the `pci_x` during the transactions is similar.

PCI-X transactions are different than PCI-2.2 transactions because they include an attribute phase, block cycle support, and split transaction support. See “Target Read Transactions” on page 38 for more information.

### *PCI-X Master Write Transactions*

The following are some general operating rules for PCI-X master write transactions:

- To isolate the local master device from potential data-width mismatch with local target devices, the `pci_x` must toggle data. The `pci_x` uses toggle signals (`lmo_XferHi` and `lmo_XferLo`) to qualify whether the data is valid on the upper or lower Dword.

The `pci_x` always asserts `lmo_XferHi` and `lmo_XferLo` signals with respect to the negotiated `DPhaseCnt`—regardless of whether the PCI-X target is 32-bit or 64-bit (unless the local master device or PCI-X target terminate the transaction early).

For example, as per the PCI-X specification, the `pci_x` does a “copy down” of the Hi-Dword address and byte enables during a Hi-Dword memory write cycle. If a 32-bit PCI-X target responds, the Lo-Dword never gets transferred. However, the `pci_x` will generate a dummy Lo-Dword Xfer with the assertion of the first `lmo_XferHi` to keep the number of Dwords transferred consistent with the negotiated `DPhaseCnt`. If a 64-bit PCI-X target device does respond, the local master will see both `lmo_XferHi` and `lmo_XferLo` signals assert anyway, so the `pci_x` tries to save the local master from the responsibility of data acceptance width of the PCI-X target.

Another example is of the `pci_x` initiating a PCI-X memory write block cycle where the ending address is on the Lo-Dword, i.e., the sum of the starting address and the byte count ends on the Lo-Dword. If a 64-bit PCI-X target responds, the last data phase will result in both `lmo_XferHi` and `lmo_XferLo` signals asserting. A 32-bit PCI-X target however, would complete the transaction and deassert `pci_irdy_n` after accepting only the Lo-Dword. The `pci_x` will generate a dummy `lmo_XferHi` as the last Xfer to save the local master from the responsibility of data acceptance width of the PCI-X target.

- For master burst transactions: Because the deassertion of `pci_frame_n` is not an effective transaction termination indicator when master devices have three data phases—or fewer—to transfer, the PCI-X specification requires that master devices modify the byte count to accurately reflect the data phases by:
  - Modifying the value in `lmi_TBC` while initiating memory write and memory write block cycles that start three, two, or one Qword(s) from the first ADB.
  - Setting the local master to begin the memory burst cycle when `lmi_DiscADB` is asserted and when the cycle begins three, two or one Qword(s) away from the first ADB. If this option is used, the `pci_x` will generate the byte count up to the first ADB and then drive the byte count—instead of `lmi_TBC`—in the byte count field during the attribute phase.



For PCI-X memory burst transactions where the cycle starts greater than three Qwords from the first ADB and the local master asserts `lmi_DiscADB`, the `pci_x` will not need to modify the byte count because in this scenario `pci_frame_n` can be used as a transaction termination indicator.

In PCI-X protocol, the `pci_x` support two types of 64-bit master write transactions:

- Memory single-cycle read
- Memory burst read

### PCI-X 64-Bit Single-Cycle Master Memory Write Transaction

Figure 37 shows the waveform for a PCI-X 64-bit single-cycle master memory write transaction.

Figure 37. PCI-X 64-Bit Single-Cycle Master Memory Write Transaction

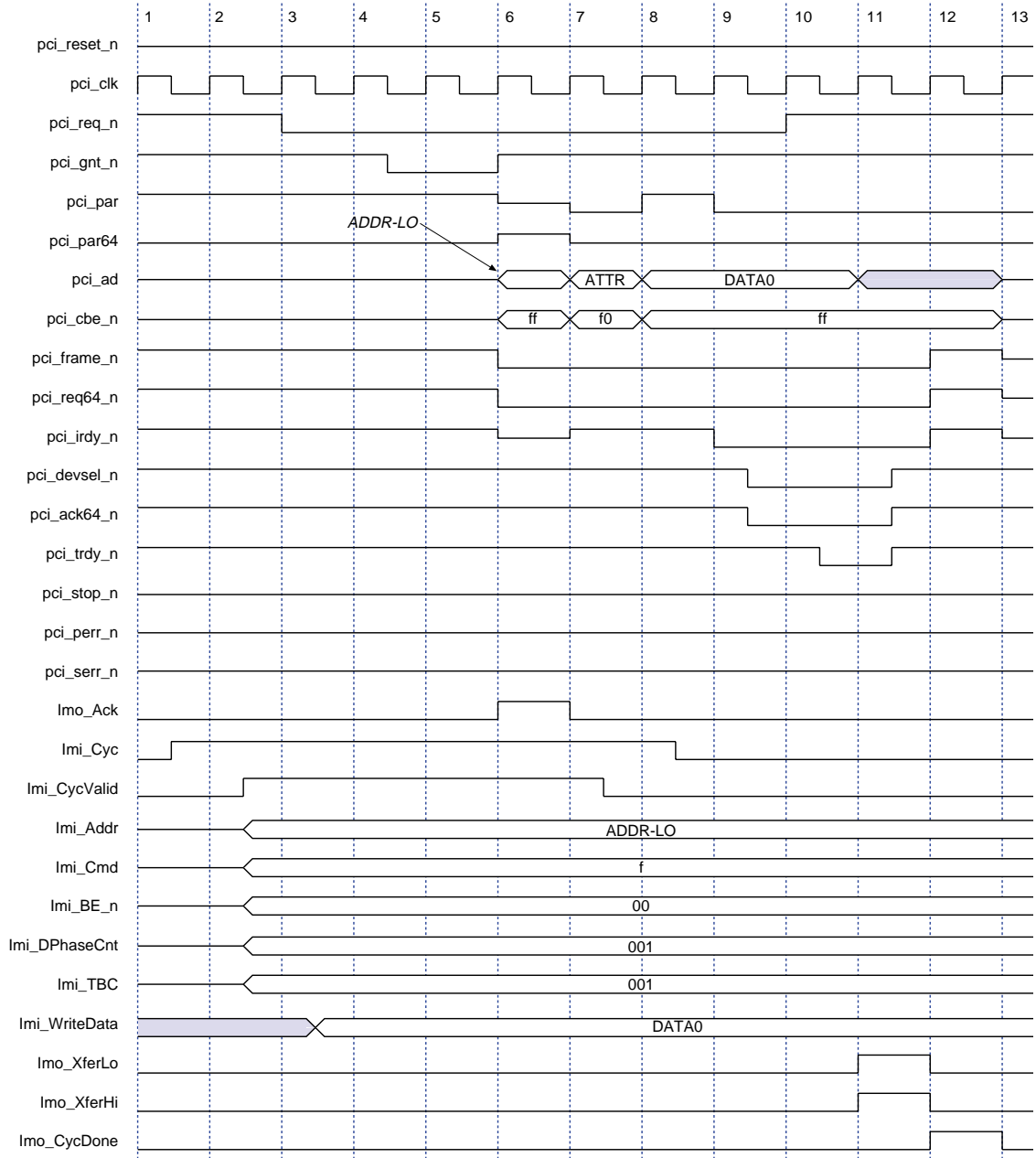


Table 5 shows the sequence of events for a PCI-X 64-bit single-cycle master memory write transaction.

<i>Table 5 .PCI-X 64-Bit Single-Cycle Master Memory Write Transactions (Part 1 of 2)</i>	
Clock Cycle	Event
1	Local side: The local master asserts <code>lmi_Cyc</code> to request ownership of the PCI-X bus.
2	Local side: The local master asserts <code>lmi_CycValid</code> to indicate valid local address ( <code>lmi_Addr[63:0]</code> ), command ( <code>lmi_Cmd[3:0]</code> ), byte enables ( <code>lmi_BE_n[7:0]</code> ), data phase count ( <code>lmi_DPhaseCnt[9:0]</code> ), and total byte count ( <code>lmi_TBC[11:0]</code> ). In addition (not shown in the waveforms), the local master provides the bus number ( <code>lmi_BusNo[7:0]</code> ), device number ( <code>lmi_DeviceNo[4:0]</code> ), function number ( <code>lmi_FunctionNo[2:0]</code> ), and tag number ( <code>lmi_TagNo[4:0]</code> ) information.
3	PCI-X side: With the assertion of <code>lmi_Cyc</code> in clock one, the <code>pci_x</code> function asserts <code>pci_req_n</code> on the PCI-X bus to request ownership of the bus. Local side: The local master continues to drive the same information as in clock 2. In addition, the local master drives the 64-bit data on <code>lmi_WriteData[63:0]</code> .
4	PCI-X side: The arbiter asserts <code>pci_gnt_n</code> to indicate bus ownership has been granted. However, it may not always be the case that the arbiter asserts <code>pci_gnt_n</code> one clock after receiving <code>pci_req_n</code> .
5	PCI-X side: The arbiter continues to assert <code>pci_gnt_n</code> . Local side: The local master continues to drive the same information as in clock 2, including <code>lmi_WriteData[63:0]</code> .
6	PCI-X side: The address phase. The <code>pci_x</code> asserts <code>pci_frame_n</code> to indicate the beginning of the transaction. The <code>pci_req64_n</code> is also asserted to indicate that a 64-bit transaction is being requested. Local side: The <code>pci_x</code> drives <code>lmo_Ack</code> to indicate that the PCI-X bus has been granted to the master.
7	PCI-X side: The attribute phase. The additional information provided by the local master in clock 2 is driven on <code>pci_ad[31:0]</code> and <code>pci_cbe_n[3:0]</code> . In this transaction, the total byte count to be transferred is eight, which is equivalent to one Qword. Local side: Because the attribute phase has taken place on the PCI-X side, the local master deasserts <code>lmi_CycValid</code> .
8	PCI-X side: The <code>pci_x</code> drives <code>pci_ad[63:0]</code> with the 64-bit data registered from <code>lmi_WriteData[63:0]</code> . Local side: The local master deasserts <code>lmi_Cyc</code> (immediately after this transaction) to indicate that it has no intention of requesting ownership of the bus.
9	PCI-X side: The PCI-X target claims the 64-bit transaction request by asserting <code>pci_devsel_n</code> and <code>pci_ack64_n</code> . In this particular case, the PCI-X target is a decode B device. The <code>pci_x</code> function asserts <code>pci_irdy_n</code> to indicate that it is ready to send data.

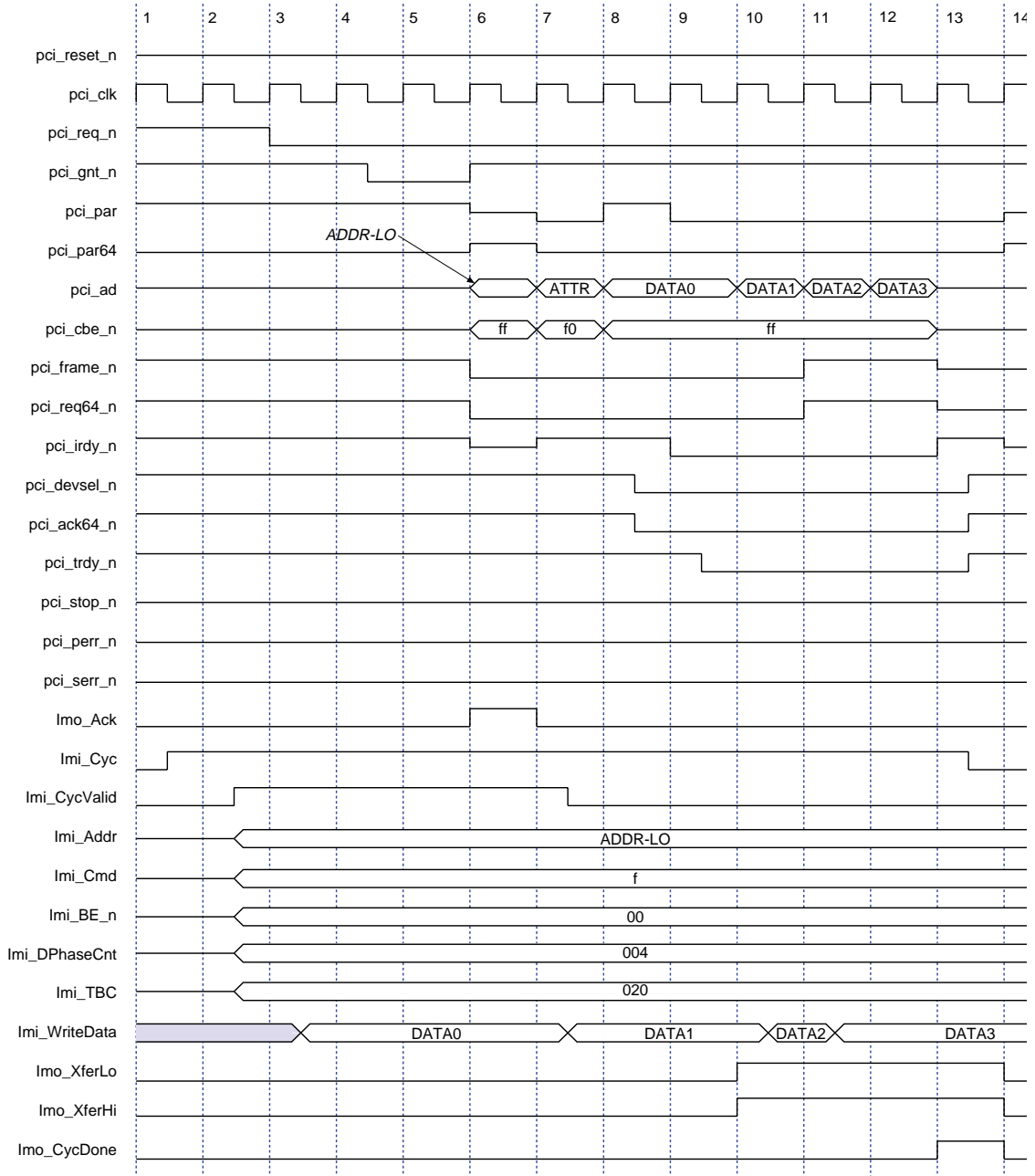
**Table 5 .PCI-X 64-Bit Single-Cycle Master Memory Write Transactions (Part 2 of 2)**

10	PCI-X side: The PCI-X target asserts <code>pci_trdy_n</code> to indicate that it is ready to accept data. Data transfer occurs on the PCI-X bus in this clock cycle.
11	PCI-X side: The PCI-X target deasserts <code>pci_devsel_n</code> , <code>pci_ack64_n</code> , and <code>pci_trdy_n</code> . Local side: The <code>pci_x</code> asserts <code>lmo_XferLo</code> and <code>lmo_XferHi</code> to indicate that a successful Qword transferred in the PCI-X bus in the previous clock cycle (clock 10).
12	PCI-X side: The <code>pci_x</code> deasserts <code>pci_frame_n</code> , <code>pci_req64_n</code> , and <code>pci_irdy_n</code> . Local side: The <code>pci_x</code> asserts <code>lmo_CycDone</code> to indicate that the transaction is complete.

### PCI-X 64-Bit Burst Master Memory Write Transaction

Figure 38 shows a 64-bit burst master memory write transaction. The sequence of events in Figure 38 is identical to Figure 37, except more data is transferred. Figure 38 shows a 64-bit zero wait state burst write transaction with four data phases. In clock 3, the local master drives the first 64-bit data on `lmi_WriteData[63:0]` and continues to drive the first Qword until clock 7. In clock 7, the local master increments to the second Qword due to the assertion of `lmo_Ack` in the previous clock (clock 6). The local master continues to drive the second Qword until the `pci_x` drives `lmo_XferLo` and `lmo_XferHi`, indicating that a successful Qword transfer on the PCI-X bus occurred in the previous cycle. Therefore, in clock 10, the local master increments to the third Qword. Because of the successful Qword transfer on the PCI-X bus in the previous cycle (with the assertion of `lmo_XferLo` and `lmo_XferHi`), the local master transfers the last Qword on `lmi_WriteData[63:0]` in clock 11.

Figure 38. PCI-X 64-Bit Burst Master Memory Write Transaction

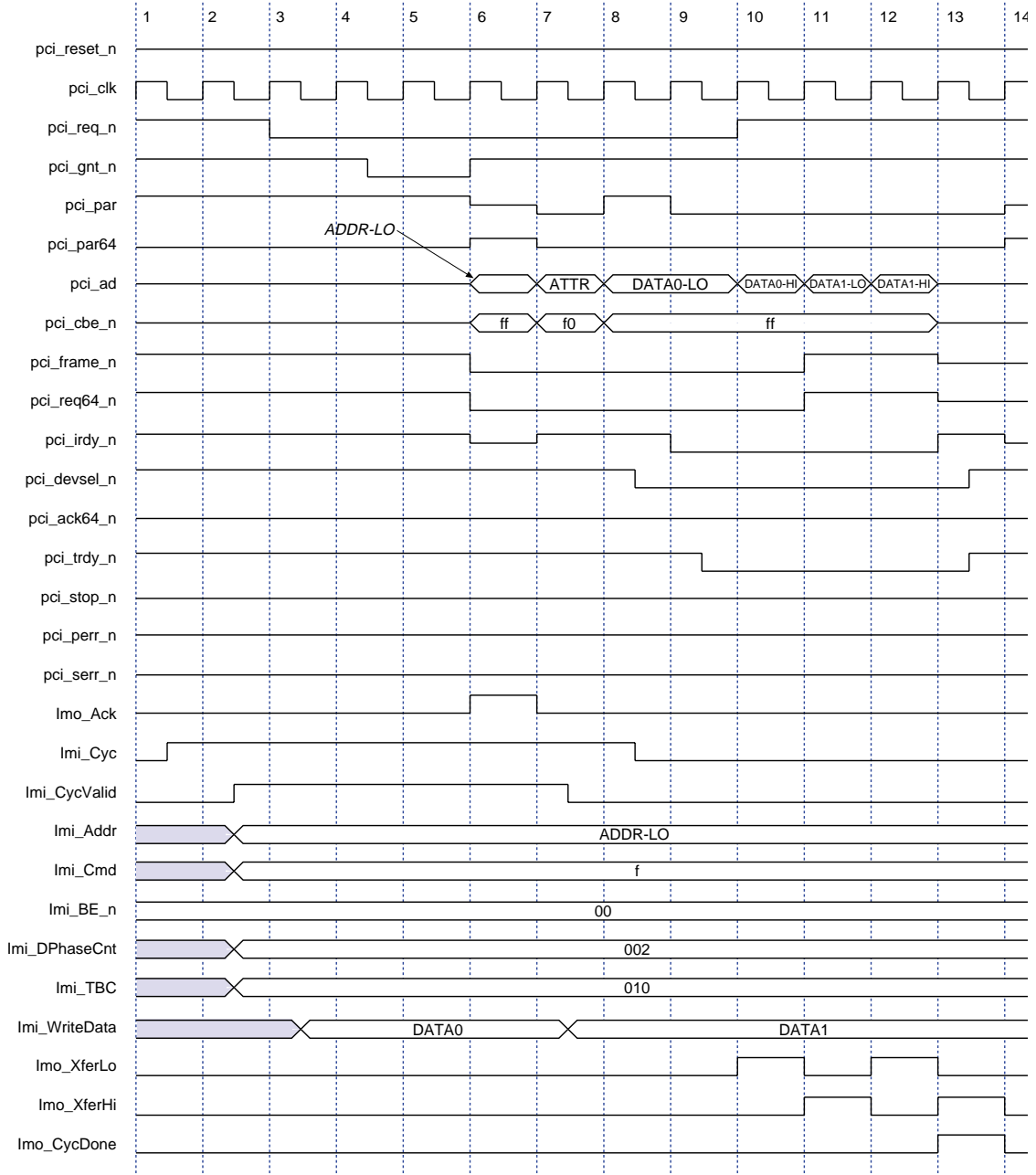




**PCI-X 64-Bit Burst Master Memory Write Transaction with 32-Bit Target**

Figure 39 shows the local master requesting a 64-bit transaction by asserting `pci_req64_n` in clock 6. However, the PCI-X target is only 32-bit (indicated by the non-active state of `pci_ack64_n` in clock 8). In clock 3, the local master drives the first 64-bit data on `lmi_WriteData[63:0]` and continues to drive the first Qword until clock 7. In clock 7, the local master increments to the second Qword due to the assertion of `lmo_Ack` in the previous clock (clock 6). Because the starting address is a low Dword boundary (`pci_ad[2] = 1'b0`), the `pci_x` asserts `lmo_XferLo` in clock 10 to indicate that the low Dword of the first Qword was transferred successfully on the PCI-X bus in the previous clock (clock 9). The toggling between `lmo_XferLo` and `lmo_XferHi` indicates that only Dword transfers are occurring on the PCI-X side.

Figure 39. PCI-X 64-Bit Burst Master Memory Write Transaction with 32-Bit Target



In PCI-X protocol, the `pci_x` can initiate two types of 32-bit master write transactions:

- I/O write
- Configuration write

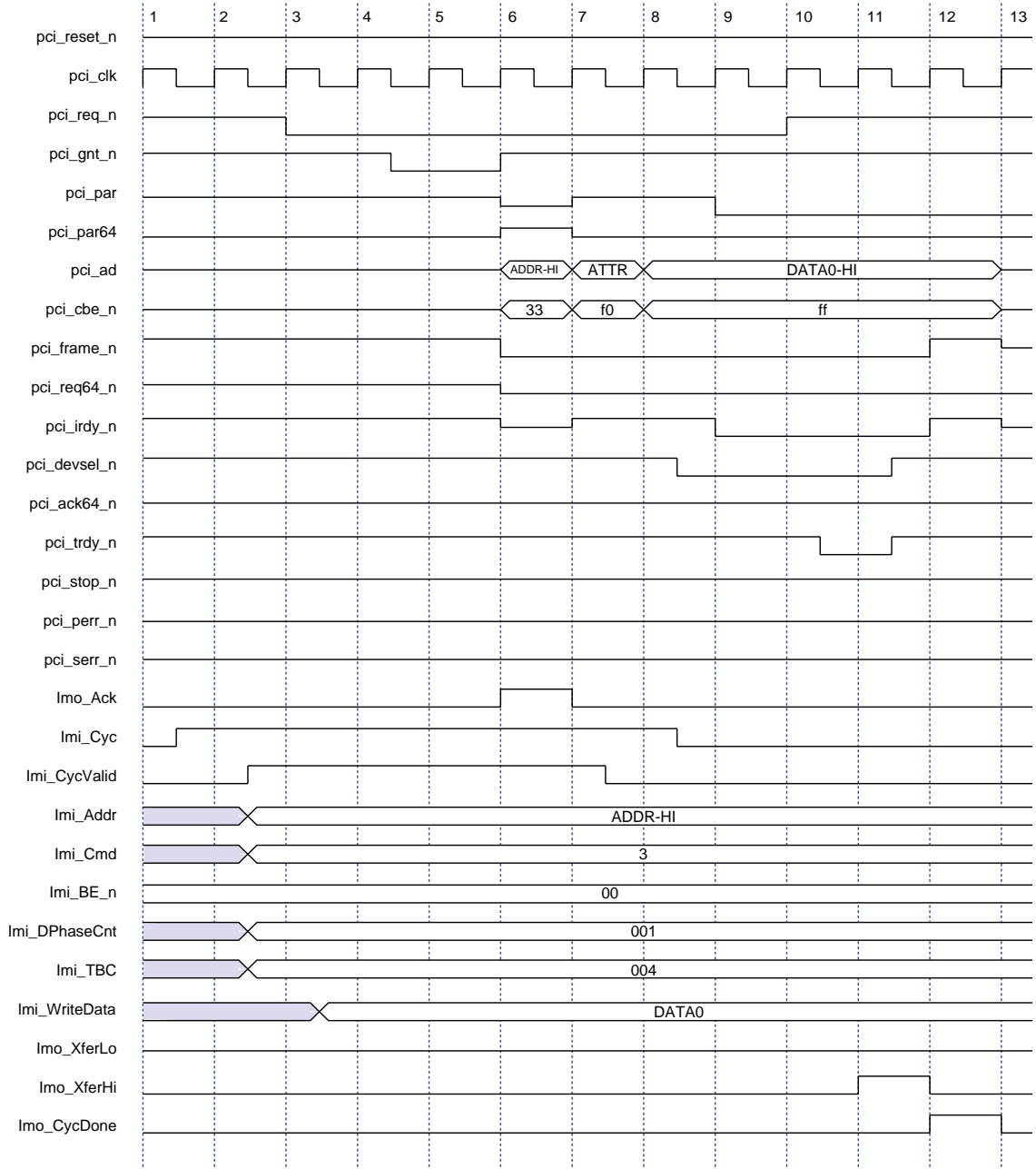


Because the `pci_x` is a 64-bit local master, it does not initiate 32-bit single-cycle or burst master memory write transactions (in PCI-X protocol).

### PCI-X Master I/O Write Transaction

By definition, a PCI-X I/O transaction is 32-bit and single-cycle. [Figure 40](#) shows a PCI-X master I/O write transaction. In clock 3, the local master drives the 64-bit data on `lmi_WriteData[63:0]`. Because the starting address is a high Dword boundary, the `pci_x` function registers `lmi_WriteData[63:32]` and transfers the Dword to the PCI-X target in clock 10. The `pci_x` function asserts `lmo_XferHi` in clock 11 to indicate that the upper Dword (`lmi_WriteData[63:32]`) was successfully transferred to the PCI-X bus in the previous clock cycle.

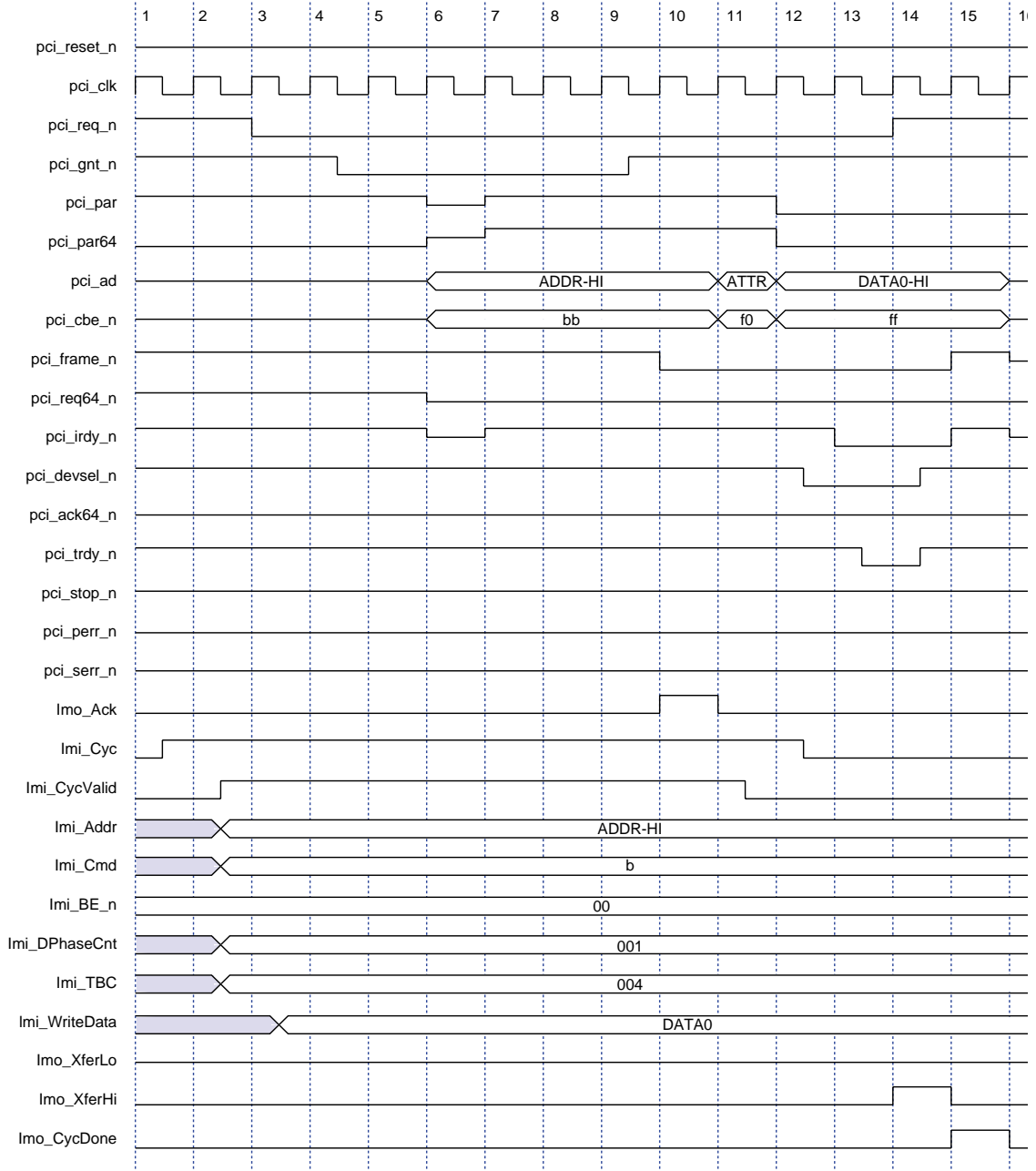
Figure 40. PCI-X Master I/O Write Transaction



### PCI-X Master Configuration Write Transaction

By definition, a PCI-X configuration transaction is 32-bit and single-cycle. [Figure 41](#) shows a master configuration read transaction. The sequence of events in [Figure 41](#) is similar to [Figure 40](#), except that the `pci_x` function performs address stepping, i.e., driving the address bus (`pci_ad[63:0]`) for four clock cycles before the PCI-X address phase. In clock 3, the local master drives the 64-bit data on `lmi_WriteData[63:0]`. Because the starting address is a high Dword boundary, the `pci_x` registers `lmi_WriteData[63:32]` and transfers the Dword to the PCI-X target in clock 13. The `pci_x` asserts `lmo_XferHi` in clock 14 to indicate that the upper Dword (`lmi_WriteData[63:32]`) was successfully transferred to the PCI-X bus in the previous clock cycle.

Figure 41. PCI-X Master Configuration Write Transaction



### PCI-2.2 Master Write Transactions

From the perspective of the local master bus, the behavior of the `pci_x` during PCI-2.2 write cycles is similar to the behavior during PCI-X write cycles, except for the following:

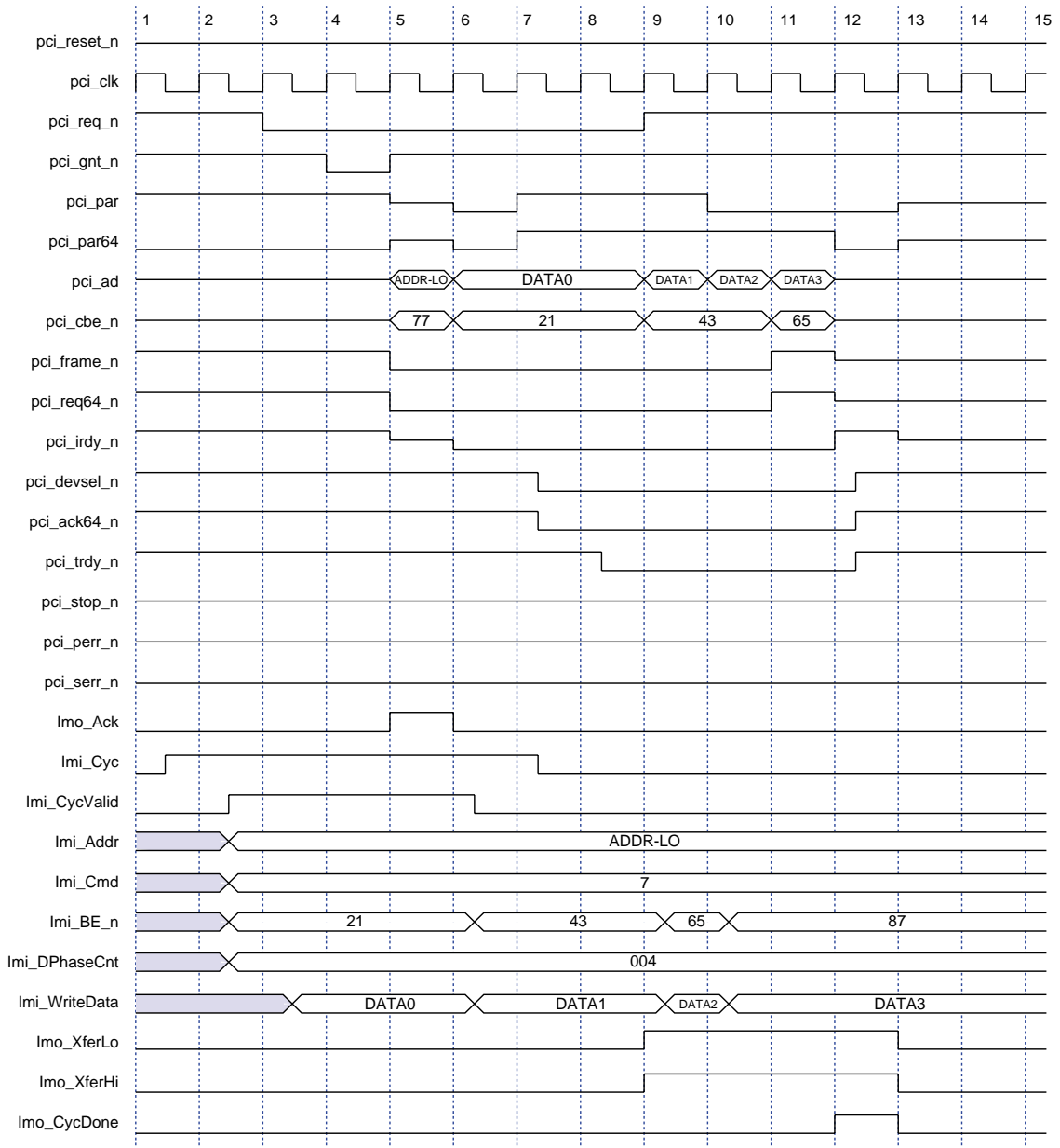
- There is no attribute phase.
- Does not initiate 64-bit single-cycle master read transactions.
- The `pci_x` function can initiate a 32-bit burst master memory write transaction if the starting address is a high Dword boundary. However, if the starting address is a low Dword boundary, the `pci_x` initiates a 64-bit transaction.

In PCI-2.2 protocol, the `pci_x` function initiates one type of 64-bit write transaction: Memory burst write

#### PCI-2.2 64-Bit Burst Master Memory Write Transaction

Figure 42 shows a 64-bit zero wait state burst master memory write transaction with four data phases. In clock 3, the local master drives the first 64-bit data on `lmi_WriteData[63:0]` and continues to drive the first Qword until clock 6. In clock 6, the local master increments to the second Qword due to the assertion of `lmo_Ack` in the previous clock cycle (clock 5). The local master continues to drive the second Qword until the `pci_x` drives `lmo_XferLo` and `lmo_XferHi`, indicating that a successful Qword transfer on the PCI-X bus occurred in the previous cycle. Therefore, in clock 9, the local master increments to the third Qword. Because of the successful Qword transfer on the PCI-X bus (with the assertion of `lmo_XferLo` and `lmo_XferHi`) the local master transfers the last Qword on `lmi_WriteData[63:0]` in clock 10.

Figure 42. PCI-2.2 64-Bit Burst Master Memory Write Transaction





In PCI-2.2 protocol, the `pci_x` responds to three types of 32-bit master write transactions:

- Memory write
- I/O write
- Configuration write



The `pci_x` function does support single-cycle and burst memory transactions, but does not support bursting of configuration or I/O cycles.

### PCI-2.2 32-bit Master Memory Write Transactions

Memory transactions are either single-cycle or burst. In PCI-2.2 protocol, the `pci_x` function can initiate a 32-bit single-cycle, depending on `lmi_BE_n[7:0]`. To initiate a 32-bit single-cycle with a low Dword starting address, set `lmi_BE_n[7:0] = 8'Hf0`. To initiate a 32-bit single-cycle with a high Dword starting address, set `lmi_BE_n[7:0] = 8'h0F`. In PCI-2.2 protocol, the `pci_x` only initiates 32-bit burst master memory write transactions if the starting address is a high Dword boundary.

**Figure 43** shows a 32-bit single-cycle master memory read transaction. In **Figure 43**, the local master initiates a high Dword starting address transaction. In addition, the local master sets `lmi_BE_n = 8'h0F` to indicate a 32-bit single-cycle, where the upper Dword (`lmi_WriteData[63:32]`) will be transferred to the PCI side in clock 11. In clock 12, the `pci_x` drives a dummy `lmo_XferLo`.

Figure 43. PCI-2.2 32-Bit Single-Cycle Master Memory Write Transaction

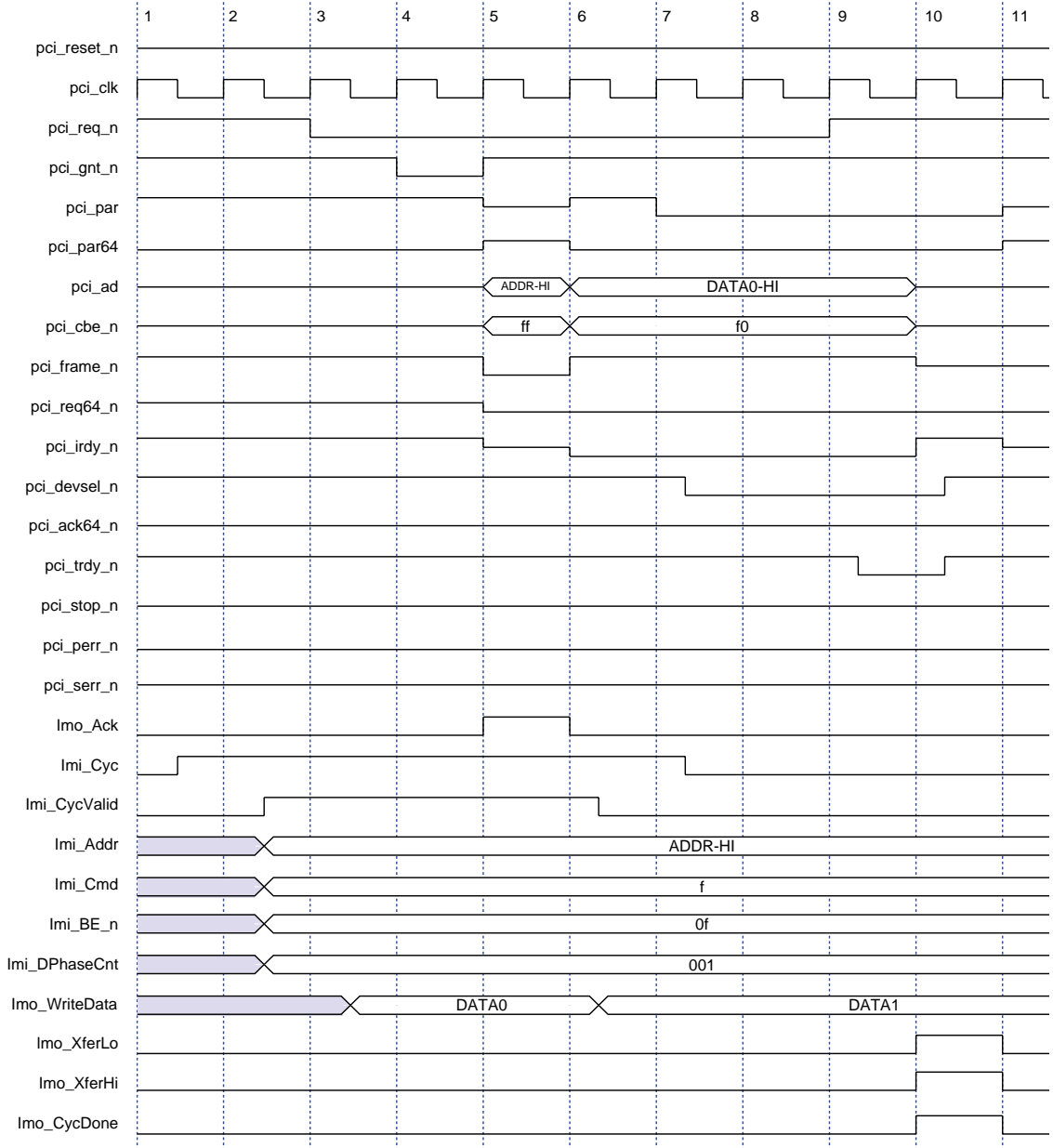
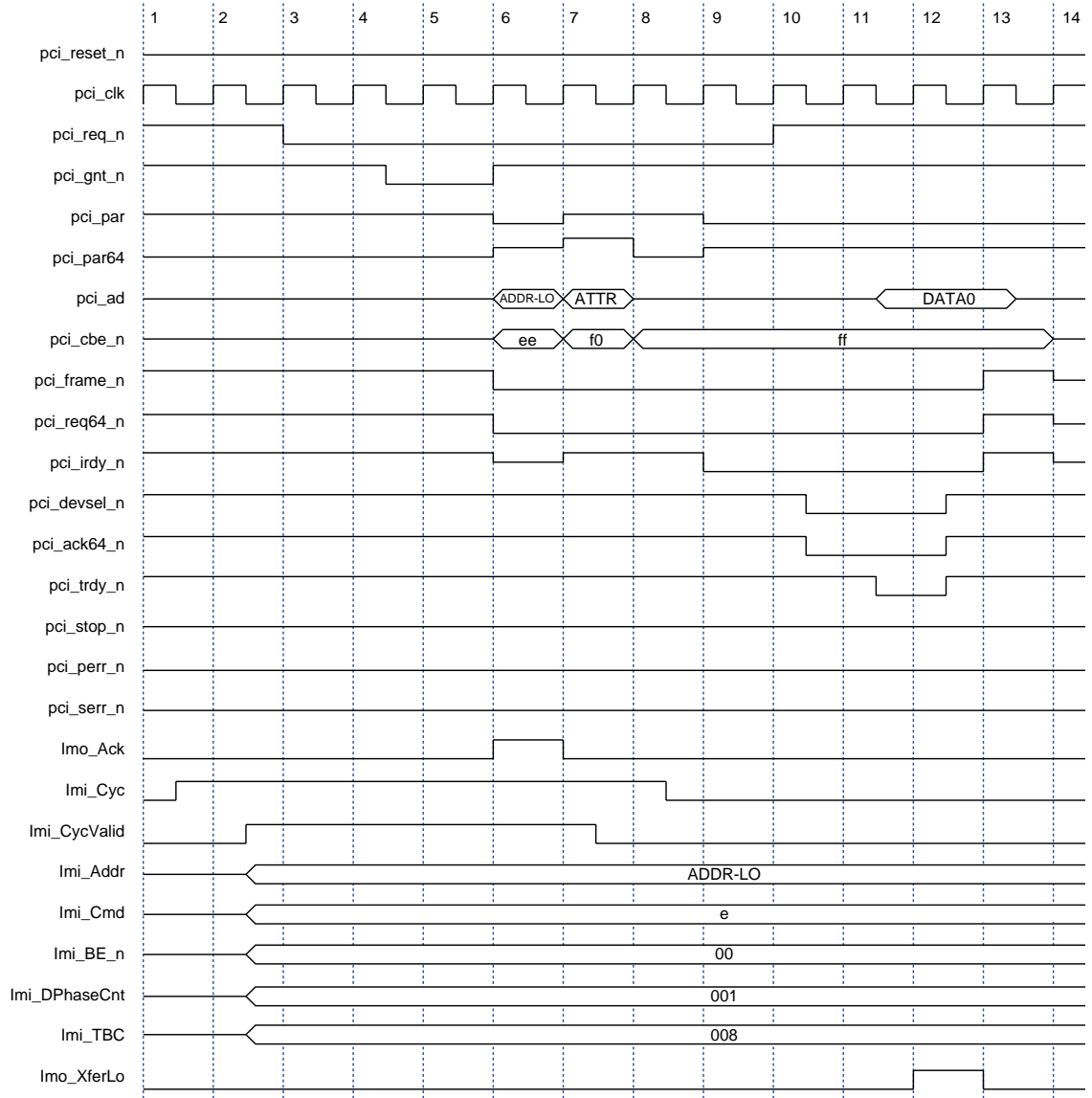


Figure 44 shows a 32-bit burst master memory write transaction. In clock 3, the local master drives the first 64-bit data on `lmi_WriteData[63:0]` and continues to drive the first Qword until clock 6. In clock 6, the local master increments to the second Qword because of the assertion of `lmo_Ack` in the previous clock cycle. Also because the starting address is at a low Dword boundary (`pci_ad[2] = 1'b1`), the `pci_x` asserts `lmo_XferHi` in clock 9 to indicate that the low Dword of the first Qword was transferred successfully on the PCI-X bus in the previous clock (clock 8). The toggling between `lmo_XferLo` and `lmo_XferHi` indicates that only Dword transfers are occurring on the PCI-X side.

Figure 44. PCI-2.2 32-Bit Burst Master Memory Write Transaction



## Split Transactions

As mentioned in “Target Read Transactions” on page 38, the PCI-X protocol provides a split transaction feature, which is the ability of PCI-X bus agents to split the data transfer. Splitting the data transfer allows other peripheral devices to perform transactions during an active PCI-X bus cycle.

The following command types can use the split transaction feature:

- Memory read block
- Alias to memory read block
- Memory read Dword
- Interrupt acknowledge
- I/O read
- I/O write
- Configuration read
- Configuration write

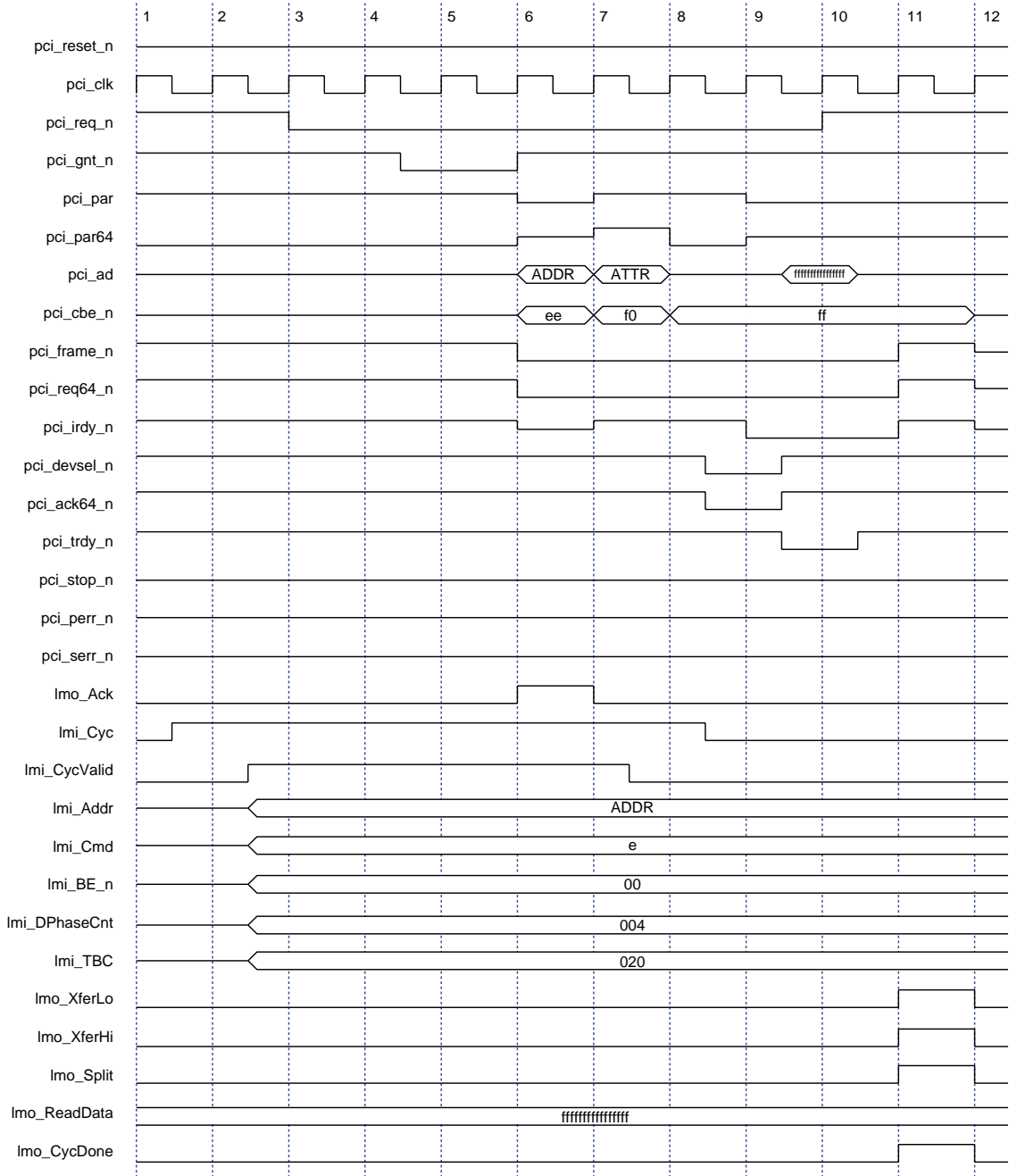
PCI specification uses additional terminology when referring to PCI-X bus devices involved with split transactions. The simple target/master terminology can become confusing, so requester and completer are also used. For example, when a master device (requester) issues a request, the target device (completer) can issue a split response. To complete this transaction, the target device must become the master device and vice versa. The following device A and device B example explains the steps of a split transaction:

1. Device A: The master device (requester) issues a 64-bit master memory read request
2. Device B: In response to the master device’s request, the target device (completer) issues a split response. At this point the master device releases itself from mastership of the bus.
3. Device B: The target device must now issue the split completion request, making the target device the master device.
4. Device A: The original master device is now on the receiving end; thus, it becomes the target device.

### Master Device Receives Split Response

Figure 45 shows a local master device requesting to read four Qwords from a target device. However, the target device issues a split response.

Figure 45. Master Receives Split Response For a PCI-X 64-Bit Master Memory Read Request

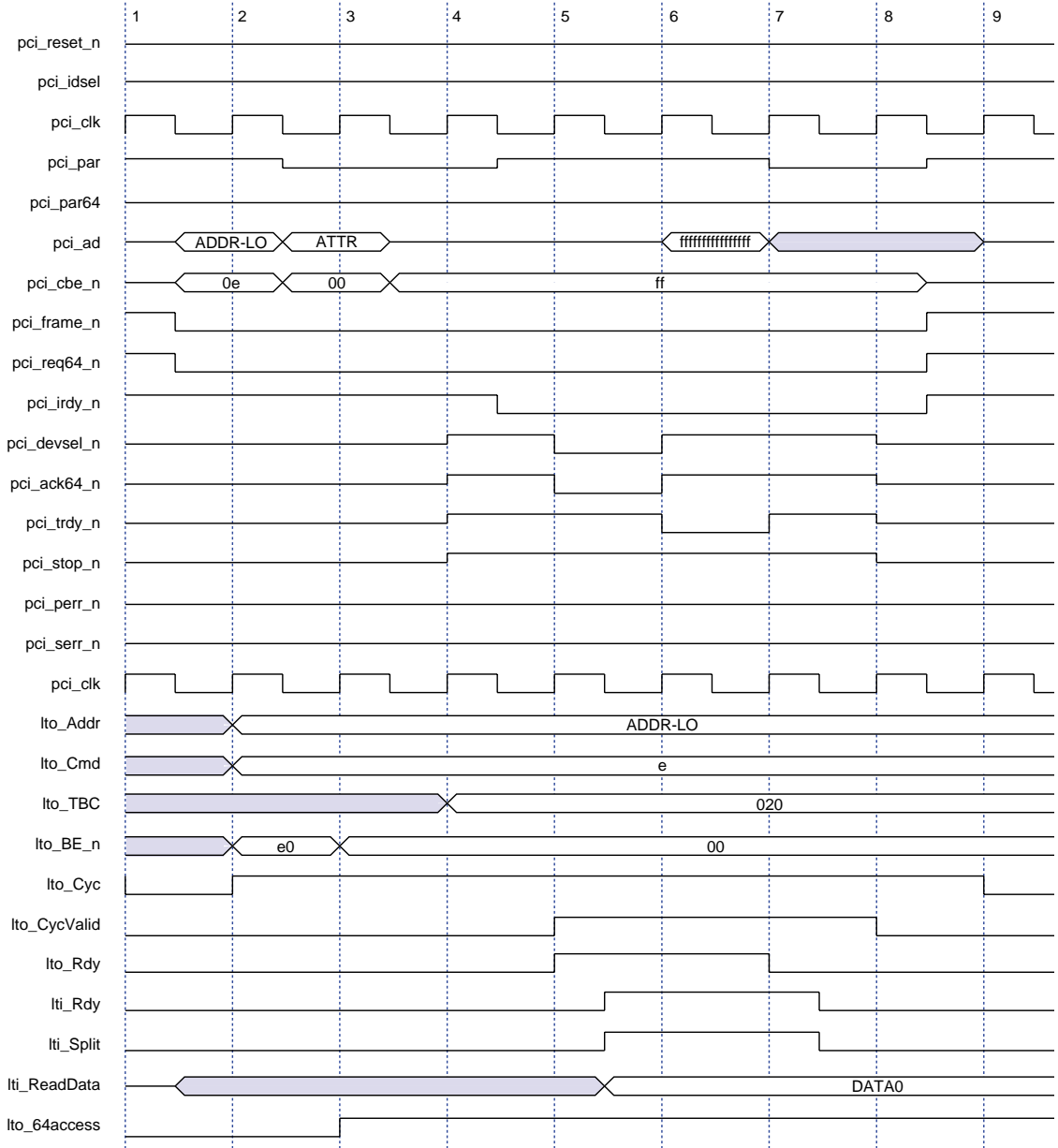


## Target Device Issues Split Response

Figure 46 shows the local target issuing a split response to a master device's 64-bit target memory read request. In Figure 46, the master is requesting to read four Qwords from the local target. However, the local target asserts `lti_Split` to indicate a split response.

When a local target issues a split response, the local target is responsible for storing the relevant transaction information and then, subsequently, issuing the split completion cycle on the local master bus. Throughout the assertion of `lto_Cyc`, the `pci_x` will maintain a stable value for the transaction information, including address (`lto_Addr`), command (`lto_Cmd`), device number (`lto_DeviceNo`), bus number (`lto_BusNo`), and tag number (`lto_TagNo`).

Figure 46. Target Issues Split Response To PCI-X 64-Bit Target Memory Read Transaction





## Master Issues Split Completion

From the perspective of the local bus, PCI-X master split completion transactions are similar to PCI-X master write transactions. Also, the `pci_x` isolates the local master from potential data-width mismatch issues with the target device, i.e., the `pci_x` asserts dummy `lmo_XferHi` and `lmo_XferLo` signals if a particular Dword is not transferred because a 64-bit transaction is responded to by a 32-bit target device.

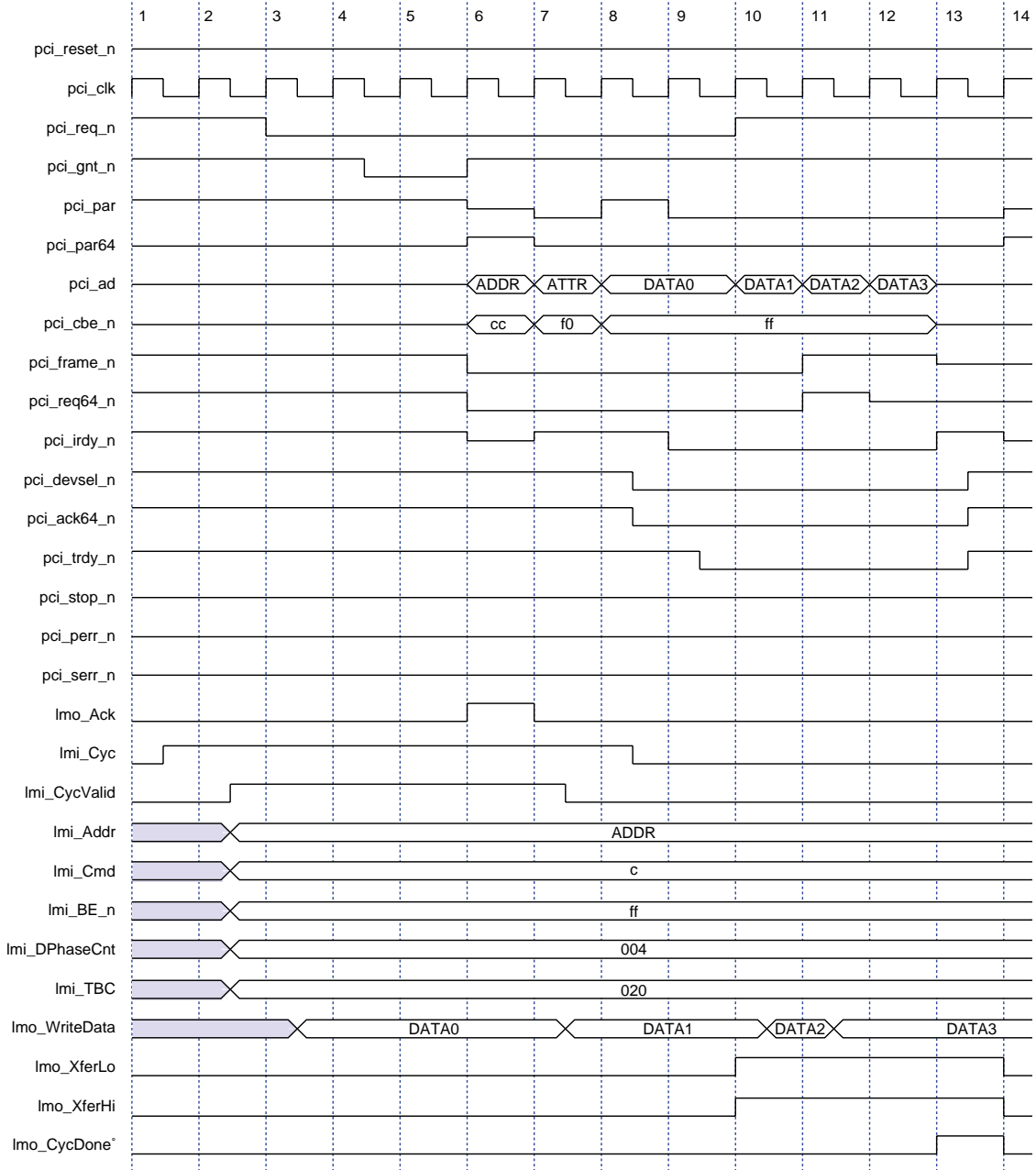
For PCI-X split completion transactions where the transaction is the only split completion for a memory read block request—or first of a series of split completion transactions for a memory read block request: The PCI-X specification requires that master devices—with three data phases or fewer to transfer—modify the byte count to accurately reflect the data phases. This requirement is needed because in this scenario the deassertion of `pci_frame_n` is not an effective transaction termination indicator.

Local master devices may choose to modify the transaction byte count (`lmi_TBC`) automatically while initiating certain split completions. If the local master modifies the TBC field to reflect a byte count that is from the starting address only up to the first ADB, the local master must set the `lmi_BCModified` signal. The `pci_x` will then set the appropriate BCM attribute bit during the split completion attribute phase.

However, the `pci_x` is also capable of modifying the byte count if the local master begins the split completion cycle (which begins three, two, or one Qword away from the first ADB) with `lmi_DiscADB` asserted. In this scenario—although the local master does not assert `lmi_BCModified`—the `pci_x` will generate the byte count up to the first ADB and set the appropriate BCM attribute bit during the split completion attribute phase. For split completion transactions where the cycle starts greater than three Qwords from the first ADB and the local master device asserts `lmi_DiscADB`, the `pci_x` will not modify the byte count because there is enough time to use the deassertion of `pci_frame_n` as a transaction termination indicator.

**Figure 47** shows the local master device issuing a split completion to a 64-bit target device. In this transaction the local master is actually the original target device (i.e., device B in the example on page 133). Thus, the local master device is completing a transaction to which it originally issued a split response. The split completion transaction in **Figure 47** is similar to a 64-bit PCI-X master memory write transaction.

Figure 47. Master Device Issues Split Completion



## Target Device Receives Split Completion

Data steering and byte enable manipulation split completion transactions are similar to memory write block transactions. Also, regardless of whether there is a 32-bit or 64-bit PCI-X completer running the transaction, the `pci_x` will forward the split completion data to the local target bus in 64-bit increments. While the PCI-X master does not drive any byte enables for a split completion transaction (`pci_be_n` bus is “reserved drive high”), the `pci_x` will generate valid byte enables on the local target `lto_BE_n` lines on a per data phase basis.

As in PCI-X memory write transactions, parity information for split completion transactions appears two clocks after the assertion of `lto_Rdy`. The `pci_x` calculates expected parity and drives `lto_Perr` to 1'b1 to indicate that the data driven with `lto_Rdy` two clocks earlier had bad parity. Local targets can choose to use this information on a per data-phase basis or on a per transaction basis.

For a split completion transaction to be claimed by the `pci_x` target interface, the decode logic must assert `claim_Cyc` to the `pci_x` for split completion cycles when requester information driven by the completer matches the requester's information.

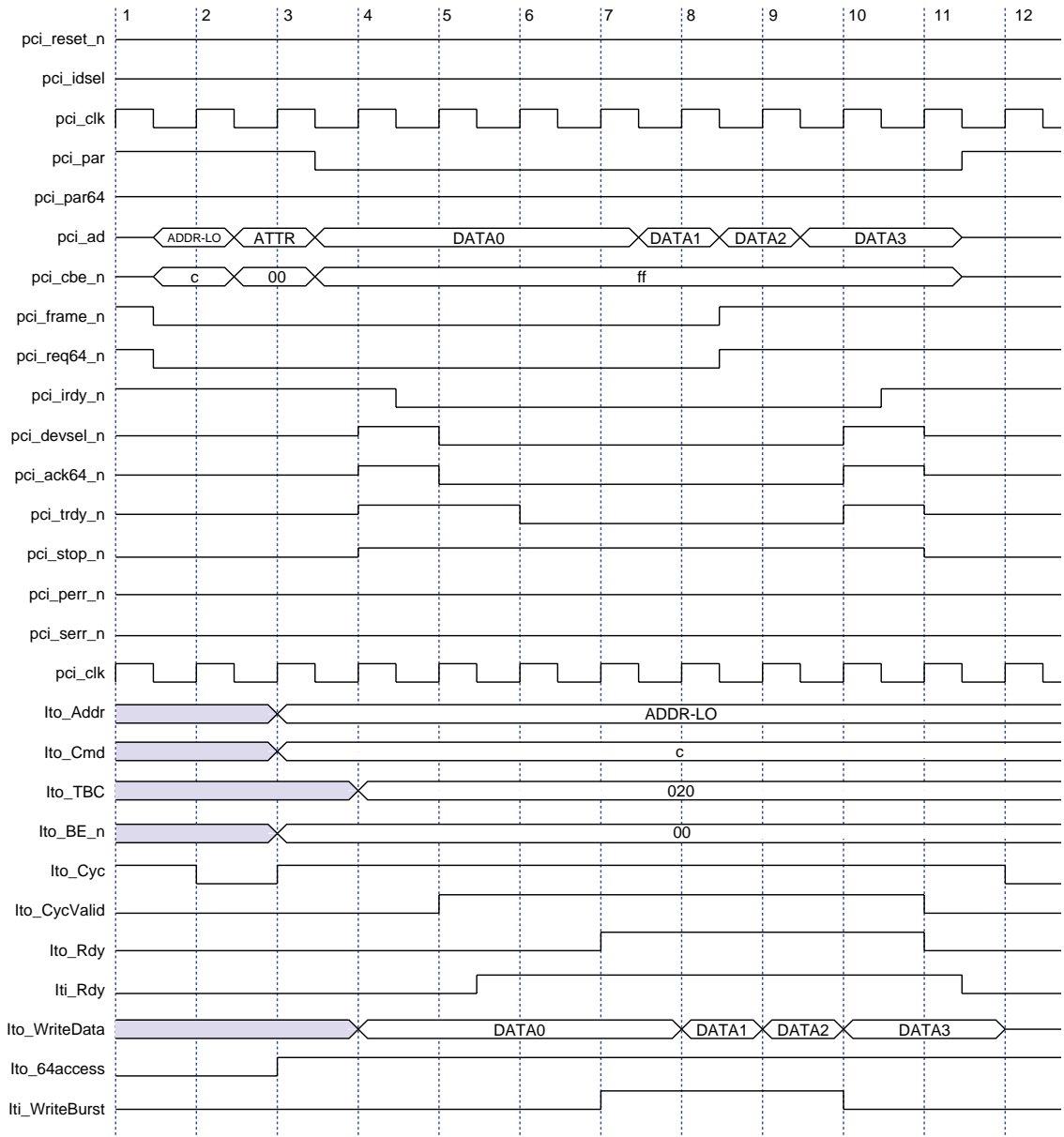
During split completions, the address presented to the local bus on `lto_Addr[63:0]` is the same as the PCI-X split completion address. The local target uses the requester's transaction information to match the transaction being received with the original transaction that was split on the local master interface. Also, because the lower seven bits of the address are driven on `lto_Addr[63:0]`, the local target has the necessary information to determine the next ADB boundary and can assert `lti_DiscADB` during the appropriate termination window if it wants the transaction to be terminated at the next ADB.

During split completion transactions, `lto_Addr[63:0]` holds the original requester's information, while other fields such as `lto_DeviceNo`, `lto_BusNo`, `lto_TagNo`, and `lto_TBC` hold the completer's information. These fields always carry information from the PCI-X master's attribute phase, regardless of whether the PCI-X master is running the cycle as a requester or completer.

Local targets must also anticipate the possibility of an unknown split completion transaction, i.e., a split completion received with a sequence number that does not match any outstanding split transactions. If this scenario occurs, the local target is obligated to assert `lti_Rdy` to allow the transaction to complete on the PCI-X bus and then bit-bucket all the data. There is also the possibility that a local target will receive a split completion exception message. See the PCI-X specification for more information.

Figure 48 shows the local target (device A) receiving a split completion, where the completer (device b) originally responded with a target split response but is now completing the transaction to the requester. In Figure 48, the 64-bit PCI-X completer issues a split completion to the local target. The local target treats the split completion transaction similarly to a 64-bit PCI-X target memory write transaction.

Figure 48. Target Receives a Split Completion



## Decode & Configuration

This section describes the contents of the decode/configuration module, which is in reference to the design file, **pcix\_config.v** (located in the **\lib** directory).

## Design Files

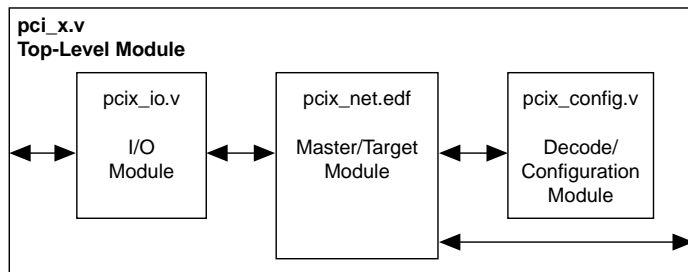
This section describes the `pci_x` Verilog HDL design files, including a figure illustrating the relationship between the files.

**Table 6** describes the `pci_x` design files.

<i>Table 6 .pci_x Design File Descriptions</i>	
Design File	Description
<code>pci_x.v</code>	PCI(X) Top-level module.
<code>pcix_net.edf</code>	PCI(X) Master/target module
<code>pcix_io.v</code>	PCI(X) Input/output (I/O) interface module
<code>pcix_config.v</code>	PCI(X) Decode/configuration space module

**Figure 49** shows the relationship between the `pci_x` design files.

**Figure 49. pci\_x Design File Relationship**



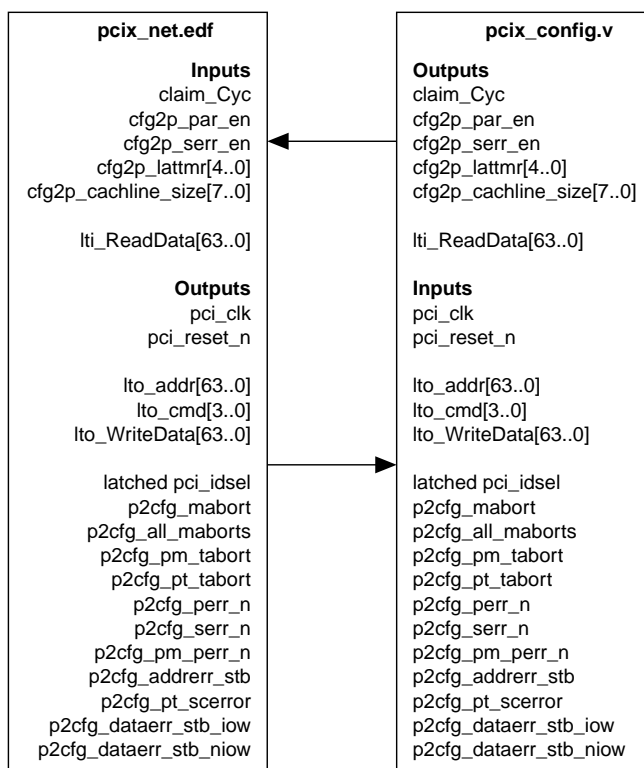
## Signal Descriptions

This section provides:

- An illustration of the master/target and decode/configuration module interface
- A description of the decode/configuration module signals

Figure 50 illustrates the interface between the master/target (**pcix\_net.edf**) and the decode/configuration modules (**pcix\_config.v**).

Figure 50. *pcix\_net.v and pcix\_config.v Interface*



**Table 7** describes the decode/configuration module (**pcix\_config.v**) signals.

<i>Table 7 .Signal Descriptions (Part 1 of 2)</i>		
Signal Name	Type	Description
pci_clk	Input	Global PCI clock input.
pci_reset_n	Input	Global PCI reset input.
lti_Addr[63:0]	Input	Latched version of the PCI address.
lti_Cmd[3:0]	Input	Latched version of the PCI command.
lti_WriteData[63:0]	Input	Latched version of the PCI data.
latched_pci_idsel	Input	Latched version of the PCI idsel.
p2cfg_mabort	Input	Indicates that the pci_x master device received a master abort on the PCI(X) bus (except for special cycles).
p2cfg_perr_n	Input	Indicates that the pci_x detected a PCI(X) data parity error.
p2cfg_pm_perr_n	Input	Indicates that a data parity error occurred when the pci_x is the master device or the target device when accepting PCI-X split completions.
p2cfg_pm_tabort	Input	Indicates that the pci_x master device received a target abort on the PCI(X) bus.
p2cfg_pt_tabort	Input	Indicates that the pci_x signaled a target abort.
p2cfg_serr_n	Input	Indicates that the pci_x asserted pci_serr_n.
p2cfg_pt_scerror	Input	Indicates that the pci_x received a split completion error message. Bit 29 of the PCI-X status register.
p2cfg_split_cmpln_disc	Input	Indicates that a split completion is discarded. Bit 18 of the PCI-X status register. User must generate logic to set this bit. Refer to section 5.4.4-5.4.5 of the <i>PCI-X Addendum, Revision 1.0</i> .
p2cfg_unexpected_split_cmpln	Input	Indicates unexpected split completion. Bit 19 of the PCI-X status register. User must generate logic to set this bit. Refer to section 5.4.4-5.4.5 of the <i>PCI-X Addendum, Revision 1.0</i> .
spci_bus_idle	Input	Registered signal indicates that the PCI bus is idle.
claim_Cyc	Output	Indicates that the pci_x has decoded that it is the target of the current access. Subsequently, the pci_x will assert pci_devsel_n.
cfg2p_cacheline_size [7:0]	Output	Cacheline size. Used by the pci_x for PCI-2.2 latency timer expiration during MWI cycles.
cfg2p_lattmr[7:3]	Output	Latency timer value.
cfg2p_par_en	Output	Enable parity checking.
cfg2p_perrecov_en	Output	Data parity error recovery enable. Bit 0 of the PCI-X command register.

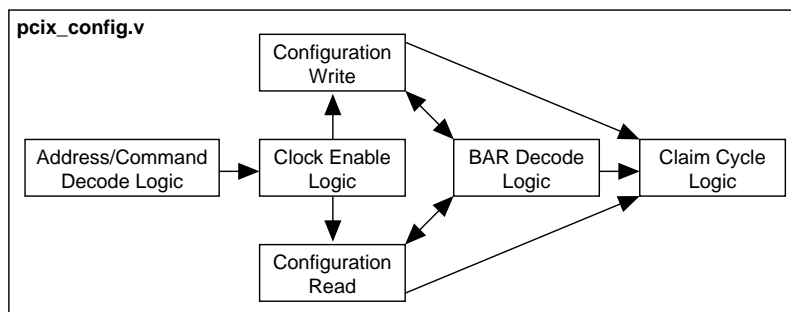


cfg2p_rlxord_en	Output	Relaxed ordering enable. Bit 2 of the PCI-X command register.
cfg2p_serr_en	Output	Enables pci_serr_n assertion on the PCI(X) bus.
s1_cfg_cyc_vld	Output	Registered signal indicates a valid configuration cycle.

## Functional Blocks

This section describes the functionality of the decode/configuration module. Figure 51 shows the functional blocks of the decode/configuration module.

Figure 51. Functional Blocks of the Decode/Configuration Module



### Address/Command Decode Logic

The address/command functional block decodes the PCI address and command signals. Using `lti_Addr[6:2]`, the logic determines which configuration register is being addressed. Using `lti_Cmd[3:0]`, the logic determines whether the current access is a configuration, I/O, or memory transaction.

### Clock Enable Logic

The clock enable functional block determines if the current configuration transaction is a valid cycle by decoding a configuration command, checking the assertion of the PCI(X) `idsel`, and checking that the two LSB's of the address (`lti_Addr[1:0]`) are zeros

### *Configuration Write*

If the current transaction is a valid configuration cycle and `lti_Cmd[0] = 1'b1` for a configuration write transaction, the configuration write functional block writes to the read/write registers of the configuration space.

### *Configuration Read*

If the current transaction is a valid configuration cycle and `lti_Cmd[1] = 1'b0` for a configuration read transaction, the configuration read functional block reads from the configuration registers.

### *BAR Decode Logic*

The BAR decode functional block compares the current transaction address to determine if the `pci_x` is the target of the current I/O or memory cycle.

### *Claim Cycle Logic*

The claim cycle functional block determines if the `pci_x` is the target of the current configuration, I/O, or memory transaction. If it is the target of the current transaction, the signal `claim_Cyc` is asserted.

## Design Considerations

This section discusses design considerations when modifying the decode/configuration module for your specific application. The detail implementation notes are in the design file, `pcix_config.v`.

As part of the capabilities list, the `pcix_config.v` file implements the first 64-bytes of the configuration space as well as the PCI-X registers. You can modify the Verilog HDL code to support only the configuration registers required for your design.

### *BAR Decode Logic*

In the beginning of the design, there are parameters that determine the type (I/O or memory) and the size of the space. The parameter for the read/write registers will determine the size of the space required. Each BAR can implement a 32-bit address space. Two BARs are needed to implement a 64-bit address space. The design file, **pcix\_config.v**, uses 2 base address registers (BAR0 and BAR1). BAR0 and BAR1 can each implement a 32-bit address space, but for a 64-bit address space, both BAR0 and BAR1 are needed. If your design requires more than 2 BARs, copy the BAR0 and BAR1 parameters and logic.

### *Claim Cycle Logic*

If more than 2 BARs are implemented for your design, you will need to add the conditions for the additional BAR hits.

### *PCI-X Split Completion Cycle Decode*

Targets must claim split completion transactions if the sequence ID driven in the split completion address matches the target's sequence ID.

The following "Partial Decode of Split Completion Method" is one way to decode a split completion cycle:

#### **Partial Decode of Split Completion Method**

Set the decode/configuration module to assert `claim_Cyc` when the device issuing a split completion transaction has the requester's bus number and device number fields of the split transaction, i.e., address matching the bus number and device number.

Because only a partial decode cycle has occurred to this point, the local target needs to determine (based on the tag number) if the split completion is valid or if the cycle is an unexpected split completion. Because `pci_devsel_n` is already asserted, unexpected split completion transactions need to be bit-bucketed by asserting `lti_Rdy`. Also, the local module is responsible for setting the unexpected split completion status bit in the PCI-X status register. Refer to section 5.4.4-5.4.5 of the **PCI-X Addendum, Revision 1.0**.

The "Partial Decode of Split Completions" method allows the decode/configuration module to do a "hit check" based on static information, i.e., the device's bus number and device number only. Also, because the bus number and device number already match, the local target is only required to match the split completion's tag number.



*Notes:*