LogiCORE™ IP Initiator/Target v6.8 for PCI-X

User Guide

UG263 April 24, 2009

XILINX®

Initiator/Target v6.8 for PCI-X User Guide

UG263 April 24, 2009

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
7/13/06	1.1	Initial Xilinx release
2/15/07	1.2	Updated core version to 6.2, added Virtex-5 LXT support..
5/17/07	1.5	Changed document title and text references to PCI/PCI-X to comply with PCI-SIG trademark guidelines. Advanced support for IUS to v5.7.
8/08/07	2.0	Updated for Jade Minor release.
10/10/07	2.1	Updated for IP2 Jade Minor release.
3/24/08	2.5	Updated for ISE v10.1 release.
4/25/08	3.0	Added support for Virtex-5 FXT devices; improved clock-management logic for PCI-X 133 MHz mode.
9/19/08	3.5	Updated for the Xilinx ISE v10.1 SP3 release.
4/24/09	4.0	Updated to support ISE v11.1.

Table of Contents

Chapter 12: Error Detection and Reporting

Appendix: Protocol Compliance Checklist

Schedule of Figures

Chapter 8: Basic Initiator Transaction Control

Chapter 9: Advanced Initiator Transaction Control

Chapter 10: Initiator Burst Transfers

Chapter 11: Other Bus Cycles

Chapter 12: Error Detection and Reporting

Appendix: Protocol Compliance Checklist

Schedule of Tables

About This Guide

This LogiCORE™ IP Initiator/Target v6.8 for PCI-X User Guide provides information about the Xilinx core interface for Peripheral Component Interconnect Extended (PCI-X), which provides a fully verified, pre-implemented PCI-X bus interface targeting devices based on the Virtex® architecture.

This guide serves as a comprehensive reference for use during the design phase of a project.

Guide Contents

This manual contains the following chapters:

- Chapter 1, "Getting Started" provides information about additional supporting documentation, getting technical support, and providing feedback to Xilinx about the Initiator/Target core for PCI-X interface and its accompanying documentation.
- Chapter 2, "Signal Descriptions" defines the PCI-X Bus interface signals and the user interface signals.
- Chapter 3, "General Design Guidelines" provides guidelines for turning a Initiator/Target core for PCI-X interface into a fully functioning design integrated with user application logic.
- Chapter 4, "Initialization and Interoperability" discusses system interoperability and the initialization of the core by the host system.
- Chapter 5, "Basic Target Transaction Control" describes the logic required to generate the load and output select signals for a typical target register in the user application.
- Chapter 6, "Advanced Target Transaction Control" discusses how the user application can control the target transactions to accommodate its own ability to source or sink data.
- Chapter 7, "Target Burst Transfers" describes the performance advantages derived from using burst transactions where maximum bandwidth is the goal.
- Chapter 8, "Basic Initiator Transaction Control" discusses the logic required in the user application to generate the load and output select signals for a typical initiator register.
- Chapter 9, "Advanced Initiator Transaction Control" discusses how the user application can control the initiator transactions to accommodate its own ability to source or sink data
- Chapter 10, "Initiator Burst Transfers" describes the performance advantages derived from using initiator burst transfers where maximum bandwidth is the goal.
- Chapter 11, "Other Bus Cycles" demonstrates the use of the more esoteric commands available with the Initiator/Target core for PCI-X.

- Chapter 12, "Error Detection and Reporting"defines how the core interface generates and checks parity, and how errors are reported.
- Appendix ,"Protocol Compliance Checklist" provides the PCI-SIG protocol compliance checklist to assist in the design review process.

Conventions

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Signal names, and system messages, prompts, and program files	`speed grade: - 100`
`Courier bold`	Literal commands you enter in a syntactical statement	**`ngdbuild`** *design_name*
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	`ngdbuild [option_name] design_name`
Braces { }	A list of items from which you must choose one or more	**`lowpwr ={on|off}`**
Vertical bar |	Separates items in a list of choices	**`lowpwr ={on|off}`**

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis ...	Omitted repetitive material	**`allow block`** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See "Additional Resources" for details. See "Title Formats," Chapter 1 for details.
Blue, underlined text	Hyperlink to a website (URL)	Go to www.xilinx.com for the latest speed files.

Chapter 1

Getting Started

The Initiator/Target core for PCI-X provides a fully-verified, pre-implemented PCI-X bus interface that targets devices that are based on the Virtex architecture. All code samples in this guide are provided in Verilog-HDL. An example design in both Verilog-HDL and VHDL is provided in the *Initiator/Target v6.8* for *PCI-X Getting Started Guide*.

About the Core

The Initiator/Target core for PCI-X is a Xilinx CORE Generator™ IP core, included in the latest IP Update on the Xilinx IP Center. For detailed information about the core, go to PCI/PCI-X. For information about licensing options, see "Chapter 2, Licensing the Core" in the *Initiator/Target v6.8* for *PCI-X Getting Started Guide.*

Other Documentation

For more information about the core, see the following documents, provided in the CORE Generator software zip file:

- *Initiator/Target v6.8* for *PCI-X Release Notes*
- *Initiator/Target v6.8* for *PCI-X Getting Started Guide*

Further information is available in the Mindshare PCI System Architecture text, and the PCI Local Bus Specification, available from the PCI Special Interest Group site.

Technical Support

For technical support, visit www.xilinx.com/support. Questions are routed to a team of engineers with expertise using the Initiator/Target core for PCI-X.

Xilinx provides technical support for use of this product as described in the *Initiator/Target v6.8* for *PCI-X User Guide* and the *Initiator/Target v6.8* for *PCI-X Getting Started Guide*. Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the core and the documentation supplied with the core.

PCI-X Interface Core

For comments or suggestions about the core, please submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

Document

For comments or suggestions about this document, please submit a WebCase from www.xilinx.com/support/clearexpress/websupport.htm. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

Chapter 2

Signal Descriptions

This chapter defines the PCI-X standard bus interface signals and all user interface signals.

PCI-X Bus Interface Signals

Table 2-1 defines the interface signals that comprise the PCI-X Local Bus. These signals are shown on the left side of Figure 2-1. Note that Figure 2-1 represents a 64-bit design; 32-bit designs have a narrower data path and fewer control signals. These differences are described in the following sections.

Pin locations are device and package dependent. See the appropriate user constraints file for specific device configurations.

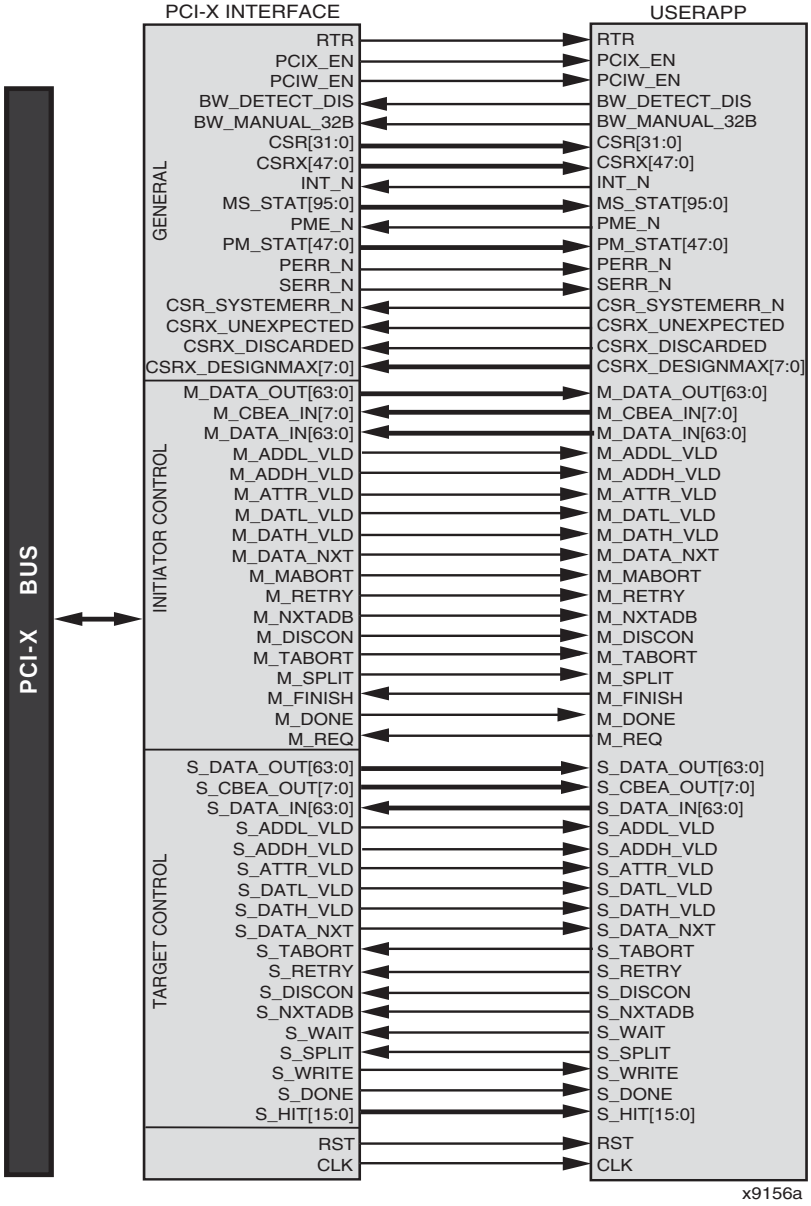


Figure 2-1: **Top-level Block Diagram**

Table 2-1: **PCI-X Bus Interface Signals**

Signal Name	Type	Functional Description
Address and Data Path		
AD_IO[63:0]	t/s	A time-multiplexed address, attribute, and data bus. Each bus transaction consists of an address phase followed by an attribute phase and one or more data phases. In 32-bit versions of this interface, this bus is half as wide.

Table 2-1: **PCI-X Bus Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
CBE_IO[7:0]	t/s	A time-multiplexed bus command and byte enable bus. Bus commands are asserted during an address phase on the bus. Byte enables are driven during attribute and data phases. In 32-bit versions of this interface, this bus is half as wide.
PAR_IO	t/s	Generates and checks even parity across AD_IO[31:0] and CBE_IO[3:0]. When the core interface is the source of an address, attribute, or data, the interface generates even parity across AD_IO[31:0] and CBE_IO[3:0] and presents the result on PAR_IO. When the interface receives an address, attribute, or data, the interface checks for even parity across AD_IO[31:0] and CBE_IO[3:0] and compares it to PAR_IO. Parity errors are reported via PERR_IO.
PAR64_IO	t/s	Generates and checks even parity across AD_IO[63:32] and CBE_IO[7:4]. When the core interface is the source of an address, attribute, or data, the interface generates even parity across AD_IO[63:32] and CBE_IO[7:4] and presents the result on PAR64_IO. When the interface receives an address, attribute, or data, the interface checks for even parity across AD_IO[63:32] and CBE_IO[7:4] and compares it to PAR64_IO. Parity errors are reported via PERR_IO. In 32-bit versions of this interface, this signal is not present.

Table 2-1: **PCI-X Bus Interface Signals** *(Continued)*

Signal Name	Type	Functional Description

FRAME_IO	s/t/s active low	Driven by an initiator to indicate a bus transaction. FRAME_IO is asserted for the duration of the operation and is deasserted to identify the end of the transaction.
		When operating as an initiator, the interface will only assert FRAME_IO when all of the following conditions are met:
		• GNT_I is asserted.
		• IRDY_IO and FRAME_IO are deasserted, meaning the bus is idle.
		• The bus master enable bit (**CSR2**) is set in the command register.
		• The user application has asserted M_REQ.
		The interface deasserts FRAME_IO upon any of the following conditions:
		• The specified byte count is satisfied.
		• The user application asserts M_FINISH
		  • In PCI mode the transaction will finish as soon as possible.
		  • In PCI-X mode the transaction will finish at the next ADB (allowable disconnect boundary).
		• The interface receives a termination from the addressed target.
		• *Not* receiving a DEVSEL_IO assertion from the addressed target (master abort).
		• The internal latency timer has expired and the system arbiter is no longer asserting GNT_I.

REQ64_IO	s/t/s active low	Driven by the initiator to indicate a 64-bit bus transaction. It is asserted for the duration of the operation and is deasserted to identify the end of the transaction. Its behavior is similar to FRAME_IO. It is only asserted if the user application has asserted M_REQ and a 64-bit transaction was specified in the transfer attributes. In 32-bit versions of this interface, this signal is not present.

Table 2-1: **PCI-X Bus Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
DEVSEL_IO	s/t/s active low	Indicates that a target has decoded the address presented during the address phase and is claiming the transaction. This occurs when the address matches one of the Base Address Registers in the target.
ACK64_IO	s/t/s active low	Indicates that a target has decoded the address presented during the address phase and is claiming the transaction as a 64-bit target. This occurs when the address matches one of the Base Address Registers in the target, and that Base Address Register is configured to accept 64-bit transfers. In 32-bit versions of this interface, this signal is not present.
TRDY_IO	s/t/s active low	Indicates that the target is ready to complete the current data phase. When asserted, the target is ready to transfer data. Data transfer occurs when both TRDY_IO and IRDY_IO are asserted on the bus.
IRDY_IO	s/t/s active low	Indicates that the initiator is ready to complete the current data phase. When asserted, the initiator is ready to transfer data. Data transfer occurs when both TRDY_IO and IRDY_IO are asserted on the bus.
STOP_IO	s/t/s active low	Indicates that the target has requested to stop the current transaction. The target uses STOP_IO to signal various termination conditions. The interface asserts STOP_IO under the control of the user application. However, the interface automatically asserts it under the following conditions: • Non-linear memory transactions. • Implied single data phase commands.
IDSEL_I	in	Indicates that the interface is the target of a configuration cycle.
Interrupts		
INT_O	o/d active low	Indicates the interface requests an interrupt. May be disabled by setting the interrupt disable bit in the command register.

Table 2-1: **PCI-X Bus Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
PME_O	o/d active low	Indicates that a power management event has taken place.
Error Signals		
PERR_IO	s/t/s active low	Indicates a parity error was detected while the interface was the target of a write transfer or the initiator of a read transfer. Parity error reporting is enabled by setting the report parity errors bit (CSR6) in the command register. Parity errors, except those during special cycles, are always reported in the status register (CSR31). The initiator reports parity errors during a transaction when it was the bus master. The error is reported via the data parity error detected bit (CSR24) in the status register if the report parity errors bit (CSR6) is set in the command register.
SERR_IO	o/d active low	Indicates a parity error was detected during an address or attribute cycle, except during special cycles. System errors are reported on the signaled system error bit (CSR30) in the status register if the SERR_IO enable bit (CSR8) and the report parity errors bit (CSR6) are set in the command register.
Arbitration		
REQ_O	t/s active low	Indicates to the arbiter that the initiator requests access to the bus. The initiator may only request the bus if enabled by setting the bus master enable bit (CSR2) in the command register.
GNT_I	t/s active low	Indicates the arbiter has granted the bus to the initiator. If GNT_I is asserted and there is *not* a pending request, or the bus master enable bit is not set, then the interface performs bus parking.
System Signals		
RST_I	in active low	The bus reset signal used to bring registers, sequencers, and signals to a consistent state. Any time RST_I is asserted, all output signals are three-stated.

Table 2-1: **PCI-X Bus Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
REF_I	in	A reference clock input used to calibrate input delays and operate the DCM Dynamic Reconfiguration Port, present in specific Virtex-5 implementations of this interface.
CLK_I	in	The bus clock signal that provides timing for all transactions on the bus and is an input to every device. The frequency of CLK_I may vary, as allowed in the *PCI Local Bus Specification*.

in = input only signal

out = output only signal

t/s = bidirectional, three-state signal

s/t/s = bidirectional, sustained three-state signal

o/d = open drain

User Interface Signals

The user interface to the core interface provides a superset of the necessary data paths and control signals required for typical applications. This provides ultimate flexibility for specialized user applications.

Table 2-2 describes the interface signals available to the user application. The direction of the signals in the table are respective to the core interface. These signals are displayed on the right side of Figure 2-1.

Table 2-2: **User Interface Signals**

Signal Name	Type	Functional Description
General Signals		
CLK	out	The bus clock driven by a global clock buffer. Use this clock for all flip-flops that are synchronized to the bus clock.
RST	out	An inverted copy of the bus reset signal. This signal may be used as an asynchronous reset signal for the user application.
RTR	out	An output signal used in dual configuration designs. When asserted, the user application must reconfigure the FPGA device with an alternate bitstream.
PCIX_EN	out	When asserted, the bus is operating in PCI-X mode.
PCIW_EN	out	When asserted, the bus is operating in 64-bit mode.
CSRX_DESIGNMAX [7:0]	in	This input bus is used to control the values reported for the following items in the PCI-X Capability Item in extended configuration space: • Designed Maximum Cumulative Read Size (bits 7 down to 5) • Designed Maximum Outstanding Split Transactions (bits 4 down to 2) • Designed Maximum Memory Read Byte Count (bits 1 down to 0) The user application must assign appropriate values to this port. For simple designs, constant values may be appropriate. For more complex designs, the values may need to be adjusted at run time based on the state of the PCI-X command register, CSRX[47:0].
BW_DETECT_DIS	in	The bus width detect disable port lets the user application force the width of the PCI-X bus as seen by the interface. When this bit is set to zero, the core auto-detects the bus width by sampling bus signals at the rising edge of the bus reset signal. When this bit is set to one, the interface instead samples BW_MANUAL_32B.
BW_MANUAL_32B	in	When BW_DETECT_DIS is set to one, this port controls the bus width assumed by the interface. Logic one indicates a 32-bit bus, while logic zero indicates a 64-bit bus.

Table 2-2: **User Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
BM_DETECT_DIS	in	This port must be tied to a constant in the interface wrapper file. Do not modify the wrapper files or the value tied to this port. This port is not made available to the user application.
BM_MANUAL_PCI	in	This port must be tied to a constant in the interface wrapper file. Do not modify the wrapper files or the value tied to this port. This port is not made available to the user application.
PERR_N	out	Signals that PERR# is asserted on the bus.
SERR_N	out	Signals that SERR# is asserted on the bus.
INT_N	in	The interrupt event control signal. It should remain asserted until the interrupt is cleared. The mechanism for clearing an interrupt is design-specific.
MS_STAT[95:0]	out	Provides interrupt support information, required for message signalled interrupts. For more information on the MSI capability structure, see section 6.8.1 of the *PCI Local Bus Specification v3.0*. This signal is mapped to the MSI capability structure as follows: MS_STAT[95:80] = Message Control MS_STAT[79:64] = Message Data MS_STAT[63:32] = Message Upper Address MS_STAT[31:0] = Message Lower Address
PME_N	in	The power management event control signal. This signal should be asserted for one cycle to indicate a power management event.
PM_STAT[47:0]	out	Provides access to the power management extended capability registers. For more information on the Power Management Interface capability structure, see section 3.2 of the *PCI Bus Power Management Interface Specification v1.2*. This signal is mapped to the PMI capability structure as follows: PM_STAT[47:40] - Data PM_STAT[39:32] - PMCSR_BSE Bridge Support Ext. PM_STAT[31:16] - P.M. Control/Status Reg. (PMCSR) PM_STAT[15:0] - Power Mgmt. Capabilities (PMC)

Table 2-2: **User Interface Signals** *(Continued)*

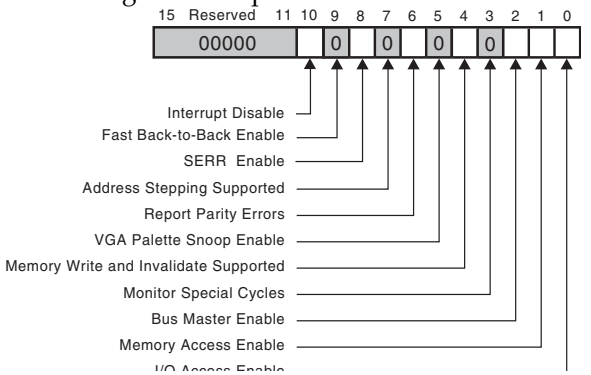
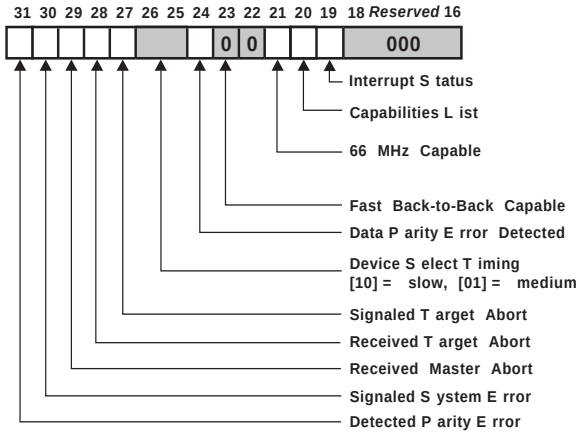
Signal Name	Type	Functional Description
CSR[15:0]	out	Provides access to the command register state bits. These bits are directly set or reset through the system configuration software. All values in the command register are either registered or read-only. **Note:** The bus master enable bit must be set in the command register before the initiator can access the bus. The I/O access enable bit and/or the memory access enable bit must be set in the command register before the target will respond. 
CSR[31:16]	out	Provides access to the status register state bits. These are set automatically by the interface. Individual status bits are reset by the system software by writing a '1' to the bit location to be reset. All values in the status register are either registered or read-only. 
CSRX[47:0]	out	Provides access to the PCI-X extended capability registers. CSRX[47:16] is the PCI-X Status register, and CSRX[15:0] is the PCI-X Command register.
CSR_SYSTEMERR_N	in	Provides a mechanism for the user application to assert the system error signal, SERR_IO at its own discretion. Asserting this signal sets the relevant bits in the device status register.

Table 2-2: **User Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
CSRX_UNEXPECTED	in	Provides a mechanism for the user application to set the unexpected split completion status bit in the PCI-X device status register. This signal should be used when a received completion tag does not match any of the outstanding tags.
CSRX_DISCARDED	in	Provides a mechanism for the user application to set the discarded split completion status bit in the PCI-X device status register. This signal should be used when an attempted completion is rejected by the requester -- usually due to a master abort or target abort.
Target Signals		
S_DATA_OUT[63:0]	out	Target address / data / attribute. The S_DATA_OUT bus is qualified by control signal outputs described below. In 32-bit versions of this interface, this bus is half as wide.
S_CBEA_OUT[7:0]	out	Target command / byte enable / attribute. The S_CBEA_OUT bus is qualified by control signal outputs described below. In 32-bit versions of this interface, this bus is half as wide.
S_DATA_IN[63:0]	in	This bus is the target data input from the user application. In response to an initiator request, the target portion of the user application presents data to the bus interface by driving S_DATA_IN. In 32-bit versions of this interface, this bus is half as wide.
S_ADDL_VLD	out	When asserted, indicates that the low 32-bits of a valid target address are currently driven on S_DATA_OUT[31:0]. Also indicates that a valid bus command is currently driven on S_CBEA_OUT[3:0].
S_ADDH_VLD	out	When asserted, indicates that the high 32-bits of a valid target address are currently driven on S_DATA_OUT[31:0].Also indicates that a valid bus command is currently driven on S_CBEA_OUT[3:0].
S_ATTR_VLD	out	When asserted, indicates that the transfer attributes are currently driven on S_DATA_OUT[31:0] and S_CBEA_OUT[3:0]. This signal is only asserted when the interface is operating in PCI-X mode.

Table 2-2: **User Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
S_DATL_VLD	out	During target writes, the assertion of this signal indicates that valid target data is present on S_DATA_OUT[31:0] for 32-bit data objects in the user application. Also indicates that valid byte enables are present on S_CBEA_OUT[3:0] for 32-bit data objects. When sourcing data during target reads as a 32-bit data object, this signal indicates successful data transfer on the PCI-X bus.
S_DATH_VLD	out	During target writes, the assertion of this signal indicates that valid target data is present on S_DATA_OUT[63:0] for 64-bit data objects in the user application. Also indicates that valid byte enables are present on S_CBEA_OUT[7:0] for 64-bit data objects. When sourcing data during target reads as a 64-bit data object, indicates successful data transfer on the PCI-X bus. Not present in 32-bit versions of this interface.
S_DATA_NXT	out	During target reads, the assertion of this signal indicates the user application must provide the next piece of data during the next clock cycle. For 32-bit data objects, drive the next piece of 32-bit data on S_DATA_IN[31:0]. For 64-bit data objects, drive the next piece of 64-bit data on S_DATA_IN[63:0].
S_TABORT	in	Used by the user application to terminate the current target transaction with a target abort.
S_RETRY	in	Used by the user application to terminate the current target transaction with a retry. Do not use S_RETRY to signal a disconnect once data transfer has taken place.
S_DISCON	in	Used by the user application to terminate the current target transaction with disconnect with data.
S_NXTADB	in	Used by the user application to terminate the current target transaction at the next allowable disconnect boundary. This signal has no effect when the interface is operating in PCI mode.
S_WAIT	in	Used by the user application to insert initial wait states at the beginning of a target transaction. Must not be used once data transfer has taken place. The interface does not support wait states in the middle of a transfer in either bus mode.
S_SPLIT	in	Used by the user application to terminate the current target transaction with a split response. This signal has no effect when the interface is operating in PCI mode.
S_WRITE	out	This output signal indicates the direction of the current target transaction.

Table 2-2: **User Interface Signals** *(Continued)*

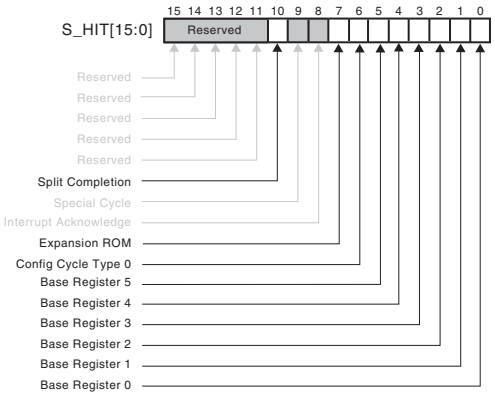
Signal Name	Type	Functional Description
S_DONE	out	This output signal is asserted for one cycle to mark the end of the current target transaction.
S_HIT[15:0]	out	This bus indicates to the user application that one of the address decoders has detected a match. As a result, the interface will respond as a target. 
Initiator		
M_DATA_OUT[63:0]	out	The initiator data output bus to the user application. In 32-bit versions of this interface, this bus is half as wide.
M_CBEA_IN[7:0]	in	The initiator command / byte enable / attribute input bus from the user application. In 32-bit versions of this interface, this bus is half as wide.
M_DATA_IN[63:0]	in	The initiator address / data / attribute input bus from the user application. In 32-bit versions of this interface, this bus is half as wide.
M_ADDL_VLD	out	Asserted by the interface before starting a bus transaction. When asserted, the user application must drive the low half of the bus address on M_DATA_IN[31:0].
M_ADDH_VLD	out	Asserted by the interface before starting a bus transaction. When asserted, the user application must drive the high half of the bus address on M_DATA_IN[63:32], unless the initiator has been configured for 32-bit operation only. In that case, drive the high half of the bus address on M_DATA_IN[31:0].
M_ATTR_VLD	out	Asserted by the interface before starting a bus transaction. When asserted, the user application must drive the transfer attribute information on M_DATA_IN[31:0] and M_CBEA_IN[3:0].

Table 2-2: **User Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
M_DATL_VLD	out	During initiator reads, the assertion of M_DATL_VLD indicates that valid initiator data is present on M_DATA_OUT[31:0] for 32-bit data objects in the user application. When sourcing data during initiator writes as a 32-bit data object, this signal indicates successful data transfer on the PCI-X bus.
M_DATH_VLD	out	During initiator reads, the assertion of this signal indicates that valid initiator data is present on M_DATA_OUT[63:0] for 64-bit data objects in the user application. When sourcing data during initiator writes as a 64-bit data object, it indicates successful data transfer on the PCI-X bus. Not present in 32-bit versions of this interface.
M_DATA_NXT	out	During initiator writes, the assertion of this signal indicates that the user application must provide the next piece of data during the next clock cycle. • For 32-bit data objects, drive the next piece of 32-bit data on M_DATA_IN[31:0]. • For 64-bit data objects, drive the next piece of 64-bit data on M_DATA_IN[63:0]. • For BURST commands that require byte enables, drive the byte enables with the data on M_CBEA_IN[3:0] or M_DATA_IN[7:0], as appropriate.
M_MABORT	out	When asserted, indicates the attempted initiator transaction was terminated due to a master abort condition.
M_RETRY	out	When asserted, indicates the attempted initiator transaction was terminated due to a disconnect without data. In PCI-X mode, this can only be a retry. In PCI mode, this may be a retry or a disconnect without data.
M_NXTADB	out	When asserted, indicates the attempted initiator transaction was terminated due to a disconnect at an allowable disconnect boundary.
M_DISCON	out	When asserted, indicates the attempted initiator transaction was terminated due to a disconnect with data. In PCI-X mode, this can only be a single data phase disconnect. In PCI mode, this can be any disconnect with data.
M_TABORT	out	When asserted, indicates the attempted initiator transaction was terminated due to a target abort.
M_SPLIT	out	When asserted, indicates the attempted initiator transaction was terminated due to a split response.

Table 2-2: **User Interface Signals** *(Continued)*

Signal Name	Type	Functional Description
M_FINISH	in	When asserted during an initiator transaction, causes the interface to finish the current transaction in a premature manner.
M_DONE	out	This output signal is asserted for one cycle to mark the end of the current initiator transaction.
M_REQ	in	The assertion of this signal by the user application indicates a request to begin an initiator bus transaction. This signal is not disabled by the busmaster enable bit in the command register. A device may always initiate split completion transactions, but must defer other transactions while the busmaster enable bit is cleared.

Configuration Map Signals

The configuration map signals are informational and allow the user to verify user settings established in the Xilinx CORE Generator tool. In addition, advanced users may potentially use these signals to drive options in the user application based on the configuration of the Initiator/Target core for PCI, promoting design reuse.

Table 2-3 describes the configuration map signals available to the user application. Any bits not listed are reserved. Note that these signals do *not* appear in the port diagram in Figure 2-1.

For more information on the functional descriptions, consult the *PCI Local Bus Specification Rev. 3.0* and *PCI-X Protocol Addendum Rev. 2.0a*.

Table 2-3: **Configuration Map Signals**

Signal Name	Type	Functional Description
Device Identification		
CFG[15:0]	out	Vendor ID
CFG[31:16]	out	Device ID
CFG[39:32]	out	Revision ID
CFG[63:40]	out	Class Code: • CFG[63:56] – Base Class • CFG[55:48] – Sub-Class • CFG[47:40] – Interface
CFG[303:288]	out	Subsystem Vendor ID
CFG[303:288]	out	Subsystem Revision ID
BARs		
CFG[95:64]	out	BAR 0 Size/Type, bits map to BAR 0 bits 31-0: • CFG[95:68] – '1' indicates the BAR bit is writable, '0' indicates the bit is not writable. Set bits are contiguous from CFG[95] downward. If only bit CFG[95] is set, the BAR size is 2 GB; if only bits CFG[95:94] are set, the size is 1 GB; etc. • CFG[67] – If a memory space, '1' indicates BAR is prefetchable. If an I/O space, this bit is reserved. • CFG[66:65] – If a memory space, '00' indicates a 32-bit address space, '10' indicates a 64-bit address space and a pairing with BAR 1 (CFG[127:96]). If an I/O space, these bits are reserved. • CFG[64] – '1' indicates a memory space, '0' indicates an I/O space.
CFG[408]	out	BAR 0 64-bit User Interface – '1' indicates that BAR 0 accepts 64-bit transactions (width of the data, not to be confused with the width of the BAR address).
CFG[127:96]	out	BAR 1 Size/Type, bits map to BAR 1 bits 31-0. Bit map is similar to that of CFG[95:64], unless paired with BAR 0 to create a 64-bit BAR.
CFG[409]	out	BAR 1 64-bit User Interface – '1' indicates that BAR 1 accepts 64-bit transactions.
CFG[159:128]	out	BAR 2 Size/Type, bits map to BAR 2 bits 31-0. Bit map is similar to that of CFG[95:64]. May be paired with BAR 3 to create a 64-bit BAR

Table 2-3: **Configuration Map Signals** *(Continued)*

Signal Name	Type	Functional Description
CFG[410]	out	BAR 2 64-bit User Interface – '1' indicates that BAR 2 accepts 64-bit transactions.
CFG[191:160]	out	BAR 3 Size/Type, bits map to BAR 3 bits 31-0. Bit map is similar to that of CFG[95:64], unless paired with BAR 2 to create a 64-bit BAR.
CFG[411]	out	BAR 3 64-bit User Interface – '1' indicates that BAR 3 accepts 64-bit transactions.
CFG[223:192]	out	BAR 4 Size/Type, bits map to BAR 4 bits 31-0. Bit map is similar to that of CFG[95:64]. May be paired with BAR 5 to create a 64-bit BAR
CFG[412]	out	BAR 4 64-bit User Interface – '1' indicates that BAR 4 accepts 64-bit transactions.
CFG[255:224]	out	BAR 5 Size/Type, bits map to BAR 5 bits 31-0. Bit map is similar to that of CFG[95:64], unless paired with BAR 4 to create a 64-bit BAR.
CFG[413]	out	BAR 5 64-bit User Interface – '1' indicates that BAR 5 accepts 64-bit transactions.
CFG[351:320]	out	Expansion ROM BAR Size, bits map to Expansion ROM Address bits 31-0: • CFG[351:331] – '1' indicates the BAR bit is writable, '0' indicates the bit is not writable. Set bits are contiguous from CFG[351] downward. If only bit CFG[351] is set, the BAR size is 2 GB; if only bits CFG[351:350] are set, the size is 1 GB; etc. • CFG[330:321] – Reserved • CFG[320] – Expansion ROM Address enable
CFG[414]	out	Expansion ROM BAR 64-bit User Interface – '1' indicates that the BAR accepts 64-bit transactions.
Message-Signaled Interrupts		
CFG[488:486]	out	Multiple Message Capable – Settings are: • 000 – 1 message requested • 001 – 2 messages requested • 010 – 4 messages requested • 011 – 8 messages requested • 100 – 16 messages requested • 101 – 32 messages requested • 110 and 111 – Reserved

Table 2-3: **Configuration Map Signals** *(Continued)*

Signal Name	Type	Functional Description
Power Management		
CFG[483]	out	PME D1 – '1' if D1 power state is supported.
CFG[484]	out	PME D2 – '1' if D2 power state is supported.
CFG[454]	out	PME D3 – '1' if D3 power state is supported.
Split Transactions		
CFG[415]	out	Split Transaction Width – '1' if 64-bit Split Transactions are supported
CFG[499:497]	out	Maximum Outstanding Split Transactions: • 000 – 1 supported outstanding split transaction • 001 – 2 supported • 010 – 3 supported • 011 – 4 supported • 100 – 8 supported • 101 – 12 supported • 110 – 16 supported • 111 – 32 supported
Extended Capabilities		
The Initiator/Target core for PCI-X specifically fixes the following capabilities linked-list data structures: • 40h – Power Management Event (PME) capability data structure • 48h – Message-Signaled Interrupt (MSI) capability data structure • 58h – PCI-X capability data structure Each register below is a pointer to the next data structure in the linked list as defined above. If the next data structure is user-specified, the pointer is set to 80h-FFh as specified in the CORE Generator tool. If no further data structures exist, the pointer is set to 00h.		
CFG[359:352]	out	Initial Capability Pointer – Points to the first or only extended capability data structure in the linked list.
CFG[383:376]	out	PME Next Capability Pointer – Points to the data structure in the linked list *following* the PME capability data structure.
CFG[391:384]	out	MSI Next Capability Pointer – Points to the data structure in the linked list *following* the MSI capability data structure.

Table 2-3: **Configuration Map Signals** *(Continued)*

Signal Name	Type	Functional Description
CFG[399:392]	out	PCI-X Next Capability Pointer – Points to the data structure in the linked list *following* the PCI-X capability data structure.
Bus Arbitration		
CFG[367:360]	out	*MIN_GNT* – Requested minimum burst length
CFG[375:368]	out	*MAX_LAT* – Requested maximum time between bus grants
Miscellaneous Settings		
CFG[287:256]	out	CardBus CIS Pointer
CFG[489]	out	Bus Width – Set to '1' as a 64-bit core
CFG[490]	out	PCI-X 133 MHz Capable – '1' if 133 MHz capable
CFG[491]	out	Device Complexity – '1' if a bridge device, '0' if a simple device

XILINX®

Chapter 3

General Design Guidelines

This chapter provides guidelines for turning a Initiator/Target core for PCI-X into a fully functioning design integrated with user application logic. A target-only design does not require any of the initiator steps unless the design requires the use of split transactions; however, an initiator always requires the target interface. The burst support steps may require four separate sub-steps—read and write operations for both target and initiator.

Design Steps

- Configure the Base Address Register(s)
- Configure the contents of the Configuration Space Header ROM
- Configure the core interface options

Target Designs

- Build an interface to read and write locations in the user application. See Chapter 5, "Basic Target Transaction Control."
- Create logic to signal various target termination conditions if required by the user application. See Chapter 6, "Advanced Target Transaction Control."

Initiator Designs

- Build an initiator control state machine and the required support logic, and an interface to read and write locations in the user application. See Chapter 8, "Basic Initiator Transaction Control."
- Create logic to control initiator data phases. See Chapter 9, "Advanced Initiator Transaction Control."

Burst Designs

- Provide pipelined data sources that correctly respond to various target and initiator termination conditions, and build an address counter. See Chapter 7, "Target Burst Transfers," and Chapter 10, "Initiator Burst Transfers."
- Build FIFOs for the specific application, if required.

Advanced Designs

- Implement configuration space registers to support additional features and capabilities, if required. See Chapter 11, "Other Bus Cycles."

- Modify the initiator or target logic to support special bus commands, if desired. See Chapter 11, "Other Bus Cycles."

Know the Degree of Difficulty

A fully compliant core interface is challenging to implement in any technology and especially so in FPGA devices.

Table 3-4 provides information about the degree of difficulty in implementing various PCI-X designs, which is sharply influenced by:

- Maximum system clock frequency
- Targeted device architecture
- Nature of the user application

All PCI-X implementations need careful attention to system performance requirements. Pipelining, logic mapping, placement constraints, and logic duplication are all methods that help boost system performance.

Table 3-4: **Degree of Difficulty for Various PCI Implementations**

Device Family	Implementation	Difficulty
Virtex-5	PCI mode, 33 MHz	Easy
	PCI-X mode, 66 MHz	Easy
	PCI-X mode, 133 MHz	Moderate

Keep it Registered

To simplify timing and increase system performance in an FPGA design, keep everything registered, that is, all inputs and outputs from the user application should come from, or connect to, a synchronous element such as a flip-flop or block RAM. While registering signals may not be possible for all paths, it increases performance.

Make Only Allowed Modifications

The core interface is not user-modifiable. Do not make modifications as they may have adverse effects on system timing and PCI-X protocol compliance. All modifications to the core interface must be completed using CORE Generator software. In addition, do not modify the wrapper files provided with the core.

Unsupported Devices

If you wish to target a device/package combination that is not officially supported (not listed in the *Initiator/Target for PCI-X Data Sheet*), you may use the UCF Generator for PCI/PCI-X to create a user constraints file that implements a suitable pinout for your target device. This tool is available in the Xilinx CORE Generator tool under **UCF Generator for PCI/PCI-X**.

For more information on this tool, consult the *UCF Generator for PCI/PCI-X Data Sheet*.

Note: It is important to verify the UCF files generated by this tool to confirm that the timing

requirements of your application are met. Xilinx cannot guarantee that every UCF file generated by the UCF Generator tool will work for every application.

Chapter 4

Initialization and Interoperability

This chapter discusses a number of important concerns involving initialization and interoperability. It is helpful to know in advance the type of system you want: an open system that requires full compliance, or for an embedded system where you may have significant control over the system and the bus, including knowledge of the bus mode(s) and bus width(s).

Device Initialization

The *PCI Local Bus Specification* defines a parameter Tpvrh, of 100 ms—the minimum time allowed from "power valid" to "reset high." Power valid is when the bus power rails are within their specified limits. Reset high is of interest because at the de-assertion of RST#, some information is broadcast that tells PCI and PCI-X devices about the bus. Bus width and bus mode information is communicated at this time, and most devices must have this information to function properly. From power valid on the bus, there is limited time for:

1. Local power rails to become valid, if local regulation is used (generally the case)
2. FPGA to do its preconfiguration housecleaning process
3. FPGA to load its bitstream
4. FPGA to enter user mode

Considering all the factors, there is less than 100 ms for the FPGA device configuration data to transfer. It is therefore necessary to transfer configuration data as quickly as possible using a fast and wide FPGA configuration mode. It is important to understand this requirement for a successful board design using the core interface.

Note: If you are designing an embedded system, and have control over the bus reset, the need for high speed FPGA configuration may be eliminated by providing adequate time from power valid to reset deassertion.

Design Initialization

Immediately after FPGA configuration, both the core interface and the user application are initialized by the startup mechanism present in all Xilinx FPGA devices.

Bus Reset

During normal operation, the assertion of RST# on the PCI-X bus re-initializes the core interface and three-state all PCI-X bus signals. This behavior is fully compliant with the *PCI Local Bus Specification*. The core interface is designed to correctly handle asynchronous resets.

Typically, the user application must be initialized each time the core interface is reset. In this case, use the RST output of the core interface as the asynchronous reset signal for the user application.

If part of the user application requires an initialization capability that is unrelated to PCI-X bus resets, simply design the user application with a separate reset signal.

Note that these reset schemes require the use of routing resources to distribute reset signals, since the global resource is not used. The use of the global reset resource is not recommended.

Bus Clock

The bus clock output provided by the core interface is derived from the bus clock input, and is distributed using a clock buffer. The interface itself is fully synchronous to this clock. In general, the portion of the user application that communicates with the interface must also be synchronous to this clock.

The frequency of this clock is not guaranteed to be constant. In fact, in a compliant system, the clock may be any frequency, up to and including the maximum allowed frequency, and the frequency may change on a cycle-by-cycle basis. Under certain conditions, the core may also apply phase shifts to this clock.

For these reasons, the user application should not use this clock as an input to a DLL or PLL, nor should the user application use this clock in the design of interval timers (such as DRAM refresh counters).

Interoperability

As provided, the core interface may be used to generate two different designs; one design (FPGA programming file, or bitstream) that operates in PCI-X bus mode, and another that operates in PCI bus mode. This version of the core does not currently support dynamic bus mode changes. Changing bus mode requires reconfiguration of the FPGA. This requires the storage of two bitstreams and the implementation of a reconfiguration controller external to the FPGA.

For full compliance, a PCI-X device must also support PCI. However, in many embedded systems, it is possible to reduce the burden of this requirement by only implementing support for the bus mode(s) that will actually be in use.

Bus Width Detection

A core interface that provides a 64-bit datapath needs to know if it is connected to a 64-bit bus or a 32-bit bus. The core interface is capable of sensing and adjusting to the bus width automatically. However, this behavior can be manually forced through the `BW_DETECT_DIS` and `BW_MANUAL_32B` ports.

Bus Mode Detection

A core interface that provides backward PCI compatibility must determine whether it is in PCI-X bus mode or PCI bus mode. The core is capable of sensing bus mode automatically. Adjustments must be made through reconfiguration of the FPGA.

As described above, a fully compliant design requires two bitstreams and the ability to reconfigure the FPGA after the bus mode is detected. The RTR output of the core interface will assert following the deassertion of the bus reset signal if the interface recognizes that

the incorrect bitstream is in use. When this occurs, external circuitry is responsible for re-initializing the FPGA and loading an alternate bitstream. This requires storage for two complete bitstreams and another device, such as a CPLD, for managing the reconfiguration process. Reconfiguration cannot be controlled by the FPGA because the FPGA becomes inactive during configuration.

The bitstream loaded in response to RTR becomes active after the bus reset. The design will not be present to observe the bus mode and bus width broadcast. Missing the bus mode broadcast is not an issue, as the newly loaded bitstream will be correct for the bus mode in use. However, the newly loaded bitstream cannot detect if the bus is 32-bit or 64-bit. Upon the assertion of RTR, the FPGA must save the bus width state in the CPLD so that the CPLD can restore it later.

Bus width is visible on the PCIW_EN signal when BW_DETECT_DIS is set to logic zero, even if this signal were previously set to logic one. Bus width may be forced by setting this signal to logic one and then setting BW_MANUAL_32B appropriately. While single-mode implementations will set these signals to permanent values, dual-mode implementations require the user application to control these signals. This is a board level design requirement. The exact implementation depends on the configuration method used.

Bus Clock Speed Detection

In PCI-X bus mode, a host is required to keep the bus clock speed fixed after bus reset, though the host may choose any fixed clock frequency between 50 MHz and 133 MHz. At initialization, the host broadcasts the speed at which the clock will run.

Note: More specifically, the host categorizes the bus clock speed into one of three ranges: 50-66 MHz, 66-100 MHz or 100-133 MHz.

Since the bus clock runs at a fixed frequency, the core wrapper uses a Digital Clock Manager (DCM) to derive the internal clock of the core. The Virtex-5 DCM has two operating frequency modes, Low Frequency Mode and High Frequency Mode, which are separated by a 120 MHz boundary in the -1 speed grade. A PCI-X design configures the DCM to run in Low Frequency Mode, giving it an operating range of 32-120 MHz. If the bus clock frequency initializes to a range that extends beyond the 120 MHz upper boundary, the core wrapper divides the DCM internal clock frequency in half by changing the CLKIN_DIVIDE_BY_2 attribute through its Dynamic Reconfiguration Port (DRP). The DCM DRP is clocked using the REF_I reference clock.

Note: The wrapper also changes the CLKFX_MULTIPLY and CLKFX_DIVIDE attributes to double the internal clock so that the DCM output matches the original bus clock frequency

Dynamic reconfiguration is only required in fully compliant PCI-X 133 MHz designs. PCI-X 66 MHz designs do not use the DCM DRP, and consequently do not use REF_I.

Note: If you are designing an embedded system and only operate the PCI-X bus at a single clock frequency, DRP is not required and the REF_I reference clock may be eliminated. For more details, see Xilinx Answer Record #30518 at www.xilinx.com/support/answers/30518.htm.

Configuration Space

By default, the core interface implements a complete type zero configuration space header. The first 64 bytes are used for the standard header; the next 64 bytes are reserved for the following capability items present in the core interface:

- PCI-X Capability Item
- Message Signaled Interrupt Capability Item

- Power Management Capability Item

The remaining 128 bytes are under the control of the user application. Typically, the user application should simply return zero on all configuration accesses. However, the user application may implement additional capability items in this region, or implement proprietary registers that are outside the scope of the specification.

Table 4-1: **Configuration Space Header for PCI-X**

31 16	15 0	
Device ID	Vendor ID	00h
Status	Command	04h
Class Code	Rev ID	08h
BIST / Header Type	Latency Timer / Cache Line Size	0Ch
Base Address Register 0 (BAR0)		10h
Base Address Register 1 (BAR1)		14h
Base Address Register 2 (BAR2)		18h
Base Address Register 3 (BAR3)		1Ch
Base Address Register 4 (BAR4)		20h
Base Address Register 5 (BAR5)		24h
Cardbus CIS Pointer		28h
Subsystem ID	Subsystem Vendor ID	2Ch
Expansion ROM Base Address		30h
Reserved	CapPtr	34h
Reserved		38h
Max Lat / Min Gnt	Interrupt Pin / Interrupt Line	3Ch
Power Management Capability	NxtCap / PM Cap	40h
Data / PMCSR BSE	PMCSR	44h
Message Control	NxtCap / MSI Cap	48h
Message Address		4Ch
Message Upper Address		50h
Reserved	Message Data	54h
PCI-X Command	NxtCap / PCI-X Cap	58h
PCI-X Status		5Ch
Reserved		60h-7Fh
Available User Configuration Space		80h-FFh

Note: Shaded areas are not implemented and return zero.

Shaded address locations are not implemented and return zero, with the exception of the user configuration space which returns values provided by the user application. Additional addresses may not be implemented depending on the configuration of the interface selected when the design is generated. For example, locations such as the CapPtr, or Base Address Registers, will return zeros if disabled.

Chapter 5

Basic Target Transaction Control

This chapter discusses the logic required to generate the load and output select signals for a typical target register in the user application.

In applications not using target burst transactions, data is usually transferred to and from registers in the user application. These registers are connected to control signals required for target data transfer and to any additional control and data path logic provided by the user. These registers may also connect to internal FIFOs or to I/O pins on the user application. Figure 5-1 shows a typical interfaced target register.

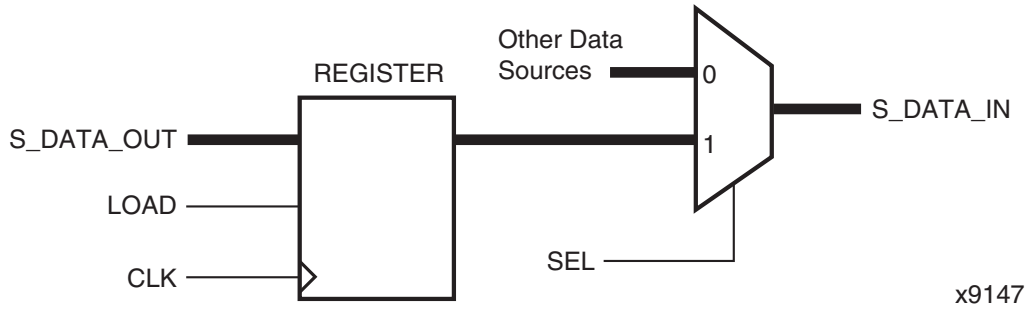


Figure 5-1: **Example Target Register**

Target Transaction Overview

The process of servicing a transaction where the user application is the target involves several steps:

1. Storing transaction attributes and parameters.
2. Identifying incoming transactions.
3. Sending or receiving data on the first data phase.
4. Sending or receiving data on subsequent data phases.

This chapter focuses on the first three steps and describes how to build a target user application that handles single-data phase transactions. Subsequent chapters explain how this mechanism is extended to handle burst transactions.

Storing Transaction Information

When a transaction starts on the PCI-X bus, the core interface sends information to the user application using the following signals:

- **S_DATA_OUT[63:0]**: Output bus provides the means for address, attribute, and data transfer from the core interface to the user application.
- **S_CBEA_OUT[7:0]**: Output bus provides the means for command, byte enable, and attribute transfer from the core interface to the user application.
- **S_ADDL_VLD**: Output indicates that a valid address is available on S_DATA_OUT[31:0], and may be used as a clock enable by the user application to capture a copy of this address. Note that the assertion of S_ADDL_VLD does *not* mean that the user application will be the selected target; it simply means that a transaction has begun on the PCI-X bus, and that its address is available on S_DATA_OUT[31:0].

  The S_HIT bus tells the user application whether it is the target of the transaction. The S_ADDL_VLD signal is asserted for a single cycle, during which time S_CBEA_OUT[3:0] reflects a valid bus command, and S_CBEA_OUT[7] indicates the width of the requested bus transfer based on REQ64_IO.
- **S_ADDH_VLD**: When a 64-bit address is broadcast to the target using the dual address cycle command, this output indicates that a valid upper 32 bits of a 64-bit address is available on S_DATA_OUT[31:0], and a valid bus command is present on S_CBEA_OUT[3:0]. As with S_ADDL_VLD, this signal tells the user application only that an address is available, not whether that address belongs to the user application.
- **S_ATTR_VLD**: Output indicates that valid attribute information (whose format is specified by the *PCI-X Addendum*) is available on S_DATA_OUT[31:0] and S_CBEA_OUT[3:0]. This signal is never asserted in PCI mode.

When S_ADDL_VLD, S_ADDH_VLD, or S_ATTR_VLD is asserted, the user application must latch the corresponding transaction parameters from S_DATA_OUT[31:0] and S_CBEA_OUT[3:0] if required by the user application. This must occur before the core interface has determined whether the user application is the intended target. The following example code demonstrates how to do this:

```
reg     [3:0] scmd;
reg    [35:0] sattr;
reg    [63:0] saddr;

always @(posedge CLK or posedge RST)
begin : slaveinfo
  if (RST)
  begin
    scmd <= 4'b0000;
    sattr <= 36'h000000000;
    saddr[31:0] <= 32'h00000000;
    saddr[63:32] <= 32'h00000000;
  end
  else
  begin
    // Always grab the low address.
    if (S_ADDL_VLD)
    begin
      saddr[31:0] <= S_DATA_OUT[31:0];
    end
    // Grab the command, direction, and high address.
    if (S_ADDL_VLD)
    begin
      saddr[63:32] <= 32'h00000000;
      scmd <= S_CBEA_OUT[3:0];
    end
    else if (S_ADDH_VLD)
    begin
```

```
            saddr[63:32] <= S_DATA_OUT[31:0];
            scmd <= S_CBEA_OUT[3:0];
          end
          // Always grab the attribute.
          if (S_ATTR_VLD)
          begin
            sattr <= {S_CBEA_OUT[3:0], S_DATA_OUT[31:0]};
          end
        end
      end
```

Figure 5-2 shows the signaling profile for the start of a transaction targeting a 32-bit address.

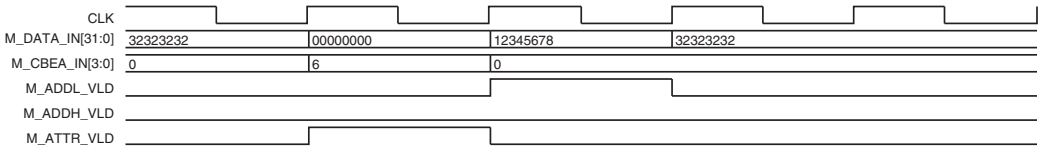


Figure 5-2: **Receiving a 32-Bit Address**

Figure 5-3 shows the signaling profile for the start of a transaction targeting a 64-bit address.

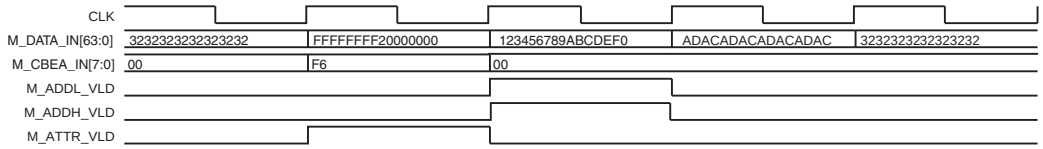


Figure 5-3: **Receiving a 64-Bit Address**

Identifying Incoming Transactions

Each time an address is broadcast on the bus, the core interface compares that address to a list of potential targets in the core interface. Potential targets include address spaces defined by the base address registers, and other targets such as a split completion target or internal configuration space.

When the core interface determines that it is the target of the current bus transaction, it asserts one of the bits of the `S_HIT[15:0]` bus for one clock cycle. This is the first indicator to the user application that a target transaction is about to begin.

As shown in Figure 5-4, the lower six bits, S_HIT[5:0], are used to signal matches with one of the base address registers. The remaining upper bits signal matches with other potential targets in the interface.

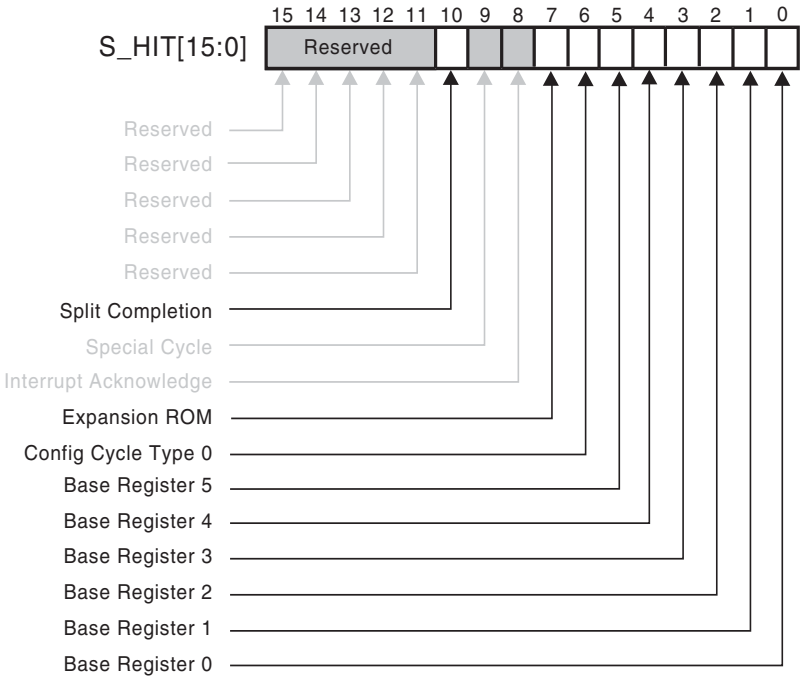


Figure 5-4: **S_HIT Interpretation**

The user application is responsible for monitoring outputs from the core interface to respond to target transactions. The signals used in target transactions are active and available at different times. The most important signal is S_HIT[x], which is asserted for a single cycle and indicates that the core interface has claimed the current PCI-X transaction.

The following logic demonstrates how to use **S_HIT[x]** to generate read and write transaction indicators, BAR_x_RD and BAR_x_WR. Whenever a particular transaction target is accessed, exactly one of these two indicator signals remains active through the entire transaction.

The sample code below introduces two more signals from the core interface:

- **S_WRITE**: Output that indicates whether the incoming transaction is a read (low) or a write (high) and is valid once the interface has determined the bus command.
- **S_DONE**: Output asserted for one clock cycle to mark the end of a transaction. Note in the example code how it is used to *turn off* BAR_x.

```
reg           BAR_x;

always @(posedge CLK or posedge RST)
begin : identify
  if (RST)
  begin
    BAR_x <= 1'b0;
  end
  else
  begin
    if (S_HIT[x]) BAR_x <= 1'b1;
```

```
    else if (S_DONE) BAR_x <= 1'b0;
  end
end
assign BAR_x_RD = (S_HIT[x] | BAR_x) & !S_WRITE;
assign BAR_x_WR = BAR_x & S_WRITE;
```

Sending or Receiving Data

Each potential target in the user application may be either a 32-bit or a 64-bit wide data object. While the core interface provides 64-bit wide connections to the user application, each target implemented in the user application may communicate with the core interface using a 32-bit data transfer or a 64-bit data transfer. Table 5-1 shows the potential targets and their data width.

Table 5-1: **Potential Targets and Data Width**

Potential Target	S_HIT[x]	Data Object
Reserved	15	Undefined
Reserved	14	Undefined
Reserved	13	Undefined
Reserved	12	Undefined
Reserved	11	Undefined
Split Completion	10	32-bit / 64-bit (user option)
Special Cycle	9	Not supported
Interrupt Acknowledge	8	Not supported
Expansion ROM	7	32-bit / 64-bit (user option)
Config Cycle Type 0	6	32-bit only
Base Register 5	5	32-bit / 64-bit (user option)
Base Register 4	4	32-bit / 64-bit (user option)
Base Register 3	3	32-bit / 64-bit (user option)
Base Register 2	2	32-bit / 64-bit (user option)
Base Register 1	1	32-bit / 64-bit (user option)
Base Register 0	0	32-bit / 64-bit (user option)

The core interface automatically aligns data and adjusts byte enables based on the addressed target.

Note: Base address registers, when paired to support 64-bit addressing, only use a single bit of the **S_HIT** bus. As a result, subsequent bits of S_HIT[5:0] are shifted down by one bit position. For instance, if all six registers were paired to create three, 64-bit base address registers, only S_HIT[2:0] are active, leaving S_HIT[5:3] permanently deasserted. The S_HIT[15:6] bits are not shifted.

When you design for a declared 32-bit wide potential target, all data transfer for that target between the interface and the user application is conducted over the low half of the data path, independent of what type of transfer takes place on the PCI-X bus. Likewise, when

you design for a declared 64-bit wide potential target, all data transfer for that target between the interface and the user application is conducted over the full width of the data path, independent of what type of transfer takes place on the PCI-X bus.

It is important to note that if the split completion data target is configured for 64-bit operation, the interface will properly align inbound data in most cases. However, for certain split completion transactions, such as split completion messages and DWORD completions, the bus address will be zero. In these cases, the split completion data target will always behave as if the data is QWORD aligned. This may require the user application to swap the single inbound DWORD to properly align it.

For a general example, you may elect to have a bank of 32-bit wide control and status registers (a register file) associated with base address register zero, and a 64-bit wide data FIFO associated with base address register one. In this case, the core interface should be configured so that BAR0 is a 32-bit wide data target and BAR1 is a 64-bit wide data target.

Target Write

During a target write operation, the user application captures data from the S_DATA_OUT bus. If the target user application performs only single data phase transactions, the data is typically captured in a register.

After a target write transaction is established (that is, through the BAR_x_WR decoding described in the previous section), valid data is available on S_DATA_OUT whenever one of the following two signals is asserted by the core interface:

- **S_DATL_VLD**: This output indicates the user application should capture valid 32-bit data from the S_DATA_OUT[31:0] bus, and is used only by potential targets that have been configured as 32-bit data objects. At this time, S_CBEA_OUT[3:0] indicates valid byte enables.
- **S_DATH_VLD**: This output indicates the user application should capture valid 64-bit data from the S_DATA_OUT[63:0] bus, and is used only by potential targets that have been configured as 64-bit data objects. At this time, S_CBEA_OUT[7:0] indicates valid byte enables.

If the user application supports byte-addressable registers, separate load signals must be generated for each byte in the register by further qualifying the expressions above with the byte-enables present on S_CBEA_OUT.

The following is an example of a 32-bit register that uses byte enables. The load control signal on the target register would be generated as follows:

```
assign LOAD = BAR_x_WR & S_DATL_VLD; //32-bit object
```

The actual register implementation could be described as shown:

```
reg    [31:0] my_reg;

always @(posedge CLK or posedge RST)
begin : write_my_reg
  if (RST) my_reg <= 32'h00000000;
  else if (LOAD)
  begin
    if (!S_CBEA_OUT[0]) my_reg[ 7: 0] <=
                           S_DATA_OUT[ 7: 0];
    if (!S_CBEA_OUT[1]) my_reg[15: 8] <=
                           S_DATA_OUT[15: 8];
    if (!S_CBEA_OUT[2]) my_reg[23:16] <=
                           S_DATA_OUT[23:16];
```

```
            if (!S_CBEA_OUT[3]) my_reg[31:24] <=
                                    S_DATA_OUT[31:24];
        end
    end
```

The time relationship is shown in the waveform of Figure 5-5 and in Figure 5-6. These waveforms include bus signals and internal user application signals.

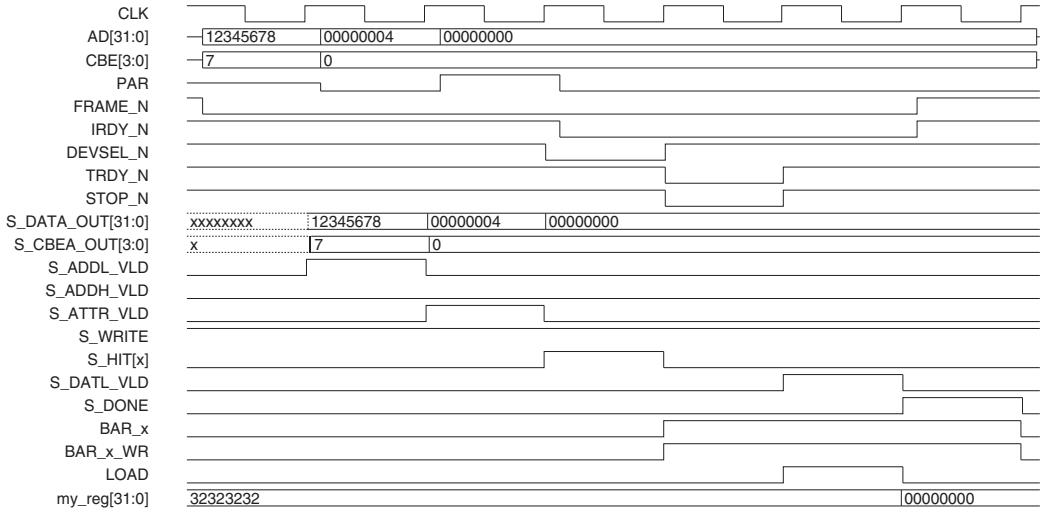


Figure 5-5: **Target Write Transaction to 32-bit Register for PCI-X**

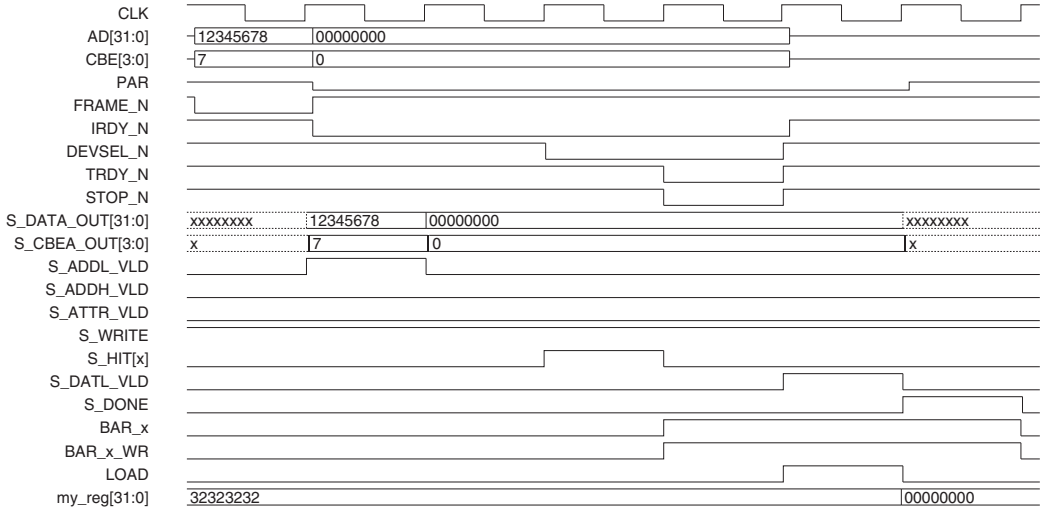


Figure 5-6: **Target Write Transaction to 32-bit Register for PCI**

The following is an example of a 64-bit register that uses byte enables. The load control signal on the target register would be generated as follows:

```
assign LOAD = BAR_x_WR & S_DATH_VLD; //64-bit object
```

The actual register implementation could be described as shown:

```
reg    [63:0] my_reg;

always @(posedge CLK or posedge RST)
begin : write_my_reg
```

```
if (RST) my_reg <= 64'h0000000000000000;
else if (LOAD)
begin
  if (!S_CBEA_OUT[0]) my_reg[ 7: 0] <=
                         S_DATA_OUT[ 7: 0];
  if (!S_CBEA_OUT[1]) my_reg[15: 8] <=
                         S_DATA_OUT[15: 8];
  if (!S_CBEA_OUT[2]) my_reg[23:16] <=
                         S_DATA_OUT[23:16];
  if (!S_CBEA_OUT[3]) my_reg[31:24] <=
                         S_DATA_OUT[31:24];
  if (!S_CBEA_OUT[4]) my_reg[39:32] <=
                         S_DATA_OUT[39:32];
  if (!S_CBEA_OUT[5]) my_reg[47:40] <=
                         S_DATA_OUT[47:40];
  if (!S_CBEA_OUT[6]) my_reg[55:48] <=
                         S_DATA_OUT[55:48];
  if (!S_CBEA_OUT[7]) my_reg[63:56] <=
                         S_DATA_OUT[63:56];
  end
end
```

The time relationship is shown in the waveform of Figure 5-7 and in Figure 5-8. These waveforms include bus signals and internal user application signals.

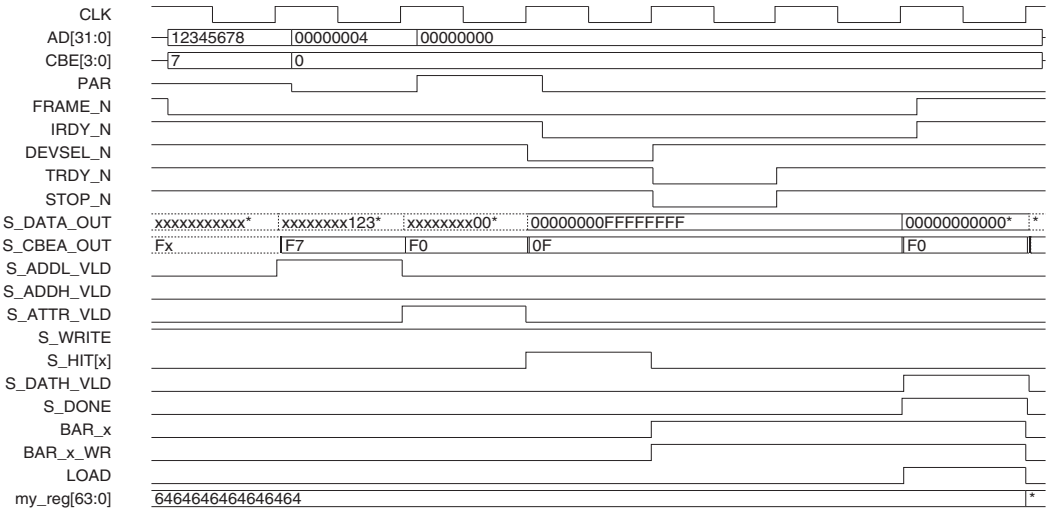


Figure 5-7: **Target Write Transaction to 64-bit Register for PCI-X**

Target Read

During a target read operation, the user application presents data to the core interface via the **S_DATA_IN** bus. If the target user application performs only single data phase transactions, the transaction typically involves a single register or a bank of registers, 32 or 64 bits wide.

Once a target read transaction is established (that is, through the BAR_x_RD described previously), valid data must be presented on S_DATA_IN. If the user application has more than one register, it must determine which register is being accessed and direct its output onto **S_DATA_IN** through a multiplexer:

```
assign SEL = fn (BAR_x_RD);
```

If the addressed target is a 32-bit data object, valid data need only appear on **S_DATA_IN[31:0]**. If the addressed target is a 64-bit data object, valid data must be presented on the entire width of the **S_DATA_IN[63:0]** bus.

Never qualify the output data with the **S_CBEA_OUT** signals even if the user application supports byte-addressable registers; drive the required width of the **S_DATA_IN** bus with valid data, even if the registers are non-prefetchable.

The time relationship is shown in the following waveforms (Figure 5-8, Figure 5-9, and Figure 5-10) which show both bus signals and user application signals.

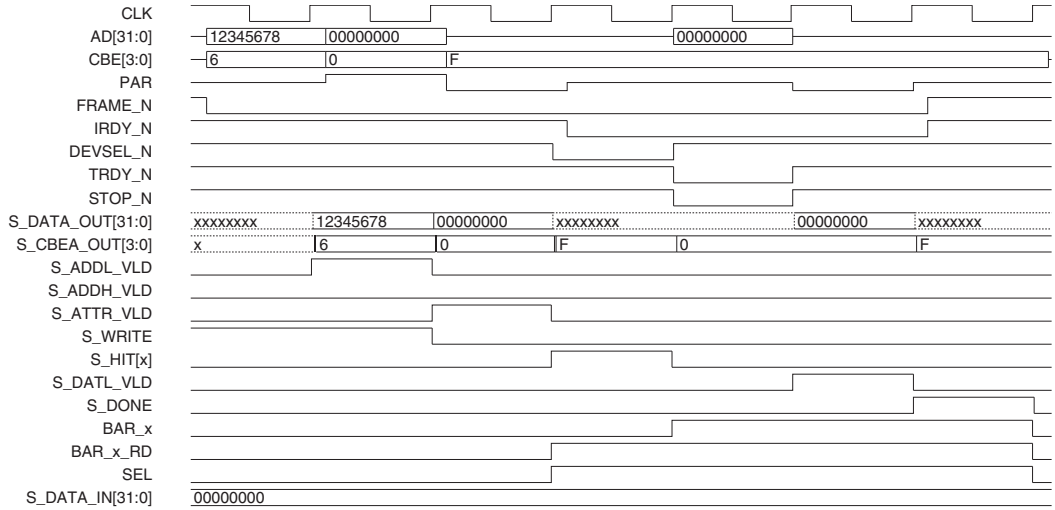


Figure 5-8: **PCI-X Target Read Transaction of 32-bit Register**

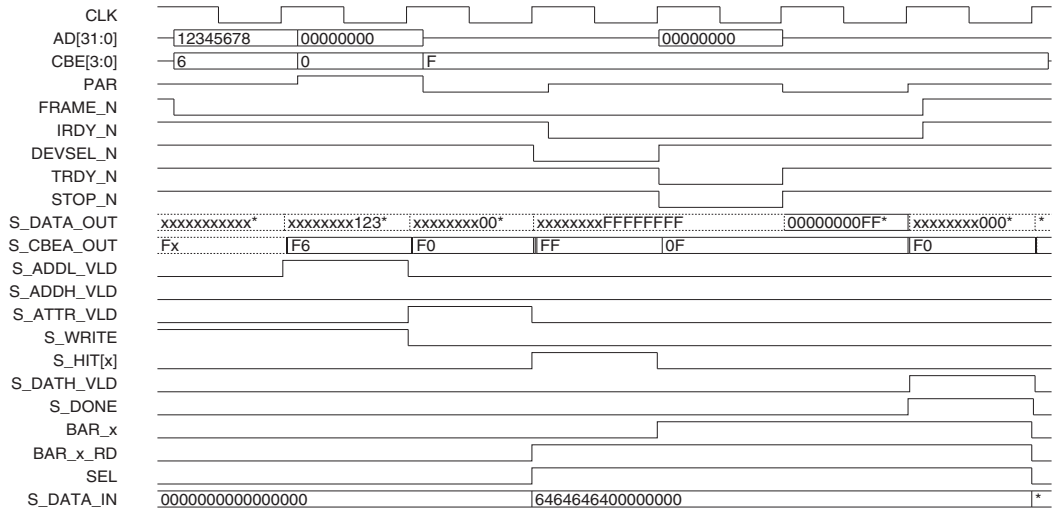


Figure 5-9: **PCI-X Target Read Transaction of 64-bit Register**

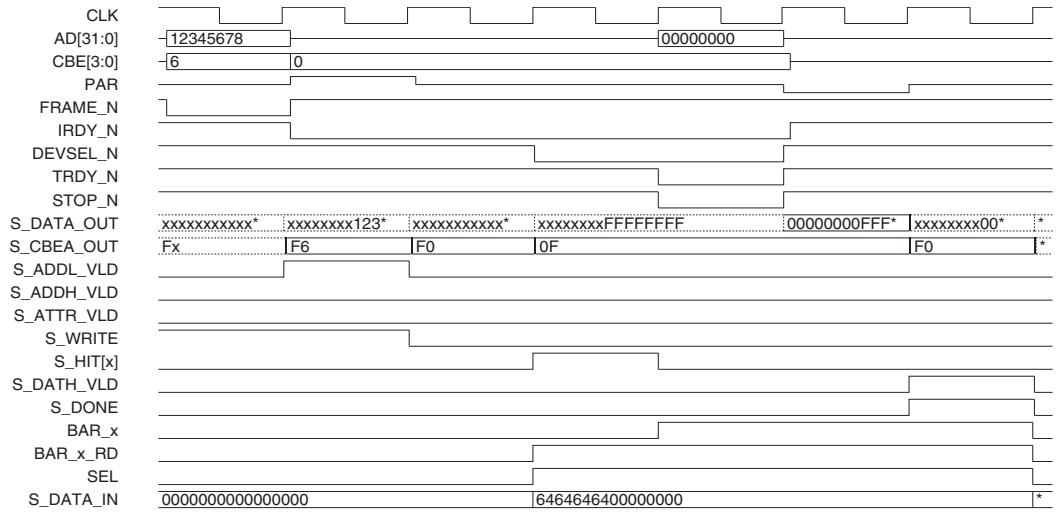


Figure 5-10: **PCI Target Read Transaction of 64-bit Register**

Terminating Target Transactions

In general, PCI-X transactions may be terminated by the initiator or the target. This section discusses simple disconnects for non-burst transactions; disconnects are discussed in Chapter 6, "Advanced Target Transaction Control."

The PCI-X protocol defines some bus commands as inherently single data phase transfers. These commands are listed below:

- I/O Read
- I/O Write
- Configuration Read
- Configuration Write
- Memory Read DWORD

In PCI, the Memory Read command is a BURST command, while in PCI-X, the comparable Memory Read DWORD is not. If the user application relies on the transaction having only one data phase, it should force a disconnect with data.

The core interface, while operating in PCI mode, will automatically disconnect with data on the first data phase for the commands listed above, with the exception of Memory Read. This enforces consistent behavior in all bus modes.

The remaining bus commands permit burst transfers. Again, if the user application does not support bursting to a particular target, the user application should force a disconnect with data.

Target terminations are controlled using the **S_DISCON**, **S_RETRY**, **S_NXTADB**, **S_WAIT**, **S_SPLIT**, and **S_TABORT** signals. To force disconnect on the first data phase for all transactions, set the control signals as follows:

```
assign S_DISCON  = 1'b1;
assign S_RETRY   = 1'b0;
assign S_NXTADB  = 1'b0;
assign S_WAIT    = 1'b0;
```

```
assign S_SPLIT   = 1'b0;
assign S_TABORT  = 1'b0;
```

This will cause all PCI and PCI-X target transactions to terminate after a single data phase. This termination behavior is exhibited in the target write and read transactions shown in this chapter.

XILINX®

Chapter 6

Advanced Target Transaction Control

This chapter describes the methods by which the user application can control target transactions to accommodate its own ability to source or sink data, including

- Target-inserted wait states provide the user application additional time at the start of a transaction to prepare itself for the first data transfer.
- Target terminations allow the user application to limit the number of data phases in a transaction, which is useful in both burst and non-burst target designs.

Control Modes

Data phase control is achieved using the S_WAIT, S_NXTADB, S_DISCON, S_RETRY, S_SPLIT, and S_TABORT signals. These will be collectively referred to as the control signals in this document. When none of these signals are asserted, the interface allows bus data phases to complete without the insertion of extra wait states or termination by the target. Assertion of one of these control signals yields one of the following modes:

- **Wait**. When S_WAIT is asserted, the interface inserts wait states at the beginning of a bus transaction (holds off the first data phase) by delaying the assertion of **TRDY_IO**. This can be used with PCI or PCI-X, subject to the initial data latency rules on wait states.
- **Disconnect on Next ADB**. Assertion of S_NXTADB causes the interface to terminates the current bus transaction at the next Allowable Disconnect Boundary. This should only be used in PCI-X mode. When the interface is operating in PCI mode, this signal has no effect.
- **Disconnect with Data**. Assertion of S_DISCON terminates the current bus transaction. In PCI-X mode, this type of disconnect may only be used on the first data phase. In PCI mode, this may be used at any time.
- **Retry**. When S_RETRY is asserted, the interface will signal a retry. This can be done in either bus mode, but must only be done on the first data phase.
- **Split Response**. Similar to a retry, assertion of S_SPLIT causes the interface to disconnect with a split response. This should only be used in PCI-X mode. In PCI mode, this signal has no effect.
- **Target Abort**. Assertion of S_TABORT terminates the current bus transaction with a target abort. This can be used in either bus mode. A target abort is a serious error and signals that data may have been lost or corrupted.

If none of these signals are asserted, the interface transfers data normally. Asserting one of these signals causes something other than *normal data transfer* to occur.

Data Phase Sequencing

The user application must follow certain rules governing the order of control modes. For example, the *normal* transfer mode cannot follow a disconnect with data. In addition, only one mode can be signaled at any one time. For example, it is illegal to assert S_DISCON and S_RETRY on the same clock cycle.

Allowable Sequence Rules

The permitted data phase control sequences for target designs using the core interface are shown in Figure 6-1.

On *the same clock cycle* that S_HIT is asserted, the available options are:

- Transferring data normally
- Inserting wait states
- Issuing a termination

In effect, the core interface begins sampling the control signals during the same cycle that it asserts one bit of the S_HIT bus.

Choices for subsequent clock cycles depend on the action of the user application during the previous clock cycle.

- If you inserted a wait state (S_WAIT asserted), you can continue to insert additional wait states, perform a normal transfer, or issue a termination by asserting one of the termination signals: S_NXTADB, S_DISCON, S_RETRY, S_SPLIT or S_TABORT.
- If you were performing normal data transfer, you can continue in this state or assert any of the termination signals *except* S_RETRY or S_SPLIT or S_DISCON (applies to PCI-X mode only). These signals cannot be asserted because retry, split response, and single dataphase disconnect (applies to PCI-X mode only) may not be issued after data has transferred.
- After asserting one of the termination signals, you must continue to assert the signal until the end of the transaction (S_DONE), with one exception. You may issue a target abort after any other type of termination by deasserting the corresponding control signal while concurrently asserting S_TABORT.
- You are not allowed to go back to a previous state after advancing, as indicated by the arrows in Figure 6-1.

The exact disconnect sequence is affected by whether or not the initiator also terminates the transaction. The interface will automatically generate the correct behavior. Note that the interface will immediately disconnect with data in several cases:

- Implied single data phase transfers in PCI-X and PCI (for example, PCI-X DWORD transactions, PCI configuration and I/O cycles).
- Non-linear or reserved addressing modes in PCI.

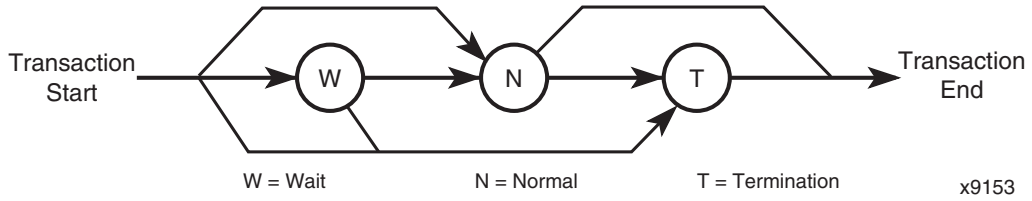


Figure 6-1: **Permitted Data Phase Control Sequences**

Wait State Insertion Rules

Wait states can only be inserted during the first data phase. To insert initial wait states, S_WAIT must be asserted during the assertion of S_HIT, and stay asserted until the user application is ready to transfer data. Once S_WAIT is deasserted, it cannot be asserted for the remainder of the transaction.

Note: Feeding S_HIT through combinational logic to generate S_WAIT or any other control signal is not recommended. It is recommended that S_WAIT and other control signals be driven by a register. After the user application is ready to transfer data, it can reset the S_WAIT register to proceed with normal data transfer, then set it again once the transaction has completed.

The core interface allows you to insert as many wait states as desired. However, the PCI and PCI-X specifications place limits on the allowed number of initial wait states. The designer is responsible for ensuring compliance with these rules in the specification.

Chapter 7

Target Burst Transfers

Performing a single data transfer across the PCI-X bus is the simplest transaction type, but wastes valuable bus bandwidth because of the overhead of distributed address decoding. The performance advantage in PCI-X is derived from burst transactions, where two or more data phases are completed during a transaction.

Building a user application that supports single target transfers is the easiest to design. However, if maximum bandwidth is the goal, building a user application that supports target burst transfers, although more complex, is preferable.

Keeping Track of the Address

In a PCI-X transaction, only the starting address is broadcast over the bus. For single transfers, this is sufficient. For burst transfers, the user application must keep track of the current address.

If the user application performs target burst transfers, it must keep a local copy of the current address and increment it after every successful data transfer. The counter must be able to support bursts throughout the entire address range of the base address register and hence could be as wide as 63 bits. In practice, however, the user application must only keep track of enough bits to represent the largest block of memory space it requests.

As an example, the address counter required for a 16 Mb block of memory space is shown in Table 7-1. A 24-bit loadable binary counter, as shown in the table, will suffice. Note that it is possible to also ignore the two least significant bits of the address. In this case, the result is a 22-bit loadable counter. For 64-bit wide data objects, it may be possible to ignore the three least significant bits of the address.

If an initiator attempts to burst beyond the 16 Mb block boundary, the target can issue a target abort, indicating that it is not able to perform the requested operation.

Table 7-1: **Address Pointer for 16 Mb Memory Space**

23	2	1	0
22-bit loadable counter		0	0

During target writes, the counter should be incremented when S_DATL_VLD is asserted (for a 32 bit wide data object) or when S_DATH_VLD is asserted (for a 64 bit wide data object). During target reads, the address pointer should be incremented when S_DATA_NXT is asserted.

Leveraging the code used in a previous example, the code below implements a loadable 22-bit counter appropriate for use with a 32-bit data object:

```
wire          advance;

assign advance = S_WRITE ? S_DATL_VLD : S_DATA_NXT;


reg     [3:0] scmd;
reg    [35:0] sattr;
reg    [23:0] saddr;

always @(posedge CLK or posedge RST)
begin : slaveinfo
  if (RST)
  begin
    scmd <= 4'b0000;
    sattr <= 36'h000000000;
    saddr[23:0] <= 24'h000000;
  end
  else
  begin
    // Always grab the low address.
    if (S_ADDL_VLD) saddr[23:0] <= S_DATA_OUT[23:0];
    else if (advance) saddr <= saddr + 24'h4;
    // Grab the command.
    if (S_ADDL_VLD) scmd <= S_CBEA_OUT[3:0];
    else if (S_ADDH_VLD) scmd <= S_CBEA_OUT[3:0];
    // Always grab the attribute.
    if (S_ATTR_VLD)
    begin
      sattr <= {S_CBEA_OUT[3:0], S_DATA_OUT[31:0]};
    end
  end
end
```

The same code, with slight modifications, also applies to a loadable 22-bit counter appropriate for use with a 64-bit data object:

```
wire          advance;

assign advance = S_WRITE ? S_DATH_VLD : S_DATA_NXT;


reg     [3:0] scmd;
reg    [35:0] sattr;
reg    [23:0] saddr;

always @(posedge CLK or posedge RST)
begin : slaveinfo
  if (RST)
  begin
    scmd <= 4'b0000;
    sattr <= 36'h000000000;
    saddr[23:0] <= 24'h000000;
  end
  else
  begin
    // Always grab the low address.
    if (S_ADDL_VLD) saddr[23:0] <= S_DATA_OUT[23:0];
    else if (advance) saddr <= saddr + 24'h8;
```

```
            // Grab the command.
            if (S_ADDL_VLD) scmd <= S_CBEA_OUT[3:0];
            else if (S_ADDH_VLD) scmd <= S_CBEA_OUT[3:0];
            // Always grab the attribute.
            if (S_ATTR_VLD)
            begin
              sattr <= {S_CBEA_OUT[3:0], S_DATA_OUT[31:0]};
            end
          end
        end
```

In practice, the width of the counter is determined by the requested memory space—the example above should be modified accordingly.

Sinking Data in Burst Transfers

During target writes, the interface transfers data using a pipelined data path. The data valid signals, S_DATL_VLD or S_DATH_VLD are used to advance the target address pointer and any other data pointers in the user application logic. At the same time the target data pointer is advanced, the user application captures valid data from the internal S_DATA_OUT and S_CBEA_OUT busses.

Using the appropriate data valid signal to capture burst data is very similar to the simple case of single transfers. The user application must enable different registers or memory addresses based on the target address pointer.

Sourcing Data in Burst Transfers

During target reads, the interface transfers data using a pipelined data path. The data source enable signal, S_DATA_NXT, is used to advance the target address pointer and any other data pointers in the user application logic. The user application is responsible for automatically providing the first piece of data during a transfer, as a result of S_HIT assertion; subsequent pieces of data must be provided when the interface asserts S_DATA_NXT.

Using S_DATA_NXT to present data for the next data phase may require additional control logic depending on the type of data source present in the user application. Keep in mind that the S_DATA_NXT signal advances the target address pointer in anticipation of the next data phase, which may or may not complete successfully.

If the target address pointer is advanced, and the data is never transferred, then the user application must decide what to do with the non-transferred data. In the case of prefetchable data sources, such as RAM or a register file, the data can be discarded. The original data remains in the RAM or the register file for future use.

This also applies in cases where a FIFO is used as a rate matching buffer and the contents of the FIFO are flushed after a transaction. Any non-transferred data is discarded from the FIFO, but the original data still remains in the source that originally provided it.

For non-prefetchable data sources, as is the case when a FIFO itself is the data source, pulling data out of the FIFO may be destructive. The unused data must be restored so it is available for future use should it not be transferred. This may require decrementing internal counters or keeping a shadow copy of the previous data.

With a non-prefetchable data source, this condition may arise at the end of a target read burst transfer, particularly when the transaction is terminated by the initiator. In this case,

the user application is not immediately aware of the termination condition, and will have advanced the data source too many times.

One way to determine the number of times the target address pointer has been over-advanced during a burst read is to monitor the difference in the number of cycles S_DATA_NXT and S_DATL_VLD and S_DATH_VLD have been asserted during a transaction. Consider the following cases:

- For a 32-bit data object, assertions of S_DATL_VLD represent successful data transfer on the bus. Assertions of S_DATA_NXT represent speculative reads of the 32-bit data object. A simple counter may be used to keep track of how many more cycles S_DATA_NXT is asserted than S_DATL_VLD. At the end of the transfer, the counter value indicates how many pieces of data must be recovered.
- For a 64-bit data object, assertions of S_DATL_VLD represent successful data transfer on the bus. These successful data transfers may be 32-bit or 64-bit wide, depending on the behavior of the initiator. Assertions of S_DATH_VLD represent one of two events, based on the state of S_DATL_VLD. If S_DATL_VLD is asserted and S_DATH_VLD is asserted, a complete quadword has been transferred. If S_DATL_VLD is deasserted and S_DATH_VLD is asserted, the transfer has ended and half a quadword has been transferred. This can occur because a 32-bit initiator may finish the transfer on a non-quadword address. As in the 32-bit case, assertions of S_DATA_NXT represent speculative reads of the 64-bit data object. A simple counter may be used to keep track of how many times S_DATA_NXT is asserted compared to S_DATL_VLD and S_DATH_VLD assertions. At the end of the transfer, the counter value indicates how many pieces of data must be recovered.

In practice, it is cumbersome to deal with non-prefetchable 64-bit data objects due to the alignment problems that may result when accessed by 32-bit initiators on the bus. For this reason, the use of prefetchable data objects is recommended. If non-prefetchable data objects are required, the design can be significantly simplified by declaring them as 32-bit data objects.

Design Example

The following example demonstrates the use of a prefetchable 64-bit data object. Prefetchable data sources, such as RAM and general purpose register files, do not exhibit side effects from reads (that is, the state of the memory is not altered by the act of reading). Figure 7-1 shows such a data source with an address counter as they might be implemented in a user application.

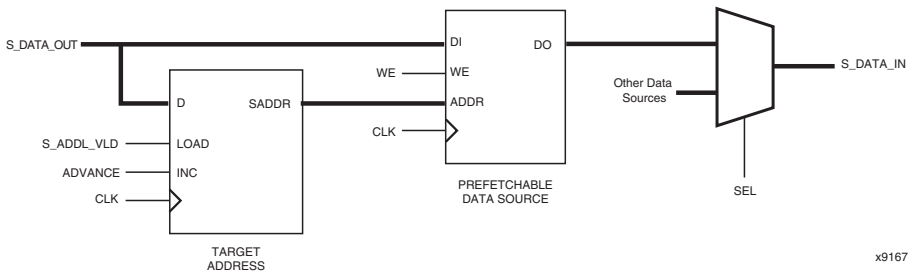


Figure 7-1: **Prefetchable Data Source**

For this example, assume that the prefetchable data source is a 256 byte asynchronous memory, organized as 32 by 64 and implemented with distributed RAM. The target address space, as specified by the associated base address register, is 4096 bytes, and

declared as a 64-bit data object. The *PCI Local Bus Specification* suggests that memory address spaces be no smaller than 4096 bytes, so in this example, the majority of the available address range is unused. The following code implements the target address pointer:

```
wire          advance;

assign advance = S_WRITE ? S_DATH_VLD : S_DATA_NXT;


reg     [3:0] scmd;
reg    [35:0] sattr;
reg     [7:0] saddr;

always @(posedge CLK or posedge RST)
begin : slaveinfo
  if (RST)
  begin
    scmd <= 4'b0000;
    sattr <= 36'h000000000;
    saddr <= 8'h00;
  end
  else
  begin
    // Always grab the low address.
    if (S_ADDL_VLD) saddr <= S_DATA_OUT[7:0];
    else if (advance) saddr <= saddr + 8'h08;
    // Grab the command.
    if (S_ADDL_VLD) scmd <= S_CBEA_OUT[3:0];
    else if (S_ADDH_VLD) scmd <= S_CBEA_OUT[3:0];
    // Always grab the attribute.
    if (S_ATTR_VLD)
    begin
      sattr <= {S_CBEA_OUT[3:0], S_DATA_OUT[31:0]};
    end
  end
end
```

The target address pointer drives the address input of the memory array. After implementing the target transaction decode logic that generates BAR_x_RD and BAR_x_WR, the next step is to implement the logic for the memory write enable and the memory data output select.

```
wire          we;
wire          sel;
wire   [63:0] ram_dout;


assign we = S_DATH_VLD & BAR_x_WR;
assign sel = BAR_x_RD;
assign S_DATA_IN = sel ? ram_dout : other_data;
```

Once this logic is in place, all that remains is to create the memory array and to assign values to the target transaction control signals.

```
// Instantiate eight 32x8 distributed RAMs

RAM32X8S RAM0 (.D(S_DATA_OUT[7:0]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
```

```
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[7:0]));

RAM32X8S RAM1 (.D(S_DATA_OUT[15:8]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[15:8]));

RAM32X8S RAM2 (.D(S_DATA_OUT[23:16]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[23:16]));

RAM32X8S RAM3 (.D(S_DATA_OUT[31:24]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[31:24]));

RAM32X8S RAM4 (.D(S_DATA_OUT[39:32]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[39:32]));

RAM32X8S RAM5 (.D(S_DATA_OUT[47:40]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[47:40]));

RAM32X8S RAM6 (.D(S_DATA_OUT[55:48]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[55:48]));

RAM32X8S RAM7 (.D(S_DATA_OUT[63:56]), .WE(we),
.WCLK(CLK), .A0(saddr[3]), .A1(saddr[4]),
.A2(saddr[5]), .A3(saddr[6]), .A4(saddr[7]),
.O(ram_dout[63:56]));
```

As implemented, this design is always ready to perform burst transfers. If desired, the control signals can be used to delay or split transactions to allow time for the user application to fill the data buffer.

```
assign S_TABORT = 1'b0;
assign S_RETRY = 1'b0;
assign S_DISCON = 1'b0;
assign S_NXTADB = 1'b0;
assign S_WAIT = 1'b0;
assign S_SPLIT = 1'b0;
```

A similar design for a 32-bit target requires three changes:

- Use of `S_DATL_VLD` instead of `S_DATH_VLD`
- Adjustment of the address counter increment to 8'h04
- Reorganization of the 32 by 64 memory array as 64 by 32

The waveforms shown in Figures 7-2 through 7-7 illustrate the behavior of this example design when it is the target of a BURST write transaction. Simple DWORD transactions are similar, and are shown in Chapter 5, "Basic Target Transaction Control."

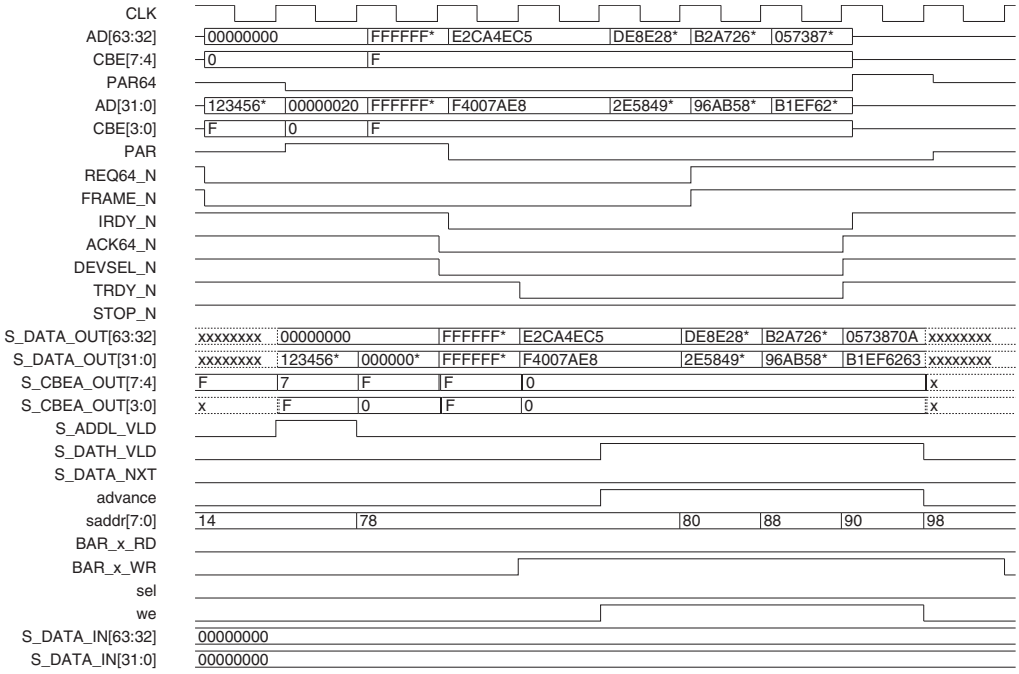


Figure 7-2: **PCI-X Burst Write, 64-Bit**

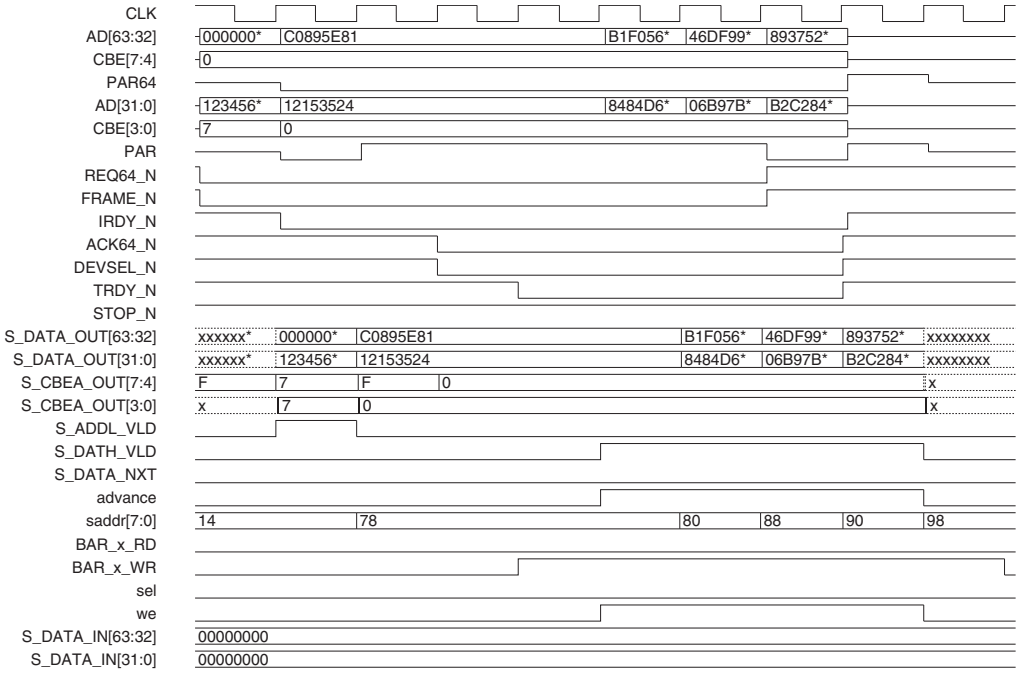


Figure 7-3: **PCI Burst Write, 64-Bit**

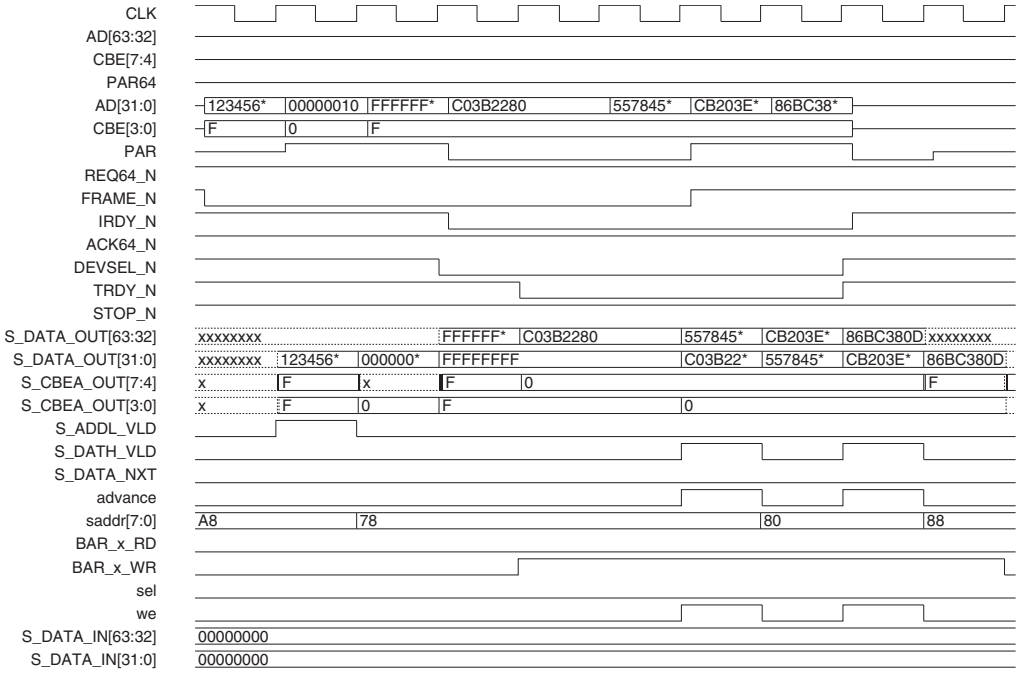


Figure 7-4: **PCI-X Burst Write, 32-Bit Aligned**

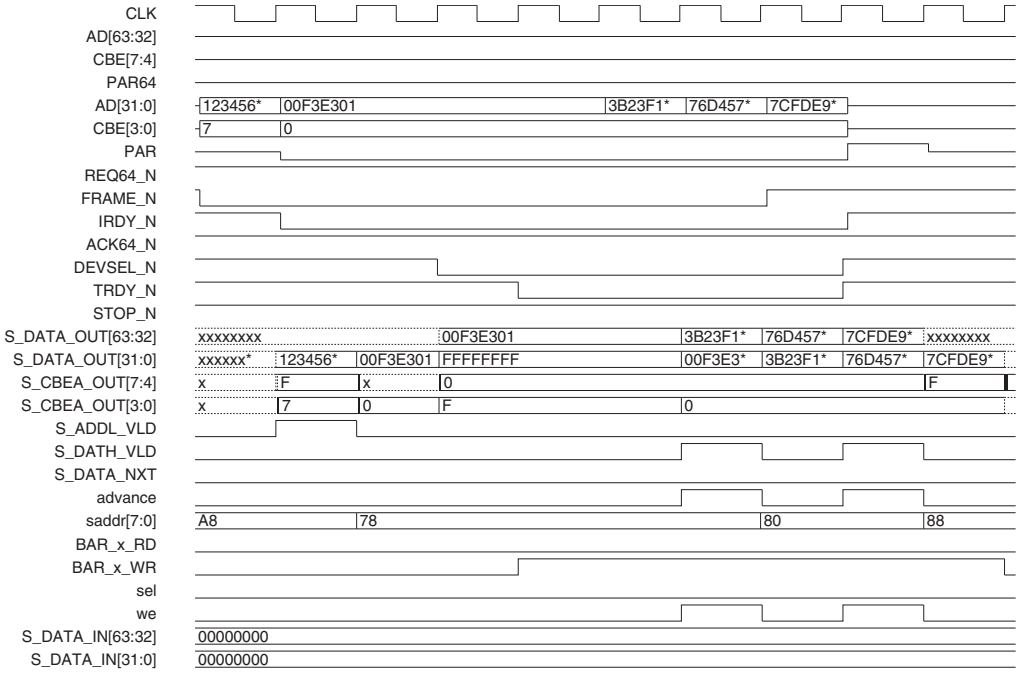


Figure 7-5: **PCI Burst Write, 32-Bit Aligned**

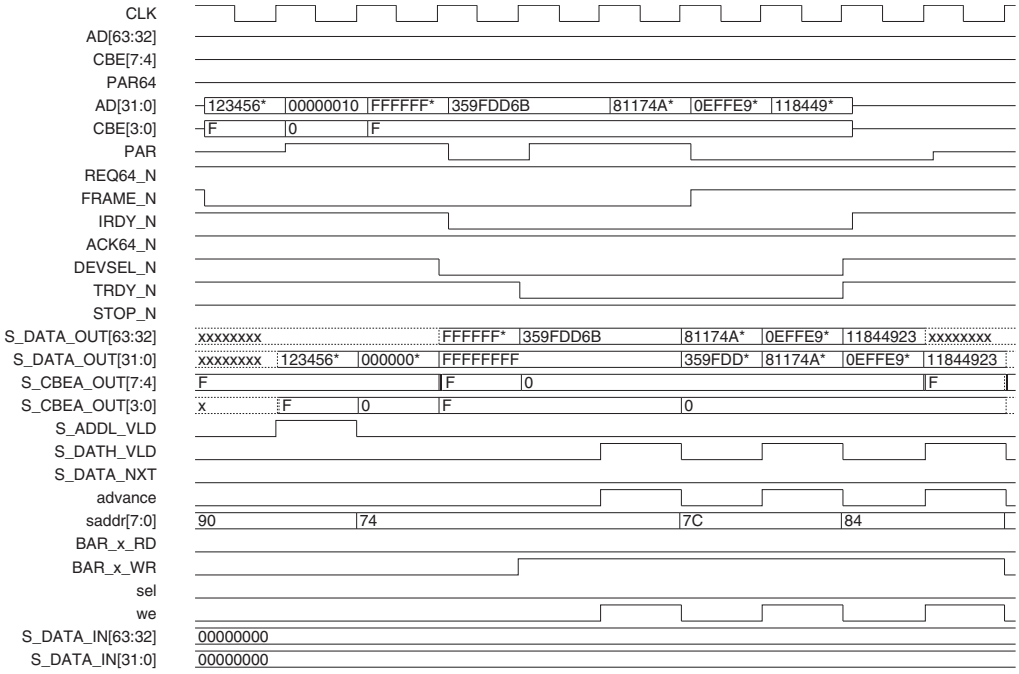


Figure 7-6: **PCI-X Burst Write, 32-Bit Unaligned**

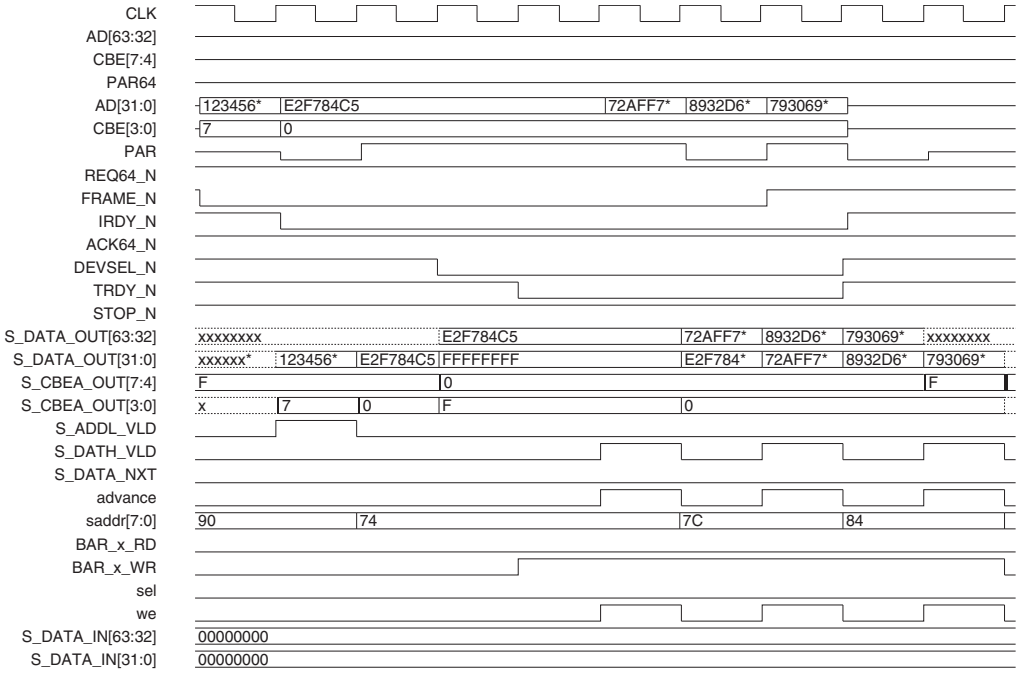


Figure 7-7: **PCI Burst Write, 32-Bit Unaligned**

The waveforms shown in Figures 7-8 through 7-13 illustrate the behavior of this example design when it is the target of a BURST read transaction. Simple DWORD transactions are similar, and are shown in Chapter 5, "Basic Target Transaction Control."

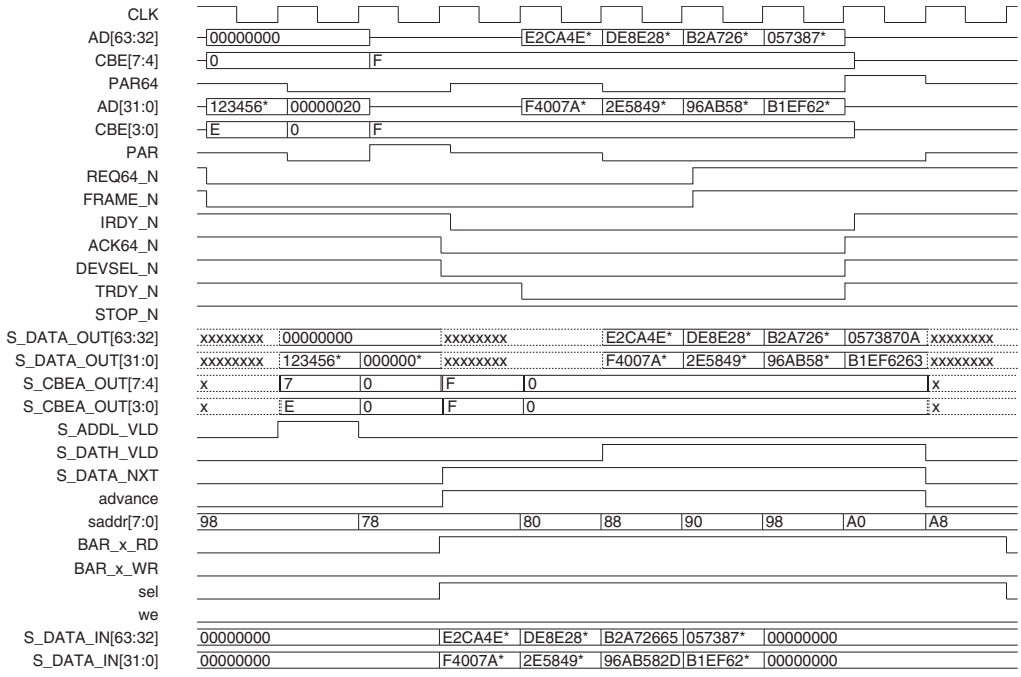


Figure 7-8: **PCI-X Burst Read, 64-Bit**

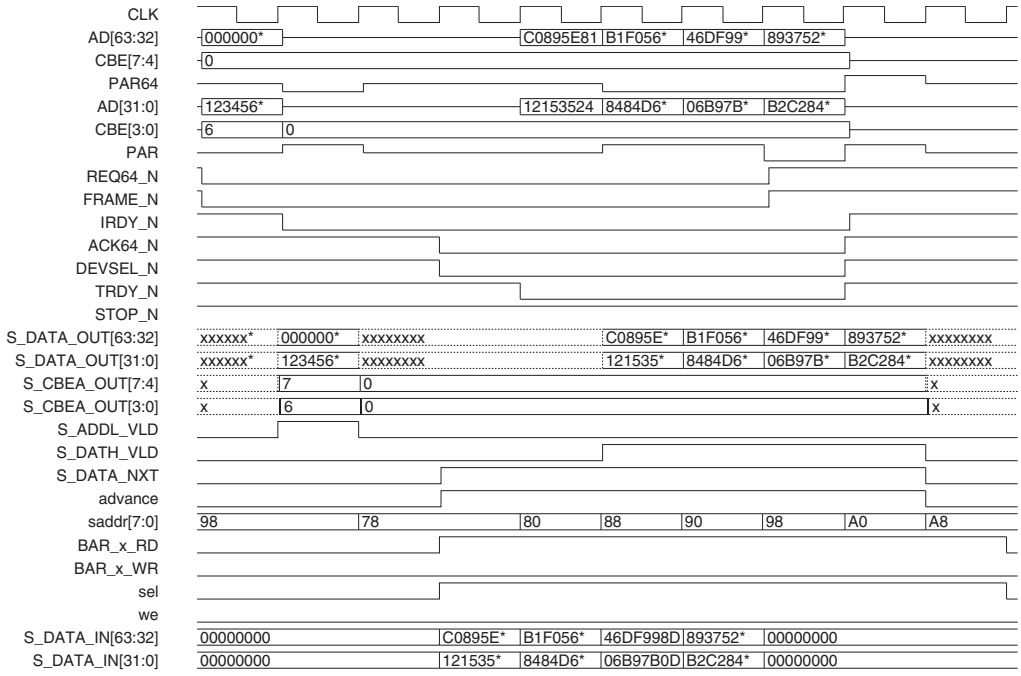


Figure 7-9: **PCI Burst Read, 64-Bit**

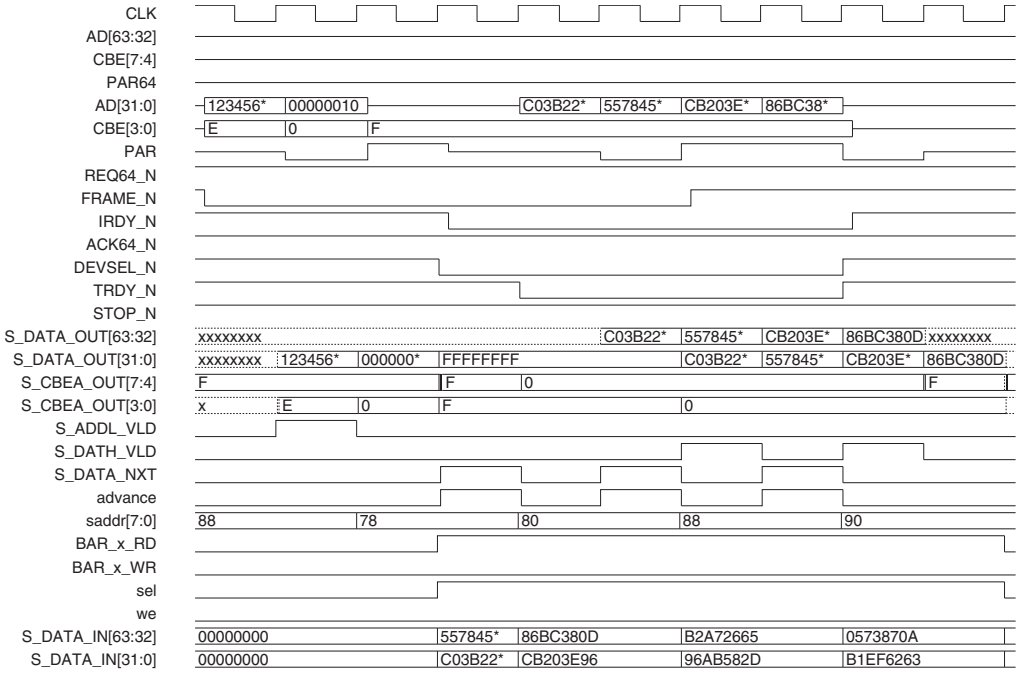


Figure 7-10: **PCI-X Burst Read, 32-Bit Aligned**

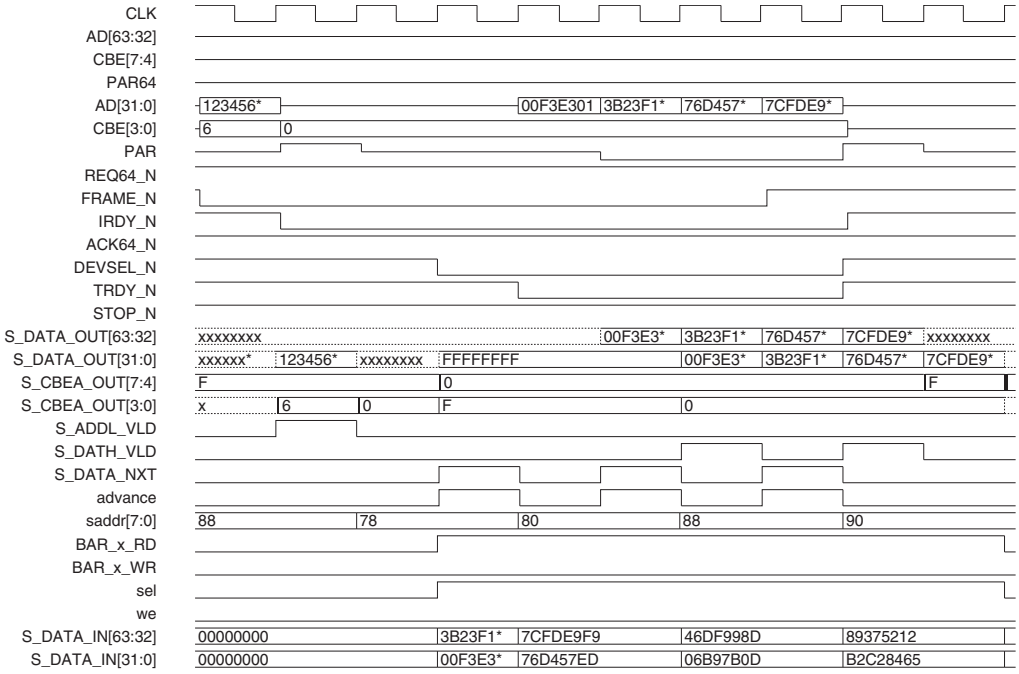


Figure 7-11: **PCI Burst Read, 32-Bit Aligned**

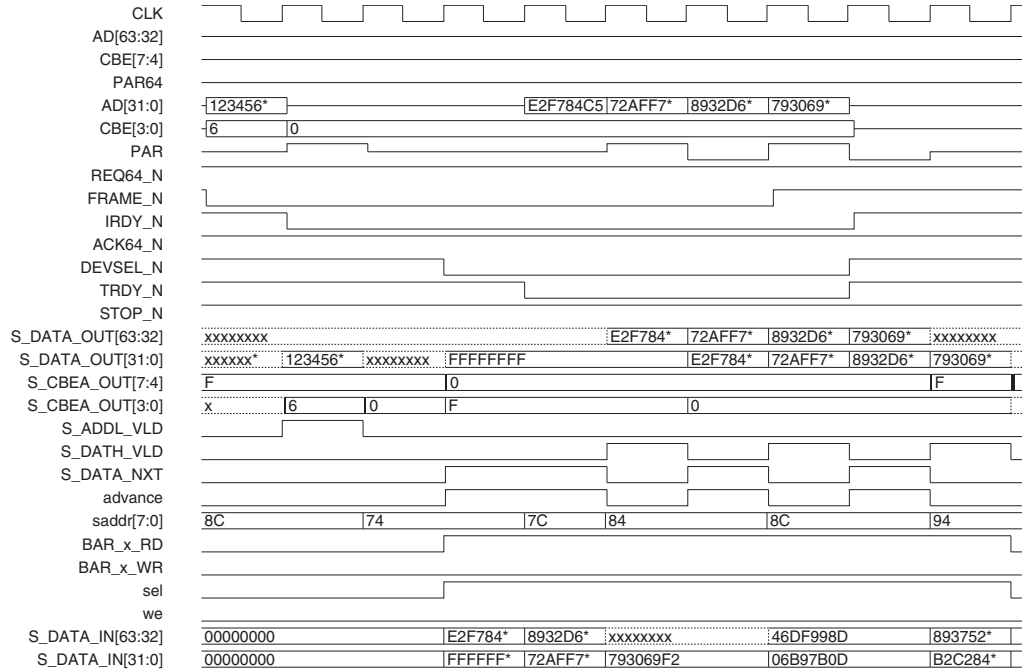


Figure 7-12: **PCI-X Burst Read, 32-Bit Unaligned**

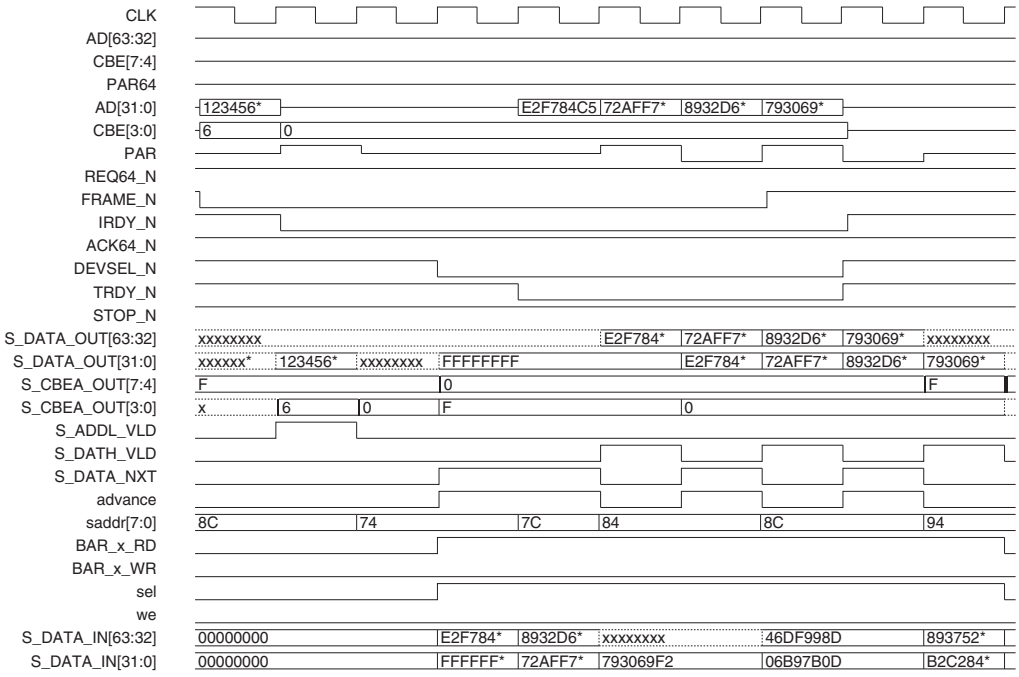


Figure 7-13: **PCI Burst Read, 32-Bit Unaligned**

Chapter 8

Basic Initiator Transaction Control

This chapter discusses the logic required in the user application to generate the load and output select signals for a typical initiator register.

In applications not using initiator burst transactions, data is usually transferred to and from registers in the user application. These registers are connected to control signals required for initiator data transfer and to any additional control and data path logic provided by the user. They may also connect to internal FIFOs or to I/O pins on the user application. Figure 8-1 illustrates the interface of a typical initiator register.

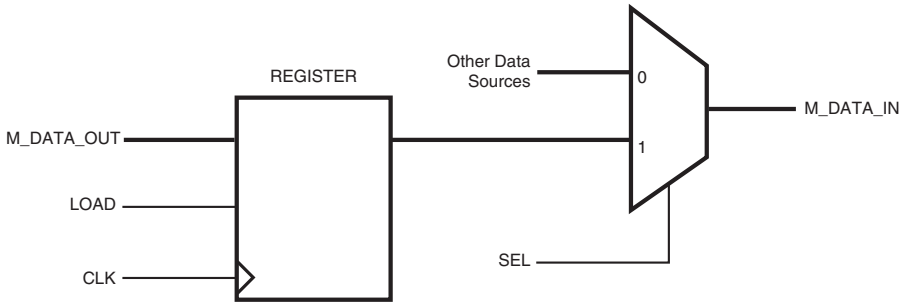


Figure 8-1: **Example Initiator Register**

Important: The core interface *does not support* initiating transactions to its own target, including operations such as reading and writing its own configuration space.

Initiator Transaction Overview

The process of servicing a transaction where the user application is the initiator includes the following steps:

1. Requesting a transaction.
2. Providing transaction attributes and parameters.
3. Sending or receiving data on the first data phase.
4. Sending or receiving data on subsequent data phases.

This chapter focuses on the first three steps and describes how to build an initiator user application that handles single-data phase transactions. The following chapters explain how this mechanism is extended to handle burst transactions.

Requesting a Transaction

When the user application wishes to initiate a transfer on the bus, it must make a request to the core interface by asserting M_REQ for a single cycle.

- M_REQ: Input that requests that the interface begin a bus transaction. A request must only be made when the user application is ready to transfer data and allowed to do so per the state of the busmaster enable bit in the command register.

In PCI-X mode, the user application is allowed to initiate split completion transactions at any time, regardless of the state of the busmaster enable bit in the command register. For other bus commands in PCI-X mode, and for all commands in PCI mode, the user application must defer assertion of M_REQ until the busmaster enable bit is set.

Once the user application has made a request for bus access, the interface will interrogate the user application for additional information about the desired transfer.

Providing Transaction Information

Before a transaction begins on the PCI-X bus, the user application must provide information about it to the core interface with the following signals:

- **M_DATA_IN[63:0]**: Input bus that provides the means for address, attribute, and data transfer from the user application to the core interface.
- **M_CBEA_IN[7:0]**: Input bus that provides the means for command, byte enable, and attribute transfer from the user application to the core interface.
- **M_ADDL_VLD**: Output that indicates that the user application must drive a valid low address on M_DATA_IN[31:0]. This address will either be a 32-bit address or the lower 32 bits of a 64-bit address. Note that in PCI-X mode, some types of split completion transactions require the user to supply an address of zero.
- **M_ADDH_VLD**: Output that indicates that the user application must drive a valid high address on M_DATA_IN. If the initiator has been configured to use the full 64-bit data path, M_ADDH_VLD will assert with M_ADDL_VLD; in this case, drive the full 64-bit address on M_DATA_IN[63:0]. If the initiator has been configured to use only a 32-bit data path, M_ADDH_VLD will follow M_ADDL_VLD in the next cycle; in this case, drive the high 32 bits of the 64-bit address on M_DATA_IN[31:0].
- **M_ATTR_VLD**: Output that indicates that the user application must drive transaction attributes on M_DATA_IN[31:0] and M_CBEA_IN[3:0]. These attributes are not a one to one match with the attribute phase in a PCI-X transaction. The attributes control the behavior of the interface and must be used even if the interface is in PCI mode.

The user application first passes the transaction information to the interface via the attributes. Figure 8-2 details how data is passed to the interface.

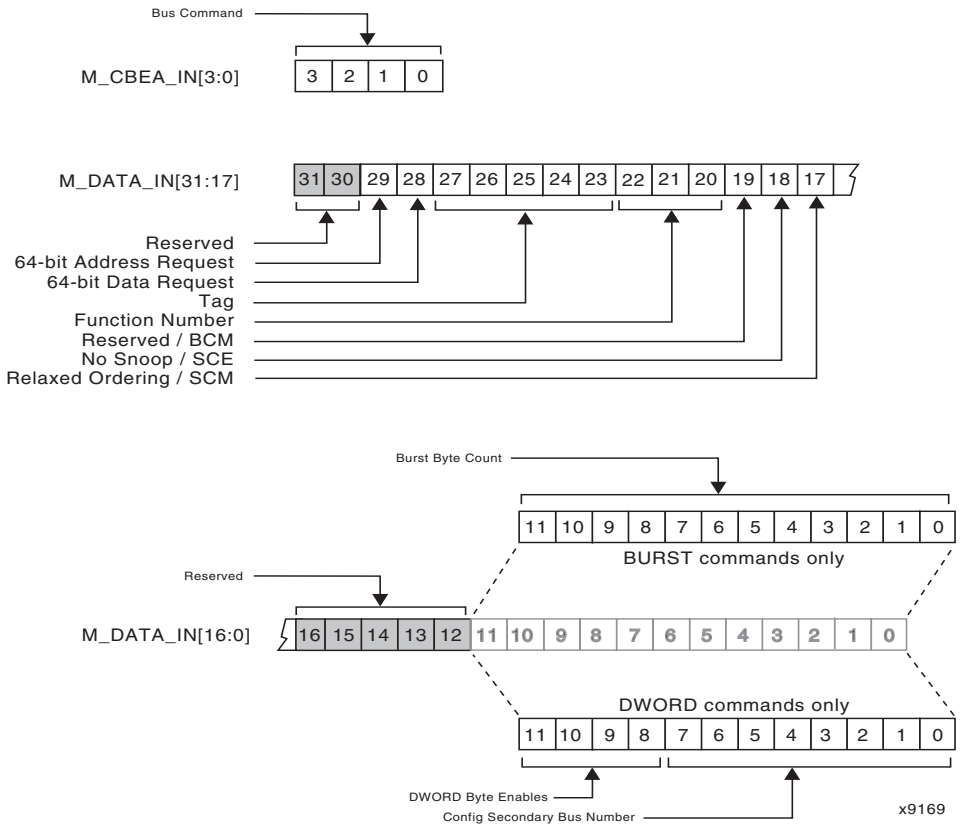


Figure 8-2: **Attribute and Command Format**

The bus command is presented on `M_CBEA_IN[3:0]`. Based on the bus command, one of two attribute formats are used. One format is used for explicit DWORD commands, and the other format is used for BURST commands. Figure 8-2 shows both formats. Important aspects of the attributes include:

- 64-bit addressing is requested by setting bit 29. If used with a memory command, setting this bit will result in a dual address cycle on the bus.
- 64-bit wide data transfer is requested by setting bit 28. If used with a memory command or a split completion, and the requested transfer is at least three QWORDS, setting this bit will result in a 64-bit data transfer request on the bus. Otherwise, the transfer will take place as a 32-bit data transfer.
- The tag field should be changed with each transaction so that split transactions, should they occur, will be uniquely identified.
- The function number field is for support of multifunction devices. The interface does not currently support multiple functions. Set this field to zero.
- The byte count modified, split completion error, and split completion message bits set the corresponding bits in split completion attributes.
- The information required to be driven on `M_DATA_IN[11:0]` depends on the command.
  - For BURST commands, `M_DATA_IN[11:0]` represents a 12-bit byte count.

- For DWORD commands, `M_DATA_IN[11:8]` represents byte enables and `M_DATA_IN[7:0]` is typically set to zero, unless the command is a configuration command. In this case, `M_DATA_IN[7:0]` should be set to the bus number.

The initiator requires the use of attributes in both PCI or PCI-X mode. These attributes do not translate directly to the attributes expressed during PCI-X transactions.

Since the byte enables for DWORD commands are presented with the attributes, `M_CBEA_IN[7:0]` is ignored during DWORD commands. Byte enables are generated automatically for all BURST commands except memory write. Note that memory read is defined to be a DWORD command in PCI-X but is a BURST command in PCI. For consistency, this interface will only perform memory reads as DWORD commands in either mode.

After requesting the transaction attributes, the core interface will prompt the user application for address information. Depending on the setting of attribute bit 29 (64-bit address request) and the configuration of the interface, some combination of `M_ADDL_VLD` and `M_ADDH_VLD` will be asserted to request address information.

In PCI bus mode, addresses used with a 64-bit wide data transfer request must be quadword aligned. In PCI-X bus mode, this restriction does not apply. However, to simplify the user application design, it is useful to enforce this requirement in both bus modes.

Figure 8-3 shows the user application requesting a transaction with a 32-bit address. Note that this behavior is independent of the initiator width configuration.

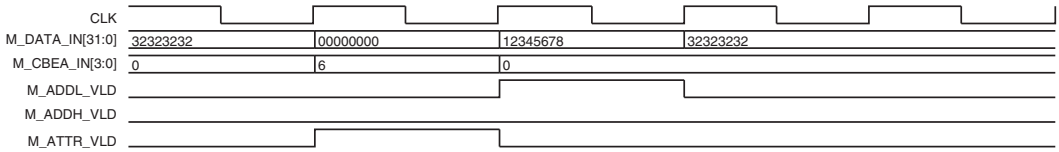


Figure 8-3: **Providing a 32-Bit Address as an Initiator**

Figure 8-4 shows the user application requesting a transaction with a 64-bit address. This behavior will occur if the initiator has been declared as 32-bit.

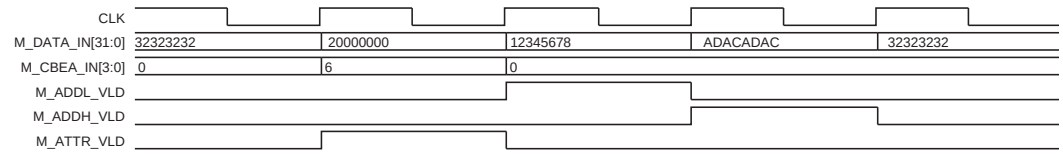


Figure 8-4: **Providing a 64-Bit Address as a 32-Bit Initiator**

Figure 8-5 shows the user application requesting a transaction with a 64-bit address. This behavior will occur if the initiator has been declared as 64-bit.

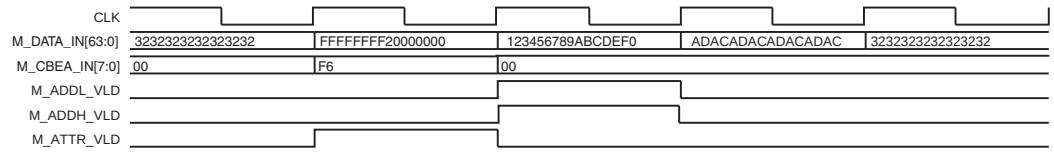


Figure 8-5: **Providing a 64-Bit Address as a 64-Bit Initiator**

Sending or Receiving Data

The initiator portion of the user application can be either a 32-bit wide data object or a 64-bit wide data object. While the core interface provides 64-bit wide connections to the user application, the initiator implemented in the user application may communicate with the interface using either a 32-bit or 64-bit data transfer. The behavior is set when the core is created based on the bus width of the interface.

When you design for a declared 32-bit initiator, all data transfer for the initiator between the interface and the user application is conducted over the low half of the data path, independent of what type of transfer takes place on the PCI-X bus. Similarly, when you design for a 64-bit wide initiator, all data transfer for the initiator between the interface and the user application is conducted over the full width of the data path, independent of the type of transfer on the PCI-X bus.

Important: If the initiator interface is configured for 64-bit operation, the interface expects the user application to present quadword aligned data, even when using DWORD commands. However, for certain split completion transactions, such as split completion messages and DWORD completions, the user application must provide an address of zero. In these cases, the initiator interface always behaves as if the data is QWORD aligned. This may require the user application to swap the single outbound DWORD to properly align it.

Initiator Read

During an initiator read operation, the user application captures data from the `M_DATA_OUT` bus. If the initiator user application performs only single data phase transactions, the data is typically captured in a register.

Once an initiator read transaction is established (that is, through the process described in the previous section), valid data is available on `M_DATA_OUT` whenever one of the following two signals is asserted by the core interface:

- **`M_DATL_VLD`**: Output that indicates the user application should capture valid 32-bit data from the `M_DATA_OUT[31:0]` bus, and is used only if the initiator interface has been configured as a 32-bit data object.
- **`M_DATH_VLD`**: Output that indicates the user application should capture valid 64-bit data from the **`M_DATA_OUT[63:0]`** bus, and is used only if the initiator interface has been configured as a 64-bit data object.

The following is an example of a 32-bit register. The `write` signal is generated by the user application. The load control signal on the initiator register would be generated as follows:

```
assign LOAD = M_DATL_VLD & !write; //32-bit object
```

The actual register implementation can be described as shown:

```
reg    [31:0] my_reg;

always @(posedge CLK or posedge RST)
begin : write_my_reg
  if (RST) my_reg <= 32'h00000000;
  else if (LOAD)
  begin
    my_reg[31:0] <= M_DATA_OUT[31:0];
  end
end
```

The waveforms in Figure 8-6 and Figure 8-7 show the time relationship. These waveforms include bus signals and internal user application signals.

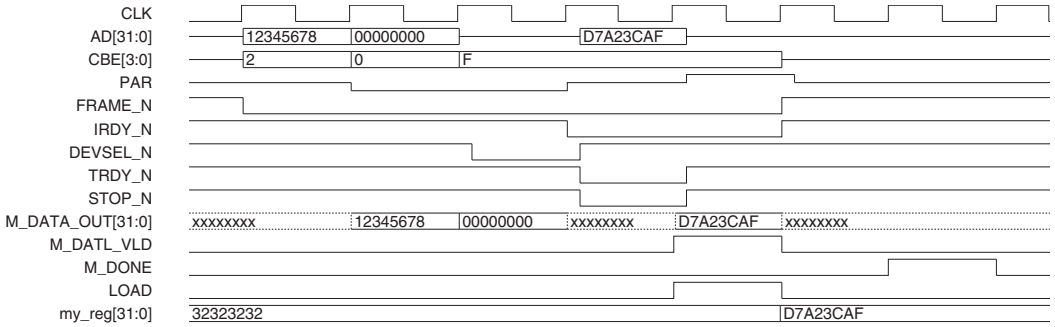


Figure 8-6: **PCI-X Initiator Read Transaction to 32-bit Register**

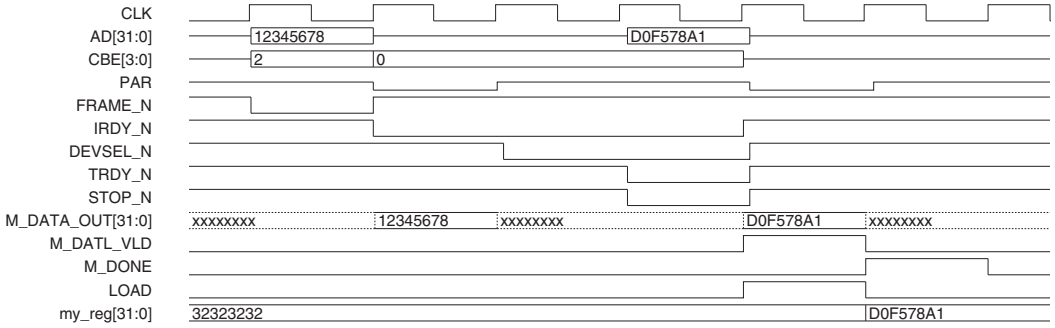


Figure 8-7: **PCI Initiator Read Transaction to 32-bit Register**

The following is an example of a 64-bit register. The load control signal on the target register would be generated as follows:

```
assign LOAD = M_DATH_VLD & !write; //64-bit object
```

The actual register implementation can be described as shown:

```
reg    [63:0] my_reg;

always @(posedge CLK or posedge RST)
begin : write_my_reg
  if (RST) my_reg <= 64'h0000000000000000;
  else if (LOAD)
  begin
    my_reg[63:0] <= M_DATA_OUT[63:0];
  end
end
```

The waveforms in Figure 8-8 and Figure 8-9 show the time relationship. These waveforms include bus signals and internal user application signals.

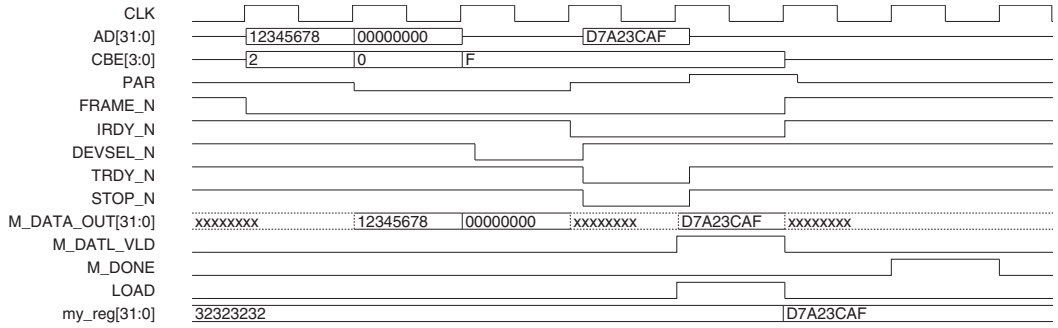


Figure 8-8: **PCI-X Initiator Read Transaction to 64-bit Register**

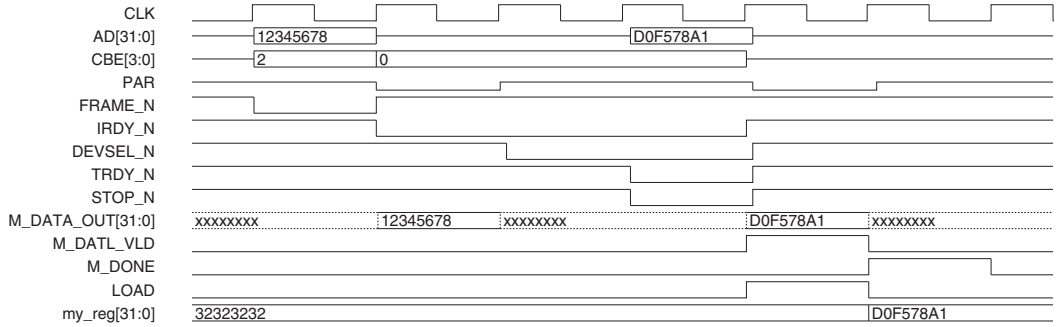


Figure 8-9: **PCI Initiator Read Transaction to 64-bit Register**

Initiator Write

During an initiator write operation, the user application presents data to the core interface via the `M_DATA_IN` bus. If the command requires byte enables, the user application presents the byte enables on `M_CBEA_IN`. If the initiator user application performs only single data phase transactions, the transaction typically involves a single register or a bank of registers, 32 or 64 bits wide.

Once an initiator write transaction is established (through the process described in the previous section), valid data must be presented immediately on `M_DATA_IN`. If the user application has more than one register, it must determine which register is being accessed and direct its output onto `M_DATA_IN` through a multiplexer:

```
assign SEL = fn (DATA_SOURCE);
```

If the initiator is a 32-bit data object, valid data need only appear on `M_DATA_IN[31:0]`. If the initiator is a 64-bit data object, valid data must be presented on the entire width of the `M_DATA_IN[63:0]` bus. Always drive the required width of the `M_DATA_IN` bus with valid data, even if the registers are non-prefetchable.

The time relationship is shown in the following waveforms (Figure 8-10 through Figure 8-13), which show both bus signals and user application signals.

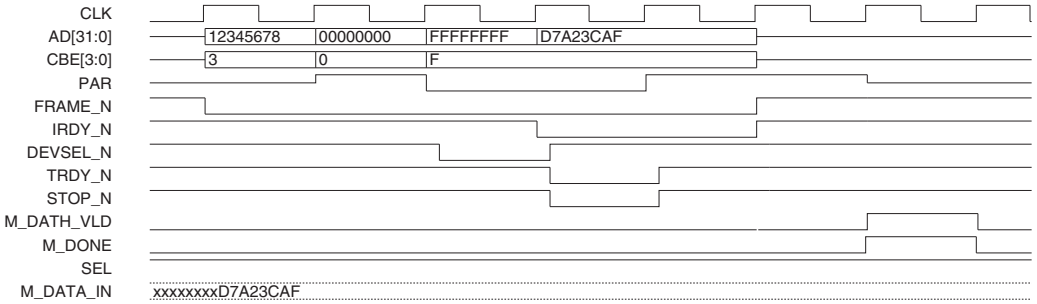


Figure 8-10: **PCI-X Initiator Write Transaction of 32-bit Register**

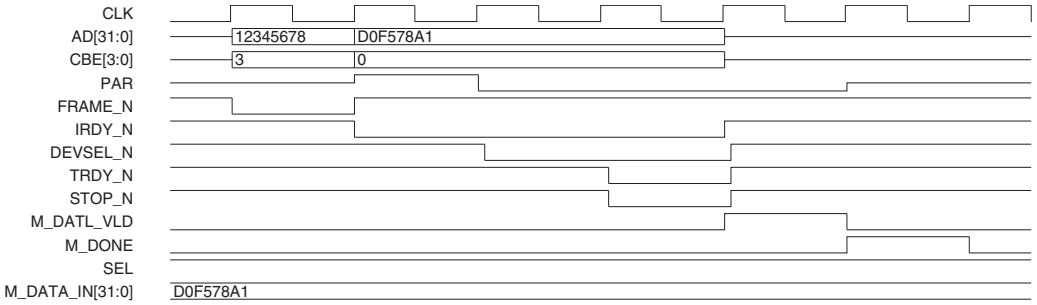


Figure 8-11: **PCI Initiator Write Transaction of 32-bit Register**

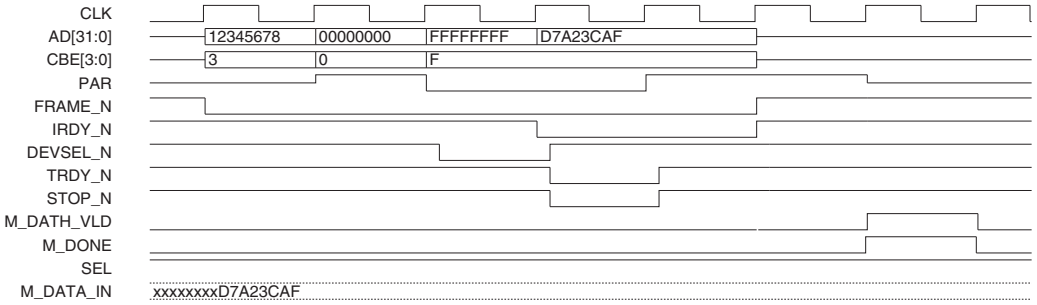


Figure 8-12: **PCI-X Initiator Write Transaction of 64-bit Register**

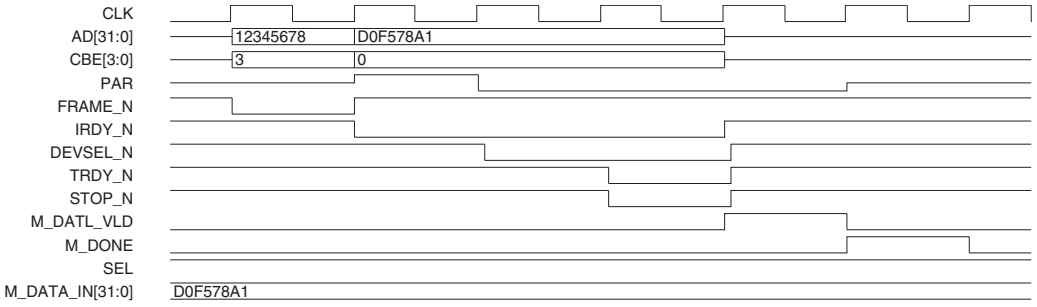


Figure 8-13: **PCI Initiator Write Transaction of 64-bit Register**

Terminating Initiator Transactions

In general, PCI-X transactions may be terminated by the initiator or the target. This section discusses simple non-burst transactions; disconnects are discussed in Chapter 6, "Advanced Target Transaction Control."

The PCI-X protocol defines some bus commands as inherently single data phase transfers. These commands are listed below:

- I/O Read
- I/O Write
- Configuration Read
- Configuration Write
- Memory Read DWORD

In PCI, the Memory Read command is a BURST command, while in PCI-X, the comparable Memory Read DWORD is not. The core interface, while operating in PCI mode, enforces consistent behavior in all bus modes by treating Memory Read as a DWORD command. The remaining bus commands permit burst transfers.

The core interface automatically completes DWORD transfers without intervention from the user application, as the interface knows that these commands are explicitly single data phase. When using BURST commands, the interface will automatically terminate the transfer when the byte count is satisfied.

Chapter 9

Advanced Initiator Transaction Control

This chapter discusses how the user application can control the initiator transactions to accommodate its own ability to source or sink data in the following ways:

- Initiator-inserted wait states are not allowed.
- Premature initiator termination allows the user application to limit the number of data phases in a transaction, which is a useful control in burst designs.

This chapter also discusses the terminations that can be received by the initiator interface during a transaction.

Control Modes

Initiator-inserted wait states are not allowed in either bus mode. Do not request an initiator transaction unless the data to be transferred is available (for a write) or sufficient empty buffers are available (for a read).

The user application should be capable of completing the entire byte count requested when the attributes were presented to the interface. Assuming the addressed target is also capable of completing the entire transfer, the core interface will automatically complete the transaction once the byte count has expired.

In the following cases, however, it may be necessary to prematurely terminate the transaction:

- Catastrophic failure in the user application
- Unforeseen buffer overflow
- Unforeseen buffer underflow

The user application can end the transaction in a premature manner by asserting `M_FINISH`. A well designed user application should not need to assert `M_FINISH` if it refrains from requesting transactions that exceed its buffering capabilities. However, if this control signal is used, the following rules must be observed:

- `M_FINISH` must not be asserted until an indication that data transfer has started. Specifically, do not assert `M_FINISH` until either `M_DATL_VLD` or `M_DATH_VLD` has been asserted by the interface.
- Absolutely never assert `M_FINISH` at the beginning of a transfer request to immediately disconnect at the next ADB. Instead, adjust the byte count of the transfer.
- `M_FINISH` must remain asserted until `M_DONE` is asserted.

In PCI-X mode, the transaction will complete at the next ADB. If `M_FINISH` is not asserted several cycles prior to an ADB, the interface will finish on the subsequent ADB.

In PCI mode, the interface will end the transaction as soon as possible, taking into account the cache line size, if required.

Transfer Status Signals

As discussed above, an initiator transaction will complete normally when the byte count has been transferred, assuming the addressed target is capable of completing the burst. If required, the user application can prematurely terminate the transfer using `M_FINISH`.

Additionally, there are several other cases which will result in a premature transfer termination, independent of actions taken by the user application:

- Target termination.
- Latency timer expiration.

The user application can determine when the transaction completes or terminates by monitoring `M_DONE`. The interface may assert one of several status signals during a transfer to provide additional information. All status signals are guaranteed to be valid at the time `M_DONE` is asserted:

- `M_MABORT` indicates that no target responded to the transaction. This is the normal completion for special cycles and can also occur during bus enumeration. At any other time, this is a fatal error.
- `M_RETRY` indicates that the target disconnected without data. In PCI-X mode, this can only be a retry. In PCI mode, this signal only indicates that a disconnect without data has taken place. If the user application needs to know a retry has taken place in PCI mode, it must also monitor the data valid signals to determine if any data transfer has taken place.
- `M_NXTADB` indicates that the target disconnected on an ADB. This signal is never asserted in PCI mode.
- `M_DISCON` indicates that the target disconnected with data. In PCI-X mode, this can only be a single data phase disconnect on the first data phase. In PCI mode, this signal only indicates that a disconnect with data has taken place.
- `M_TABORT` indicates that the target issued a target abort, which is a fatal error.
- `M_SPLIT` indicates that the target split the transaction. This signal is never asserted in PCI mode. When a transaction is terminated with a split request, no data transfer occurs. However, the interface may assert one of the data valid signals. User applications that handle split requests and generate split completions must discard any *data* transferred if `M_SPLIT` is asserted. An alternate approach is to monitor `M_SPLIT` a cycle earlier and then qualify the data valid signals with `M_SPLIT`.

If `M_DONE` is asserted, and none of the status signals are asserted, this means that one of three events took place:

- The requested number of bytes were transferred.
- The latency timer expired and the transfer ended.
- The user application asserted `M_FINISH` and the transfer ended.

No matter how the transaction ends, the user application is responsible for acting correctly. The core interface does not automatically re-request the bus after a transaction terminates prematurely. The user application may need to create a new request if the complete byte count did not transfer.

The transaction status indicators are reset once a new transaction is requested from the user application.

Chapter 10

Initiator Burst Transfers

As is the case in target transactions, single transfers as an initiator waste valuable bus bandwidth. The performance advantage in PCI-X is derived from burst transactions, where two or more data words are transferred during the transaction.

Building a user application that supports single initiator transfers is moderately complex, while building a user application that supports initiator burst transfers is even more complex, but worth the effort, where maximum bandwidth is the goal.

Keeping Track of the Address

In a PCI-X transaction, only the starting address is broadcast over the bus. For single transfers as an initiator, a register that holds the target address is sufficient. For burst transfers, however, the user application must keep track of the current address because the target can terminate the transaction at any time. If the initiator is to continue the transfer during another transaction, it must resume at the appropriate address.

The initiator must always provide and track a full address (32-bit or 64-bit, depending on the application). In some applications, however, a smaller address counter and a larger data register are sufficient to track the current address. The actual size of the counter depends on the alignment and length of the transfers required by the user application. In the worst case, the initiator may require a full 64-bit, loadable binary counter. See Table 10-1

Table 10-1: **Example Initiator Address Pointer**

63	2	1	0
62-bit loadable counter		0	0

During initiator reads, the counter should be incremented when `M_DATL_VLD` is asserted (for a 32 bit wide data object) or when `M_DATH_VLD` is asserted (for a 64 bit wide data object). During initiator writes, the address pointer should be incremented when `M_DATA_NXT` is asserted.

The examples in this chapter assume the existence of four signals provided by the user application. The generation of these signals is user application dependent:

```
wire          my_start;  // Transaction start.
wire          my_write;  // Data direction.
wire   [63:0] my_addr;   // Transfer address.
wire   [35:0] my_attr;   // Transfer attributes.
```

The code below implements a loadable counter appropriate for use with a 32-bit data object:

```
wire          advance;
wire          load;
reg    [63:0] maddr;

assign advance = my_write ? M_DATA_NXT : M_DATL_VLD;
assign load = my_start;

always @(posedge CLK or posedge RST)
begin : masterinfo
  if (RST) maddr <= 64'h00000000_00000000;
  else
  begin
    // Copy the address sent to interface.
    if (load) maddr <= my_addr;
    else if (advance) maddr <= maddr + 8'h04;
  end
end
```

The same code, with slight modifications, also applies to a loadable counter appropriate for use with a 64-bit data object:

```
wire          advance;
wire          load;
reg    [63:0] maddr;

assign advance = my_write ? M_DATA_NXT : M_DATH_VLD;
assign load = my_start;

always @(posedge CLK or posedge RST)
begin : masterinfo
  if (RST) maddr <= 64'h00000000_00000000;
  else
  begin
    // Copy the address sent to interface.
    if (load) maddr <= my_addr;
    else if (advance) maddr <= maddr + 8'h08;
  end
end
```

If the initiator does not use 64-bit addresses, and never issues a 64-bit address request (which results in a dual address cycle on the bus), then the width of the address counter need not exceed 32 bits.

Sinking Data in Burst Transfers

During initiator reads, the interface transfers data using a pipelined data path. The data valid signals, `M_DATL_VLD` or `M_DATH_VLD` are used to advance the initiator address pointer and any other data pointers in the user application logic. At the same time the initiator data pointer is advanced, the user application also captures valid data from the internal `M_DATA_OUT` bus.

Using the appropriate data valid signal to capture burst data is very similar to the simple case of single transfers. The user application must enable different registers or memory addresses based on the initiator address pointer.

Sourcing Data in Burst Transfers

During initiator writes, the interface also transfers data using a pipelined data path. The data source enable signal, M_DATA_NXT, is used to advance the initiator address pointer and any other data pointers in the user application logic. The user application is responsible for automatically providing the first piece of data during a transfer, as a result of M_ATTR_VLD assertion; subsequent pieces of data must be provided when the interface asserts M_DATA_NXT.

The byte enables for DWORD commands are presented with the attributes, thus M_CBEA_IN is ignored during data transfer with DWORD commands. Byte enables are generated automatically for all BURST commands except memory write. Note that memory read is defined to be a DWORD command in PCI-X but is a BURST command in PCI. For consistency, this interface will only perform memory reads as DWORD commands in either mode. When using the memory write command, M_CBEA_IN, as well as M_DATA_IN, must be driven with valid information.

Using M_DATA_NXT to present data for the next data phase may require additional control logic depending on the type of data source present in the user application. Keep in mind that the M_DATA_NXT signal advances the initiator address pointer in anticipation of the next data phase, which may or may not complete successfully.

If the initiator address pointer is advanced, and the data is never transferred, then the user application must decide what to do with the non-transferred data. In the case of prefetchable data sources, such as RAM or a register file, the data can be discarded. The original data remains in the RAM or the register file for future use.

This also applies in cases where a FIFO is used as a rate matching buffer and the contents of the FIFO are flushed after a transaction. Any non-transferred data is discarded from the FIFO, but the original data still remains in the source that originally provided it.

For non-prefetchable data sources, as is the case when a FIFO itself is the data source, pulling data out of the FIFO may be destructive. The unused data must be restored so it is available for future use should it not be transferred. This may require decrementing internal counters or keeping a shadow copy of the previous data.

With a non-prefetchable data source, this condition may arise at the end of an initiator write burst transfer, particularly when the transaction is terminated by the target. In this case, the user application is not immediately aware of the termination condition, and will have advanced the data source too many times.

One way to determine the number of times the initiator address pointer has been over-advanced during a burst write is to monitor the difference in the number of cycles M_DATA_NXT and M_DATL_VLD and M_DATH_VLD have been asserted during a transaction. Consider the following two cases:

- For a 32-bit data object, assertions of M_DATL_VLD represent successful data transfer on the bus. Assertions of M_DATA_NXT represent speculative reads of the 32-bit data object. A simple counter may be used to keep track of how many more cycles M_DATA_NXT is asserted than M_DATL_VLD. At the end of the transfer, the counter value indicates how many pieces of data must be recovered.
- For a 64-bit data object, assertions of M_DATL_VLD represent successful data transfer on the bus. These successful data transfers may be 32-bit or 64-bit wide, depending on the behavior of the target. Assertions of M_DATH_VLD represent one of two events, based on the state of M_DATL_VLD. If M_DATL_VLD is asserted and M_DATH_VLD is asserted, a complete quadword has been transferred. If M_DATL_VLD is deasserted and M_DATH_VLD is asserted, the transfer has ended and half a quadword has been

transferred. This can occur because a 32-bit target may finish the transfer on a non-quadword address. As in the 32-bit case, assertions of M_DATA_NXT represent speculative reads of the 64-bit data object. A simple counter may be used to keep track of how many times M_DATA_NXT is asserted compared to M_DATL_VLD and M_DATH_VLD assertions. At the end of the transfer, the counter value indicates how many pieces of data must be recovered.

In practice, it is cumbersome to deal with non-prefetchable 64-bit data objects due to the alignment problems that may result when accessing 32-bit targets on the bus. For this reason, the use of prefetchable data objects is recommended. If non-prefetchable data objects are required, the design can be significantly simplified by declaring them as 32-bit data objects.

Design Example

The following design example demonstrates the use of a prefetchable 64-bit data object. Prefetchable data sources, such as RAM and general purpose register files, do not exhibit "side effects" from reads (that is, the state of the memory is not altered by the act of reading). Figure 10-1 shows such a data source with an address counter as they might be implemented in a user application.

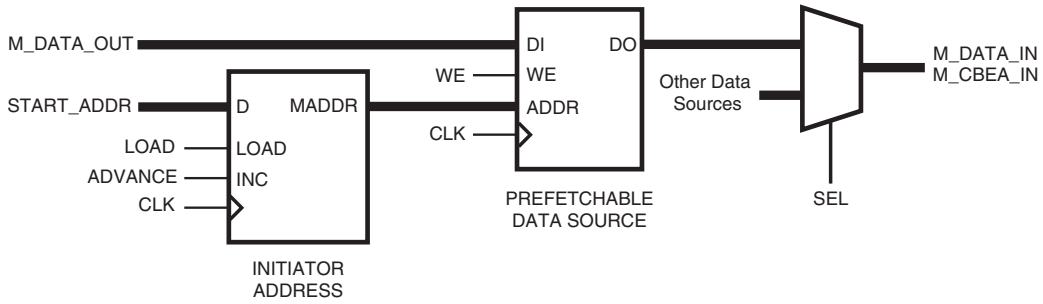


Figure 10-1: **Prefetchable Data Source**

For this example, assume that the prefetchable data source is a 256 byte asynchronous memory, organized as 32 by 64 and implemented with distributed RAM. The initiator is declared as a 64-bit data object.

Again, this example assumes the existence of four signals provided by the user application. The generation of these signals is user application dependent:

```
wire          my_start;  // Transaction start.
wire          my_write;  // Data direction.
wire   [63:0] my_addr;   // Transfer address.
wire   [35:0] my_attr;   // Transfer attributes.
```

The following code implements the initiator address pointer for a 64-bit data object:

```
wire          advance;
wire          load;
reg    [63:0] maddr;

assign advance = my_write ? M_DATA_NXT : M_DATH_VLD;
assign load = my_start;
```

```
always @(posedge CLK or posedge RST)
begin : masterinfo
  if (RST) maddr <= 64'h00000000_00000000;
  else
  begin
    // Copy the address sent to interface.
    if (load) maddr <= my_addr;
    else if (advance) maddr <= maddr + 8'h08;
  end
end
```

The format for the transfer attributes, `my_attr`, is discussed in Chapter 8, "Basic Initiator Transaction Control." A summary of the data passed to the interface is shown again in Figure 10-2.

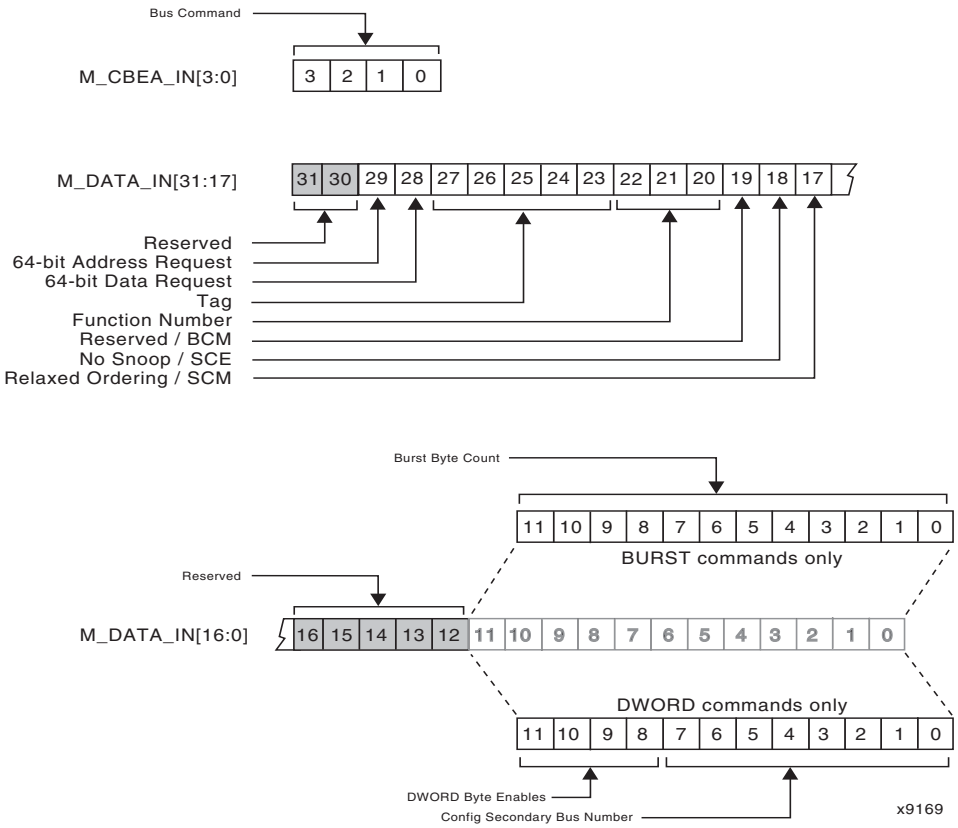


Figure 10-2: **Attribute and Command Format**

The initiator address pointer drives the address input of the memory array. The next step is to implement the logic for the memory write enable and the memory data output selects.

```
wire          we;
wire          sel_attr;
wire          sel_addr;
wire   [63:0] ram_dout;

assign we = M_DATH_VLD & !my_write;
assign sel_attr = M_ATTR_VLD;
```

```
assign sel_addr = M_ADDL_VLD;

assign M_CBEA_IN = sel_attr ?
                   {4'h0, my_attr[35:32]} : 8'h00;
assign M_DATA_IN = sel_attr ?
                   {32'h00000000,my_attr[31:0]} :
                   (sel_addr ? my_addr : ram_dout);
```

Once this logic is in place, all that remains is to create the memory array and to assign values to the initiator transaction control signals.

```
// Instantiate eight 32x8 distributed RAMs

RAM32X8S RAM0 (.D(M_DATA_OUT[7:0]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[7:0]));

RAM32X8S RAM1 (.D(M_DATA_OUT[15:8]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[15:8]));

RAM32X8S RAM2 (.D(M_DATA_OUT[23:16]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[23:16]));

RAM32X8S RAM3 (.D(M_DATA_OUT[31:24]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[31:24]));

RAM32X8S RAM4 (.D(M_DATA_OUT[39:32]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[39:32]));

RAM32X8S RAM5 (.D(M_DATA_OUT[47:40]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[47:40]));

RAM32X8S RAM6 (.D(M_DATA_OUT[55:48]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[55:48]));

RAM32X8S RAM7 (.D(M_DATA_OUT[63:56]), .WE(we),
.WCLK(CLK), .A0(maddr[3]), .A1(maddr[4]),
.A2(maddr[5]), .A3(maddr[6]), .A4(maddr[7]),
.O(ram_dout[63:56]));
```

This design does not terminate transfers prematurely. If desired the control signal can be used to stop transactions before their natural completion.

```
assign M_FINISH = 1'b0;
```

A similar design for a 32-bit initiator would require four changes:

- Use of `M_DATL_VLD` instead of `M_DATH_VLD`.
- Adjustment of the address counter increment to 8'h04.
- Reorganization of the 32 by 64 memory array as 64 by 32.

- Modification of the data select logic for `M_DATA_IN` to support two cycle address transfer.

The waveforms in Figures 10-3 through 10-8 illustrate the behavior of this example design when it is the initiator of a BURST *read* transaction. Simple DWORD transactions are similar, as defined in Chapter 8, "Basic Initiator Transaction Control."

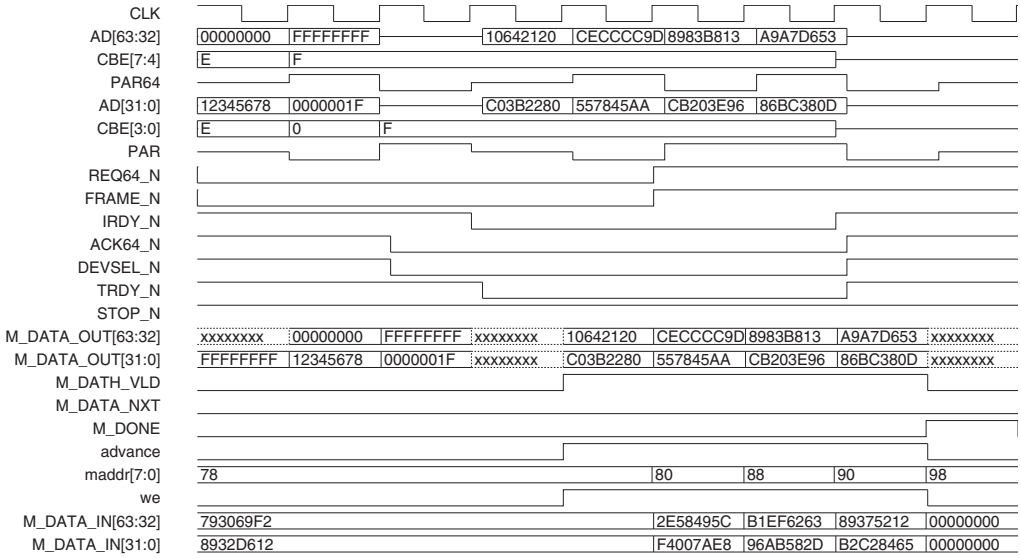


Figure 10-3: **PCI-X Burst Read, 64-Bit Request, 64-Bit Target**

Figure 10-4: **PCI Burst Read, 64-Bit Request, 64-Bit Target**

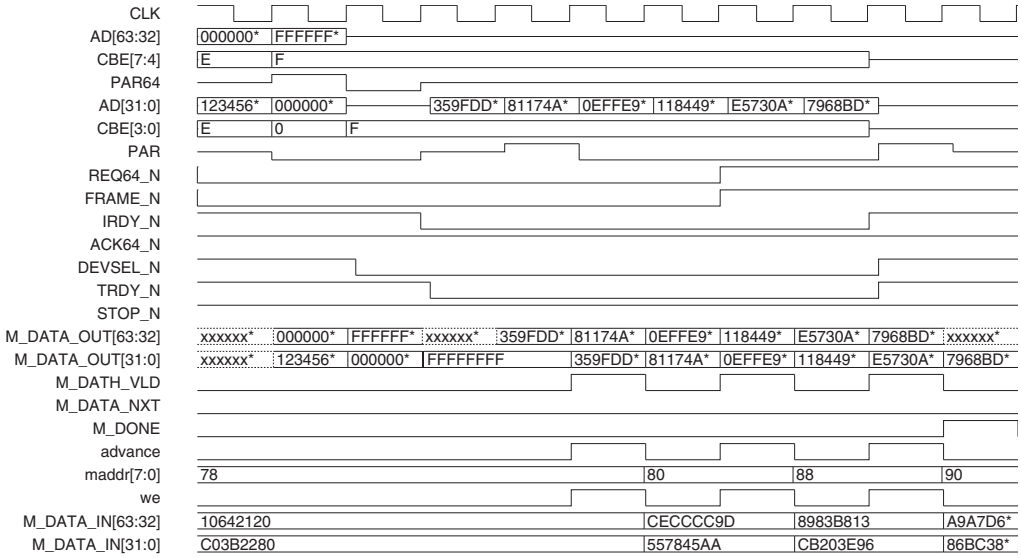


Figure 10-5: **PCI-X Burst Read, 64-Bit Request, 32-Bit Target**

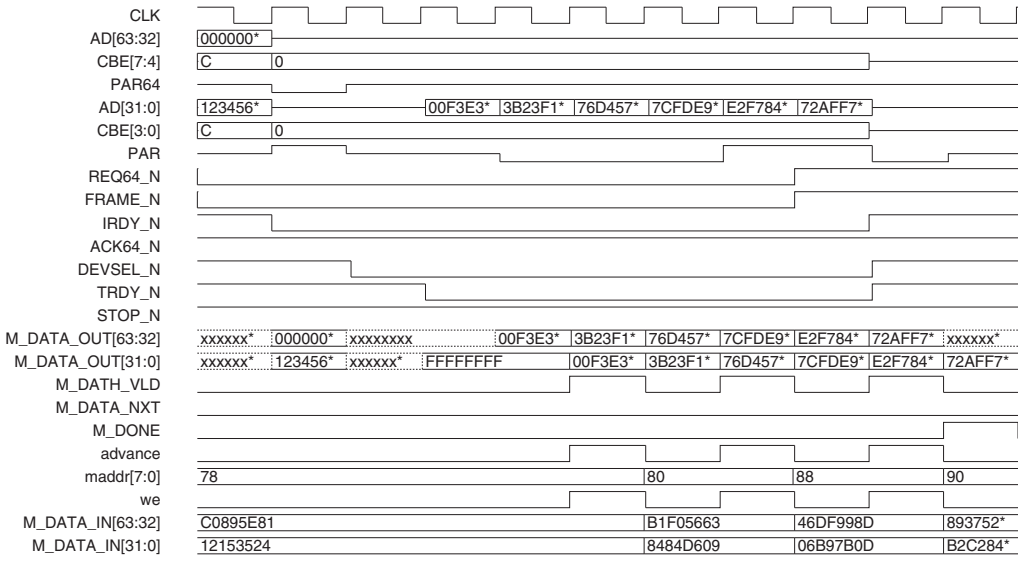


Figure 10-6: **PCI Burst Read, 64-Bit Request, 32-Bit Target**

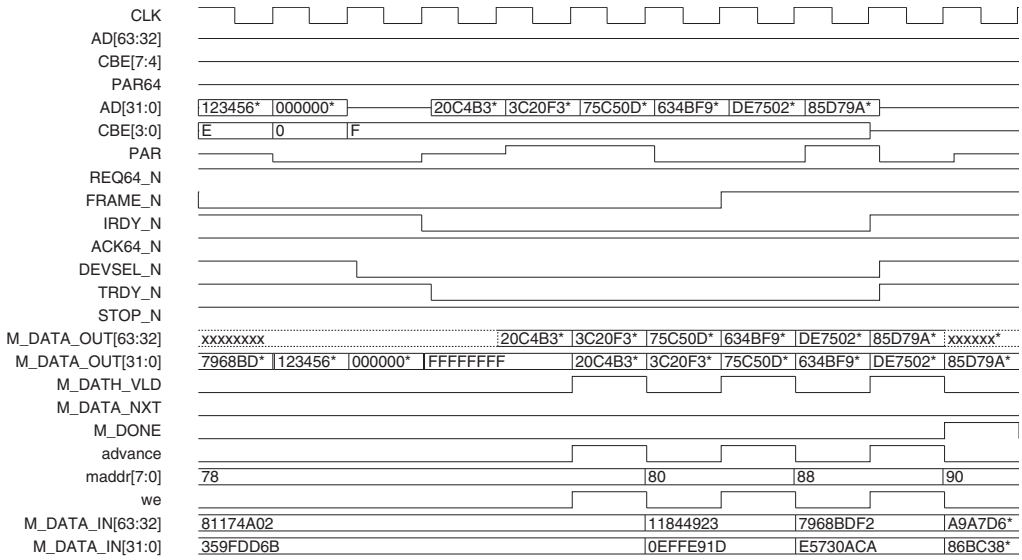


Figure 10-7: **PCI-X Burst Read, 32-Bit Request, 32-Bit Target**

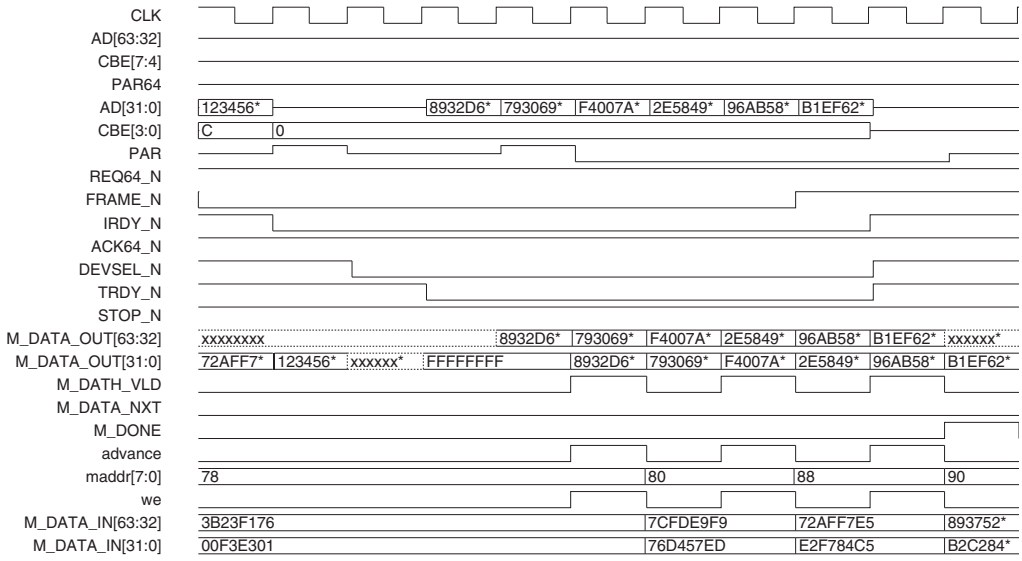


Figure 10-8: **PCI Burst Read, 32-Bit Request, 32-Bit Target**

The waveforms shown in Figures 10-9 through 10-14 illustrate the behavior of this example design when it is the initiator of a BURST *write* transaction. Simple DWORD transactions are similar, as defined in Chapter 8, "Basic Initiator Transaction Control."

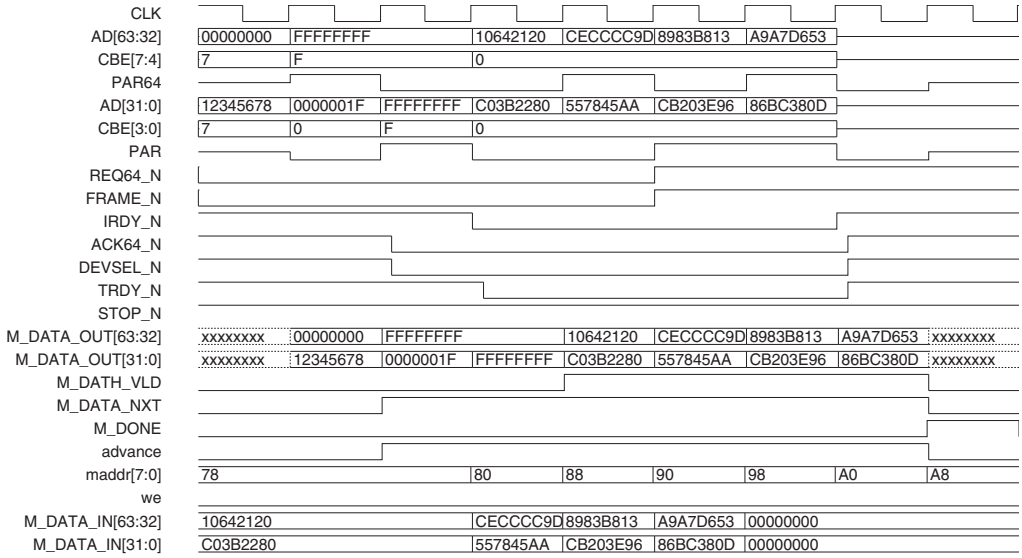


Figure 10-9: **PCI-X Burst Write, 64-Bit Request, 64-Bit Target**

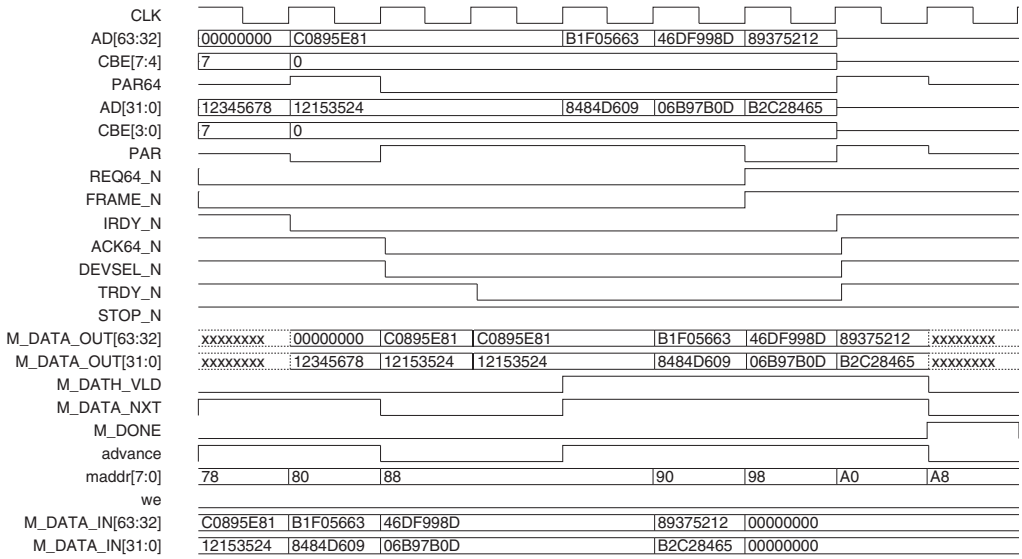


Figure 10-10: **PCI Burst Write, 64-Bit Request, 64-Bit Target**

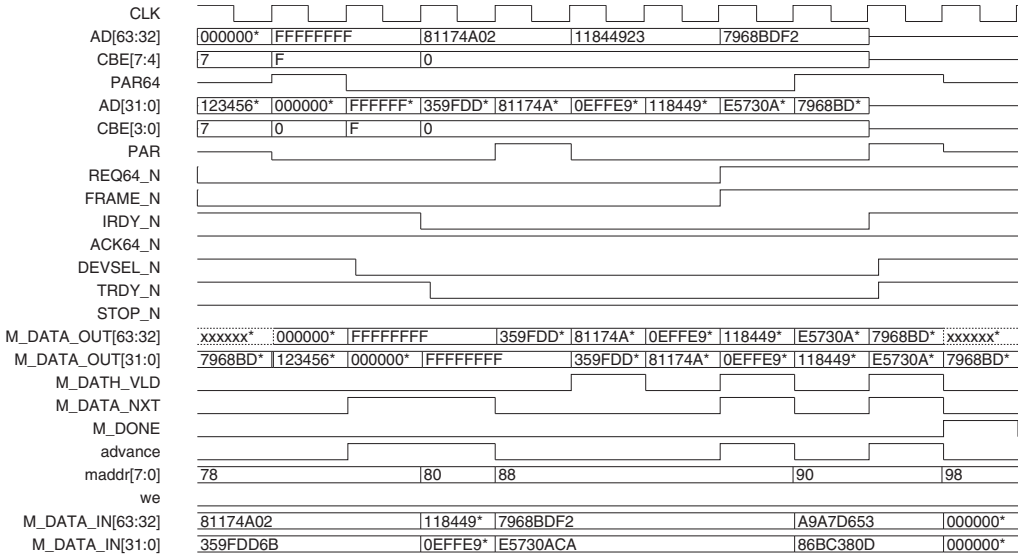


Figure 10-11: **PCI-X Burst Write, 64-Bit Request, 32-Bit Target**

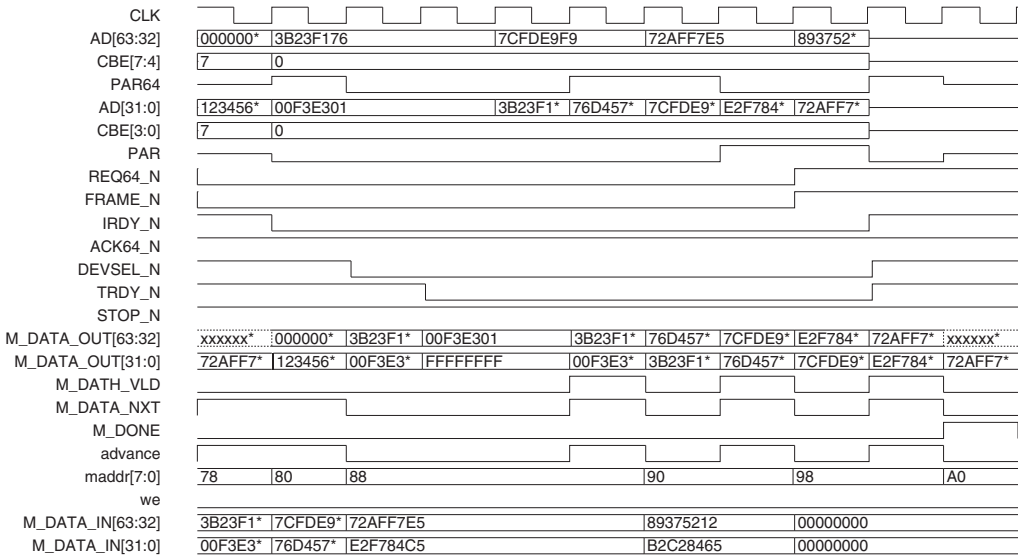


Figure 10-12: **PCI Burst Write, 64-Bit Request, 32-Bit Target**

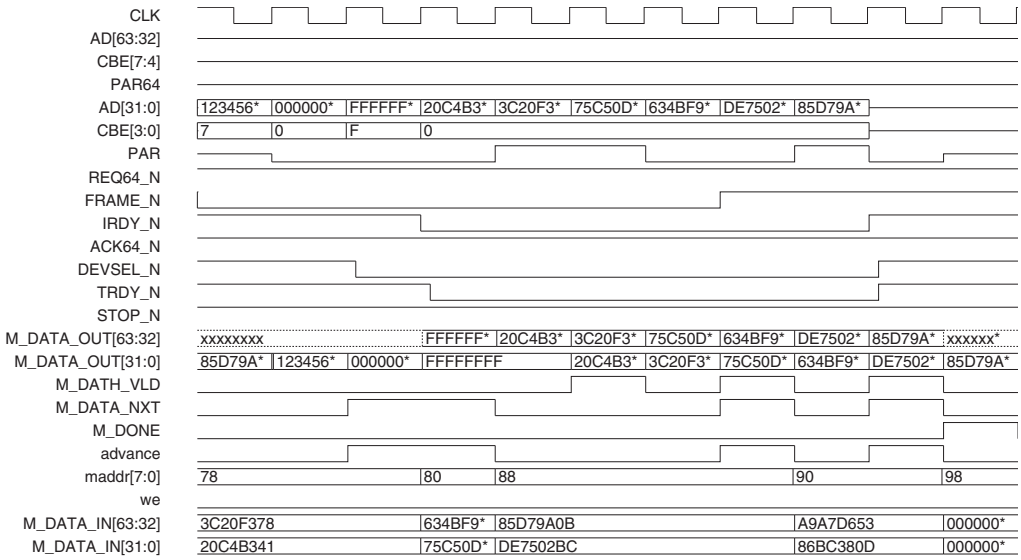


Figure 10-13: **PCI-X Burst Write, 32-Bit Request, 32-Bit Target**

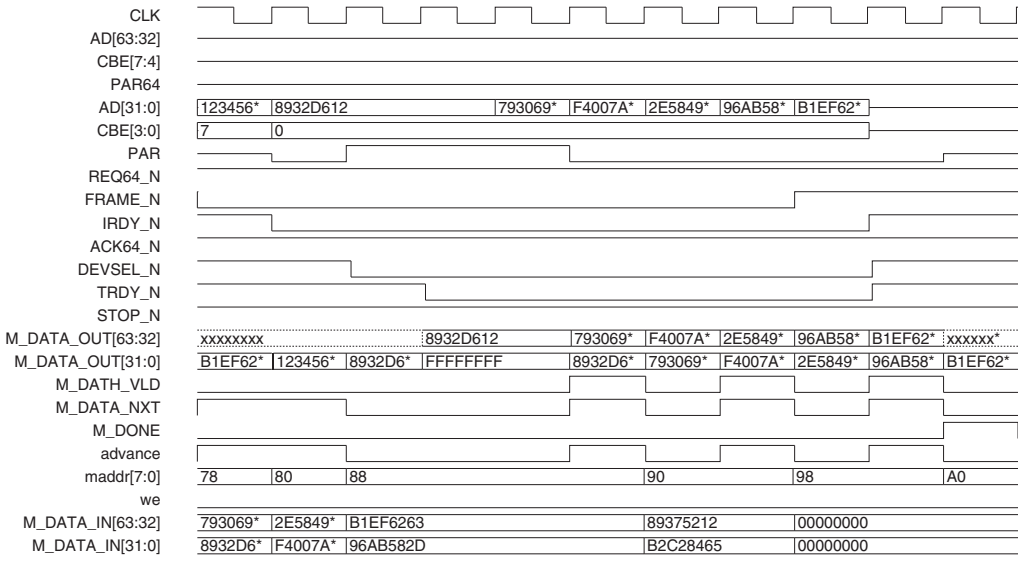


Figure 10-14: **PCI Burst Write, 32-Bit Request, 32-Bit Target**

XILINX®

Chapter 11

Other Bus Cycles

This chapter provides information about additional commands supported by the core interface. In addition to supporting memory read, memory write, I/O read, and I/O write commands, the core interface supports several other bus commands as both target and initiator. Many of these commands do not require significant additional design effort.

Supported Commands

Table 11-1 defines the PCI bus commands supported by the core interface.

Table 11-1: **PCI Bus Commands**

CBE [3:0]	Command	Initiator	Target
0000	Interrupt Acknowledge	Yes	No[a]
0001	Special Cycle	Yes	No
0010	I/O Read	Yes	Yes
0011	I/O Write	Yes	Yes
0100	Reserved	Ignore	Ignore
0101	Reserved	Ignore	Ignore
0110	Memory Read[b]	Yes	Yes
0111	Memory Write	Yes	Yes
1000	Reserved	Ignore	Ignore
1001	Reserved	Ignore	Ignore
1010	Configuration Read	Yes	Yes
1011	Configuration Write	Yes	Yes
1100	Memory Read Multiple[c]	Yes	Yes
1101	Dual Address Cycle	Yes	Yes
1110	Memory Read Line[c]	Yes	Yes
1111	Memory Write Invalidate[c]	Yes	Yes

a. Support is planned in a future release.
b. This command can only be used for a single DWORD transfer.
c. These command have fixed byte enables of 0h.

Table 11-2 defines the PCI bus commands supported by the core interface.

Table 11-2: **PCI-X Bus Commands**

CBE [3:0]	Command	Initiator	Target
0000	Interrupt Acknowledge	Yes	No[a]
0001	Special Cycle	Yes	No
0010	I/O Read	Yes	Yes
0011	I/O Write	Yes	Yes
0100	Reserved	Ignore	Ignore
0101	Reserved	Ignore	Ignore
0110	Memory Read Dword	Yes	Yes
0111	Memory Write	Yes	Yes
1000	Alias to Memory Read Block	Yes	Yes
1001	Alias to Memory Write Block	Yes	Yes
1010	Configuration Read	Yes	Yes
1011	Configuration Write	Yes	Yes
1100	Split Completion	Yes	Yes
1101	Dual Address Cycle	Yes	Yes
1110	Memory Read Block	Yes	Yes
1111	Memory Write Block	Yes	Yes

a. Support is planned in a future release.

Configuration Cycle Target

The core interface provides a mechanism for the user application to augment the standard configuration space with additional capability items or proprietary data structures.

The core interface automatically claims all valid configuration space accesses when `IDSEL_I` is asserted. All configuration transactions are DWORD transactions and are automatically terminated with disconnect with data on the first data phase.

If the user application does not implement additional data structures above configuration space address 7Fh, the user application must return zero on the data bus for configuration accesses in this range. It is possible to implement this behavior by designing the data path multiplexer to have a default output of zero.

If the user application does implement additional data structures above configuration space address 7Fh, these registers should be implemented in the same way other target registers are implemented, using `S_HIT[6]` as an indicator that an access is taking place.

Configuration Cycle Initiator

In a typical system, reading and writing of configuration registers is performed by a host bridge. The host bridge, sometimes called a north bridge, or root complex, is the hardware that bridges a host processor bus and the PCI-X Bus. The core interface is not appropriate

for use in this type of bridging application. It is intended for use in design of add-in cards. In practice, an add-in card which generates configuration cycles is unusual.

However, the core interface is capable of generating configuration cycles as an initiator. This is done by using the appropriate bus command when requesting a transaction. In almost all respects, configuration transactions resemble other DWORD operations such as I/O read and I/O write. However, there is one minor difference. During the address phase of configuration transactions, the core interface uses configuration address stepping as required by the specification. This extends the address phase over multiple clock cycles to allow the IDSEL signals (as received by other devices using a resistive connection to the AD bus bits) to stabilize at valid logic levels.

Note: Systems which use point-to-point IDSEL connections to the host bridge will not accept configuration cycles initiated by the core interface because the core interface has no mechanism to assert the point-to-point IDSEL signals. The PCI-X configuration cycle initiator only works when IDSEL is generated by resistive coupling to AD.

Special Cycle Initiator

Special cycle commands issued on the bus are broadcast cycles to one or more agents on the bus. These bus cycles are typically initiated to relay important system information across the bus. In order for this to be useful, there must be another agent on the bus capable of monitoring special cycles.

Note: The PCI-X target does not monitor special cycles and therefore cannot receive these messages from other bus agents.

The core interface is capable of generating configuration cycles as an initiator. This is done by using the appropriate bus command when requesting a transaction. In almost all respects, configuration transactions resemble other DWORD operations such as I/O read and I/O write. However, all special cycles terminate in master abort because they are never claimed by an agent on the bus.

Chapter 12

Error Detection and Reporting

The core interface generates and checks parity and reports errors as required by the *PCI Local Bus Specification*. This section will be expanded with additional information in a future release of the documentation.

XILINX®

Appendix

Protocol Compliance Checklist

The PCI Special Interest Group (PCI-SIG) provides checklists to assist in the design review of ASICs, components, system boards, add-in cards, and systems to verify their compliance with the *PCI Local Bus Specification, Revision 3.0* and *PCI-X Addendum 2.0a*.

In addition, checklists are used as a requirement to qualify a PCI-X product for the PCI-SIG Integrator's List by creating a paper testing trail for PCI compliance. Optionally, component, add-in card, and system board vendors may complete the appropriate sections and forward the checklist to their customers if desired.

For the purposes of completing a checklist, Xilinx considers the core interface core the *protocol* portion of a component.

This Appendix provides the complete protocol compliance checklist for the PCI-X core, divided into the following sections:

For electrical compliance information, see the respective data sheet for the FPGA family used to generate the physical implementation.

General PCI-X Protocol

Table 13-1: **General PCI-X Protocol Checklist**

Item	Description	Pass
XGP1.	The starting address of Configuration Read and Configuration Write transactions is aligned to a DWORD boundary	yes_✔ no___
XGP2.	For I/O and Memory Read DWORD transactions, the starting address must correspond to the first enabled byte (unless no byte enables are asserted). **This is the responsibility of the user application.**	yes_✔ no___
XGP3.	During the attribute phase C/BE#[3:0] and AD[31:0] contain the attributes.	yes_✔ no___
XGP4.	During the attribute phase C/BE#[7:4] and AD[63:32] are reserved and driven high by 64-bit initiators.	yes_✔ no___
XGP5.	The C/BE# bus is reserved (driven high) the clock after the attribute phase.	yes_✔ no___
XGP6.	All burst transactions include the byte count in the attributes.	yes_✔ no___
XGP7.	The byte count always indicates the number of bytes from the first byte of the transaction to the last byte of the sequence, inclusive.	yes_✔ no___
XGP8.	DWORD transactions do not use a byte count.	yes_✔ no___
XGP9.	Transactions using the I/O Read, I/O Write, Configuration Read, Configuration Write, Interrupt Acknowledge, Special Cycle, and Memory Read DWORD commands are initiated only as 32-bit transactions (REQ64# deasserted).	yes_✔ no___
XGP10.	Transactions using the I/O Read, I/O Write, Configuration Read, Configuration Write, Interrupt Acknowledge, Special Cycle, and Memory Read DWORD commands and Split Completion Messages are limited to a single data phase.	yes_✔ no___
XGP11.	Byte enables are included in the Requester Attributes for all DWORD transactions. Only bytes for which the byte enable is asserted are affected by the transaction.	yes_✔ no___
XGP12.	For DWORD transactions and Memory Write burst transactions only bytes for which the byte enables are asserted are affected by the transaction. **This is the responsibility of the user application.**	yes_✔ no___
XGP13.	The C/BE# bus is reserved and driven high during the single data phase of all DWORD transactions, and throughout all data phases of all burst transactions except Memory Write.	yes_✔ no___
XGP14.	DWORD transactions are permitted to have any combination of byte enables, including no byte enables asserted.	yes_✔ no___
XGP15.	Memory Write transactions are permitted to have any combination of byte enables between the starting and ending addresses, inclusive.	yes_✔ no___
XGP16.	Byte enables are deasserted for bytes before the starting address and after the ending address (if those addresses are not aligned to the width of the bus), except for Memory Write transactions when a 64-bit initiator's starting address is in the high 32-bits of the 64-bit bus. In that case, C/BE#[7:4] are copied to C/BE#[3:0]. **This is the responsibility of the user application.**	yes_✔ no___
XGP17.	For Memory Write transactions the byte count is not affected by whether byte enables are asserted.	yes_✔ no___
XGP18.	Device state machines are not affected by the states of DEVSEL#, TRDY#, and STOP# while the bus is idle (FRAME# and IRDY# both deasserted).	yes_✔ no___
XGP19.	If the device snoops a transaction, it does not drive any signals during the transaction (same as conventional PCI).	yes_ no___ n/a_✔
XGP20.	The device does not drive and receive bus signals at the same time.	yes_✔ no___

Table 13-1: **General PCI-X Protocol Checklist *(Continued)***

Item	Description	Pass
XGP21.	If the device generates interrupts, it supports message signaled interrupts and supports a 64-bit message address. If the device will be utilized in systems that do not support message signaled interrupts, it also implements hardware interrupts. **The MSI Capability Item is provided. It is the responsibility of the user application to use it.**	yes ✔ no___
XGP22.	If the device is intended for use on add-in cards, it supports PCI Power Management, as defined in PCI PM 1.1. **The PME Capability Item is provided. It is the responsibility of the user application to use it.**	yes ✔ no___
XGP23.	If the device supports power management and it is in D3Hot state but RST# remains deasserted, the device maintains its frequency and mode information (from the PCI-X initialization pattern).	yes ✔ no___
XGP24.	If the system supports PCI power management, it permits devices in D1, D2, and D3Hot to signal Split Response to a configuration transaction and initiate the corresponding Split Completion transaction (System requirement).	yes ✔ no___
XGP25.	Fast back-to-back transactions are not initiated by the device.	yes ✔ no___
XGP26.	DEVSEL# timing is measured from the address phase if the transaction uses a single address cycle and from the second address phase if the transaction uses a dual address cycle.	yes ✔ no___

Initiator

Table 13-2: **PCI-X Initiator Checklist**

Item	Description	Pass
XIP1.	The initiator begins a transaction (other than a configuration transaction) by asserting FRAME#.	yes ✔ no___
XIP2.	The initiator asserts FRAME# or deasserts REQ# within two to six clocks after GNT# is asserted and the bus is idle.	yes ✔ no___
XIP3.	The initiator deasserts FRAME# on the later of the following three conditions: 1. One clock before the last data phase. 2. Two clocks after the target asserts TRDY# (or terminates the transaction in some other way). 3. Eight clocks after the address phase if no target asserts DEVSEL# (Master Abort).	yes ✔ no___
XIP4.	The initiator asserts IRDY# two clocks after the attribute phase.	yes ✔ no___
XIP5.	The initiator deasserts IRDY# on the later of the following three conditions: 1. One clock after the last data phase. 2. Two clocks after the target asserts TRDY# (or terminates the transaction in some other way). 3. Eight clocks after the address phase if no target asserts DEVSEL# (Master Abort).	yes ✔ no___
XIP6.	The initiator uses the full address bus to indicate the starting address (including AD[1:0]) of memory and I/O transactions. Note this is the same as conventional I/O transactions but not conventional memory transactions.	yes ✔ no___
XIP7.	The initiator does not initiate a new Sequence using the same Tag until the previous Sequence is complete. **This is the responsibility of the user application.**	yes ✔ no___
XIP8.	If no target asserts DEVSEL# on or before the Subtractive decode time, the initiator ends the transaction as Master Abort.	yes ✔ no___
XIP9.	For write and split completion transactions, the initiator must drive data on the AD bus two clocks after the attribute phase.	yes ✔ no___
XIP10.	For write and split completion transactions, if the transaction is a burst with more than one data phase, the initiator advances to the second data value two clocks after the target asserts DEVSEL#.	yes ✔ no___
XIP11.	For write and split completion transactions, if the transaction is a burst with more than one data phase and the target inserts wait states, the initiator must toggle between the first and second data values until the target asserts TRDY# (or terminates the transaction).	yes ✔ no___
XIP12.	The initiator terminates the transaction when the byte count is satisfied.	yes ✔ no___
XIP13.	Burst transactions do not cross the end of the 64-bit address space. **This is the responsibility of the user application.**	yes ✔ no___
XIP14.	If a target responds with immediate completion to a burst transaction for more than a single data phase, the initiator does not signal a disconnect at any point except an ADB. (The Sequence terminates when the byte count expires).	yes ✔ no___
XIP15.	If the initiator intends to disconnect the transaction on the first ADB, and the first ADB is less than four data phases from the starting address, the initiator adjusts the byte count to terminate the transaction on that ADB. **This is the responsibility of the user application.**	yes ✔ no___
XIP16.	For a burst transaction that would cross the next ADB if the target signals Disconnect at Next ADB four data phases before an ADB or on the first data phase, the initiator deasserts FRAME# two clocks later and disconnects the transaction on the ADB.	yes ✔ no___
XIP17.	The initiator transfers data on each data phase in which the target signals Disconnect at Next ADB the same as if the target had signaled Data Transfer.	yes ✔ no___
XIP18.	The default Latency Timer value for the initiator in PCI-X mode is 64.	yes ✔ no___

Table 13-2: **PCI-X Initiator Checklist *(Continued)***

Item	Description	Pass
XIP19.	The initiator disconnects the current transaction on the next ADB if the Latency Timer expires and GNT# is deasserted at least four data phases before the ADB.	yes_✔ no___
XIP20.	The initiator does not repeat a transaction that the target terminated by signaling Target Abort. (Same as conventional PCI). **This is the responsibility of the user application.**	yes_✔ no___
XIP21.	The device does not set the No Snoop and Relaxed Order attribute bits on any transaction that is not a memory transaction. **This is the responsibility of the user application.**	yes_✔ no___
XIP22.	If the device initiates Special Cycle transactions, those transactions have no initiator wait states, have only one data phase, and have all byte enables asserted.	yes_✔ no___
XIP23.	If the device initiates Special Cycle transactions, those transactions do not set the Received Master Abort bit in the Status register.	yes_✔ no___
XIP24.	The initiator does not use reserved commands (0100b, 0101b, 1000b, and 1001b).	yes_✔ no___

Target

Table 13-3: **PCI-X Target Checklist**

Item	Description	Pass
XTP1.	Memory address ranges (including those assigned through Base Address registers) assigned to the device are no smaller than 128 bytes. **Depends on user configuration of interface.**	yes_✔ no___
XTP2.	The target ignores (must not assert DEVSEL# or change state) any transactions using a reserved command.	yes_✔ no___
XTP3.	The target treats the Alias to Memory Read Block command (1000b) as if it were a Memory Read Block (1110b) and Alias to Memory Write Block command (1001b) as if it were a Memory Write Block (1111b).	yes_✔ no___
XTP4.	The target claims the transaction by asserting DEVSEL# and leaving TRDY# and STOP# deasserted, using decodes A, B, C, or Subtractive.	yes_✔ no___
XTP5.	DEVSEL# is never asserted earlier than the clock after the attribute phase.	yes_✔ no___
XTP6.	After the target asserts DEVSEL#, it completes the transaction with one or more data phases by signaling one or more of the following: Split Response, Target Abort, Single Data Phase Disconnect, Wait State, Data Transfer, Retry, or Disconnect at Next ADB.	yes_✔ no___
XTP7.	The target never inserts an odd number of wait states for burst write and split completion transactions.	yes_✔ no___
XTP8.	The target does not signal Wait State after the first data phase.	yes_✔ no___
XTP9.	If the target signals Split Response, or Retry, or signals Target Abort in the first data phase, the target does so within eight clocks of the assertion of FRAME#. **This is the responsibility of the user application.**	yes_✔ no___
XTP10.	If the target signals Single Data Phase Disconnect, or signals Data Transfer or Disconnect at Next ADB in the first data phase, the target does so within 16 clocks of the assertion of FRAME# (except during device initialization time, Trhfa, and when the system is copying an expansion ROM image from the device to system memory). **This is the responsibility of the user application.**	yes_✔ no___
XTP11.	If the PCI-X target signals Data Transfer (with or without preceding wait states), the target disconnects the transaction only on an ADB (until the byte count is satisfied).	yes_✔ no___
XTP12.	If the target has signaled Disconnect at Next ADB, it continues to do so (or signals Target Abort) until the end of the transaction.	yes_✔ no___
XTP13.	If the target signals Split Response, Single Data Phase Disconnect, or Retry, it does so only on the first data phase (with or without preceding wait states).	yes_✔ no___
XTP14.	The target never signals Split Response on burst write or Split Completion transactions. **This is the responsibility of the user application.**	yes_✔ no___
XTP15.	The target deasserts DEVSEL#, STOP#, and TRDY# one clock after the last data phase (if they are not already deasserted) and floats them one clock after that.	yes ✔ no___
XTP16.	The target does not store state information about any type of transaction other than Split Requests. (The target does not perform Delayed Transactions in PCI-X mode).	yes_✔ no___
XTP17.	If the target signals Disconnect at Next ADB less than 4 data phase from an ADB but not on the first data phase, the target expects the transaction to continue past the first ADB and disconnect on the next ADB or until the byte count expires.	yes_✔ no___
XTP18.	If the target signals Disconnect at Next ADB 4 or more data phases from an ADB, the target expects the transaction to disconnect at that ADB.	yes_✔ no___
XTP19.	If the device responds to Special Cycle transactions, the device does not assert bus signals in response to detecting a Special Cycle transaction.	yes___ no___ n/a_✔
XTP20.	If the device responds to Interrupt Acknowledge transactions, the device responds to the bus transaction regardless of the address value.	yes_✔ no___

Table 13-3: **PCI-X Target Checklist *(Continued)***

Item	Description	Pass
XTP21.	If the target is a bridge, it responds to Split Completions only with Target Abort, Wait State, Data Transfer, Retry, and Disconnect at Next ADB. If the target is not a bridge, it responds to Split Completions only with Target Abort, Wait State, and Data Transfer. **This is the responsibility of the user application.**	yes ✔ no___
XTP22.	If the target is designed to signal Single Data Phase Disconnect for any memory write transaction with an address four or less data phases before an ADB, the target never signals Data Transfer for any other memory write transaction that begins four or less data phases from that same ADB. **This is the responsibility of the user application.**	yes ✔ no___

Simple Device

Table 13-4: **PCI-X Simple Device Checklist**

Item	Description	Pass
XSP1.	After initialization time, the device is able to accept a memory and I/O write transaction within the Maximum Completion Time Limit of 2us (267 clocks for 133 MHz mode, 200 clocks for 100 MHz mode, and 133 clocks for 66 MHz mode). **This is the responsibility of the user application.**	yes ✔ no___
XSP2.	After initialization time, the device accepts memory write transactions even while executing a previous Split Transaction. **This is the responsibility of the user application.**	yes ✔ no___
XSP3.	After initialization time, the device accepts all Split Completion transactions that correspond to the device's outstanding Split Requests without signaling Retry or Disconnect at Next ADB. **This is the responsibility of the user application.**	yes ✔ no___

Bus Arbitration

Some bus arbitration rules apply to all initiators and some apply only if the device includes the bus arbiter, as indicated below.

Table 13-5: **PCI-X Bus Arbitration Checklist**

Item	Description	Pass
XAP1.	If a device signals Split Response, arbitration within that device fairly allows the initiation of the Split Completion.	yes 3 no___
XAP2.	REQ# and GNT# signals are registered by the initiator.	yes 3 no___
XAP3.	The initiator starts a new transaction (drive the AD bus, etc.) on any clock N only if the initiator's GNT# was asserted on clock N-2, and either the bus was idle (FRAME# and IRDY# were both deasserted) on clock N-2 or FRAME# was deasserted and IRDY# was asserted on clock N-3.	yes ✔ no___
XAP4.	If the device includes multiple sources of initiator activity, all of these sources share a single REQ# and GNT# pair.	yes ✔ no___
XAP5.	If GNT# is asserted and the bus is idle for four consecutive clocks, the device (where the bus is parked) actively drives the bus (AD, C/BE#, PAR, and (if a 64-bit device) PAR64) no later than the sixth clock.	yes ✔ no___
XAP6.	A parked initiator must assert REQ# if it intends to execute more than a single transaction.	yes ✔ no___
XAP7.	If GNT# is deasserted when the Latency Timer expires, the device disconnects the current transaction as soon as possible. (This is typically the next ADB, but in some cases is the following ADB or when the byte count expires).	yes ✔ no___
XAP8.	The arbiter is fair to all devices.	yes___ no___ n/a ✔

Table 13-5: **PCI-X Bus Arbitration Checklist *(Continued)***

Item	Description	Pass
XAP9.	All REQ# and GNT# signals are registered by the arbiter.	yes__ no___ n/a_ ✔
XAP10.	The arbitration algorithm does not depend on the initiator deasserting REQ# after a target termination (STOP# asserted).	yes__ no___ n/a_ ✔
XAP11.	The arbitration algorithm allows the initiator to deassert REQ# on any clock independent of whether GNT# is asserted.	yes__ no___ n/a_ ✔
XAP12.	The arbitration algorithm allows the initiator to deassert REQ# without initiating a transaction after GNT# is asserted.	yes__ no___ n/a_ ✔
XAP13.	The arbiter allows each device a fair opportunity to execute a configuration transaction. To allow a device to execute a configuration transaction, after the arbiter asserts GNT#, the arbiter keeps it asserted for a minimum of five clocks while the bus is idle, or until the initiator asserts FRAME# or deasserts REQ#. (Note that this rule does not require that every assertion of GNT# must be for a minimum of five clocks. Only that every device must eventually have a turn to execute a configuration transaction).	yes__ no___ n/a_ ✔
XAP14.	If the arbiter deasserts GNT# to one device, it waits until the next clock to assert GNT# to another device.	yes__ no___ n/a_ ✔
XAP15.	In PCI hot plug systems, the arbiter coordinates with the Hot Plug Controller to prevent hot plug operations from interfering with other bus transactions.	yes__ no___ n/a_ ✔
XAP16.	If no initiators request the bus, the arbiter parks the bus at a device that is capable of being an initiator.	yes__ no___ n/a_ ✔

Configuration

Table 13-6: **PCI-X Configuration Checklist**

Item	Description	Pass
XCP1.	Initiator drives the address for four clocks before asserting FRAME# for configuration transactions when in PCI-X mode.	yes_✔ no___
XCP2.	Initiator includes in the transaction the target device number in AD[15:11] of the address phase. **This is the responsibility of the user application.**	yes_✔ no___
XCP3.	Initiator includes in the transaction the target bus number in AD[7:0] of the attribute phase. **This is the responsibility of the user application.**	yes_✔ no___
XCP4.	If the device is not a PCI-X bridge and it is the target of a Type 0 Configuration Write transaction, it stores its device number (from the address phase) and the bus number (from the attribute phase) in its PCI-X Status registers. If the device is a PCI-X bridge and it is the target of a Type 0 Configuration Write transaction, it stores its device number (from the address phase) in its PCI-X Status registers.	yes_✔ no___
XCP5.	If the device is not a PCI-X bridge, reading the device's PCI-X status register following PCI Reset (before writing to the device) returns 1Fh for the Device Number field and FFh for the Bus Number field.	yes_✔ no___
XCP6.	If the device is not a bridge, it does not respond to Type 1 Configuration transactions.	yes_✔ no___
XCP7.	If the arbiter asserts GNT# to a device on clock N-2, the device starts driving the bus for a configuration transaction (on clock N, N+1, or N+2), and the arbiter deasserts GNT# before clock N+3, the device does not continue the configuration transaction. It floats the bus two clocks after GNT# is deasserted.	yes_✔ no___
XCP8.	The system quiesces all devices on a bus segment before changing the Bus Number in the source bridge. (System requirement).	yes___ no___ n/a_✔
XCP9.	The system executes a write to the configuration space of each device after the bus segment's Bus Number has been changed. (System requirement).	yes___ no___ n/a_✔
XCP10.	All PCI-X devices must have the Capabilities List bit (bit 4) of the Status register set to 1.	yes_✔ no___
XCP11.	All PCI-X devices support the PCI-X Capabilities List Item.	yes_✔ no___
XCP12.	When in PCI-X mode, the Fast Back-to-Back bit (bit 7) of the Status register must be set to 0.	yes_✔ no___

Error

Table 13-7: **PCI-X Error Checklist**

Item	Description	Pass
XEP1.	The device asserts PERR# (if enabled) on the second clock after PAR64 and PAR are driven if the device is receiving data (that is, the target of a write or split completion and the initiator of a read) and detects a parity error.	yes_✔ no___
XEP2.	During read transactions, the target drives PAR64 (if responding as a 64-bit device) and PAR on clock N+1 for the read data it drives on clock N and the C/BE# bus driven by the initiator on clock N-1.	yes_✔ no___
XEP3.	During write and split completion transactions, the initiator drives PAR64 (if initiating as a 64-bit device) and PAR on clock N+1 for the write data and the C/BE# bus it drives on clock N.	yes_✔ no___
XEP4.	The device services data parity error conditions for transactions it initiates in one of the following ways: 1. The device and software driver attempt to recover from a data parity error condition. 2. The device asserts SERR# (if enabled) on a data parity error condition.	yes_✔ no___
XEP5.	The device asserts SERR# (if enabled) if it detects a parity error on an attribute phase, independent of whether the device decodes its address during the address phase.	yes_✔ no___

Table 13-7: **PCI-X Error Checklist *(Continued)***

Item	Description	Pass
XEP6.	For split transactions, the requester sets the Master Data Parity Error bit in the Status register for data parity errors on either the split request or the split completion, and upon receipt of a split completion message indicating split write data parity error.	yes ✔ no
XEP7.	If data parity error recovery is disabled, the device asserts SERR# when a data parity error occurs.	yes ✔ no
XEP8.	After a device asserts PERR#, the device drives PERR# high for one clock cycle at the end of the bus operation and before the turnaround cycle.	yes ✔ no
XEP9.	The target of a write or a split completion transaction does not check parity while it is inserting initial wait states.	yes ✔ no
XEP10.	If a Master Abort or Target Abort occurs during a split completion, the initiator (completer or bridge) discards the split completion. The completer additionally sets the split completion discarded bit and asserts SERR# (if enabled) if the original split request was a read with side effects. If the initiator is a bridge, the split completion discarded bit must be set and SERR# asserted (if enabled) regardless of the address of the original split request. **This is the responsibility of the user application.**	yes ✔ no
XEP11.	If a bridge signals Data Transfer on a non-posted write on the originating bus, and the transaction has a data parity error, the bridge discards the transaction.	yes no n/a ✔
XEP12.	If a bridge detects a data parity error on a split completion message, it discards the transaction and asserts SERR# (if enabled).	yes no n/a ✔

Bus Width

Table 13-8: **PCI-X Bus Width Checklist**

Item	Description	Pass
XWP1.	The device implements the width of the address independent of the width of the data transfer.	yes ✔ no___
XWP2.	If the device initiates memory transactions, it is capable of generating 64-bit memory addresses.	yes ✔ no___
XWP3.	If the device requests a memory range through a Base Address register, that Base Address register is 64-bits wide.	yes ✔ no___
XWP4.	If the device initiates a transaction and the address is greater than or equal to 4 Gb, the device generates a dual address cycle.	yes ✔ no___
XWP5.	The attribute phase is always a single clock.	yes ✔ no___
XWP6.	Only burst transactions (Split Completions or memory commands other than Memory Read DWORD) use 64-bit data transfers.	yes ✔ no___
XWP7.	AD[63:32] and C/BE#[7:4] are driven high during the attribute phase of transactions from a 64-bit initiator.	yes ✔ no___
XWP8.	A reserved bus is not specified and is ignored by the device receiving the bus (same as conventional PCI).	yes ✔ no___
XWP9.	If the address of a transaction is less than 4 Gb, the transaction uses a single address cycle. For 64-bit devices, AD[63:32] and C/BE#[7:4] are reserved during the address phase of these transactions (same as conventional PCI).	yes ✔ no___
XWP10.	If the initiator is 64-bits wide and the address of a transaction is 4 Gb or greater, AD[63:0] contain the full address, C/BE#[3:0] contain the dual address cycle command, and C/BE#[7:4] contain the transaction command during the first address phase (same as conventional PCI).	yes ✔ no___
XWP11.	If the initiator is 64-bits wide and the address of a transaction is 4Gb or greater, AD[63:32] and AD[31:0] contain duplicate copies of the upper half of the address, and C/BE#[7:4] and C/BE#[3:0] contain duplicate copies of the transaction command during the second address phase (same as conventional PCI).	yes ✔ no___
XWP12.	If the initiator is 32-bits wide and the address of a transaction is 4 Gb or greater, AD[31:0] contain the lower half of the address during the first address phase and contain the upper half of the address during the second address phase (same as conventional PCI).	yes ✔ no___
XWP13.	If the initiator is 32-bits wide and the address of a transaction is 4 Gb or greater, C/BE#[3:0] contain the dual address cycle command during the first address phase and contain the transaction command during the second address phase (same as conventional PCI).	yes ✔ no___
XWP14.	DEVSEL# timing designations measure from the second address phase of a transaction with a dual address cycle (same as conventional PCI).	yes ✔ no___
XWP15.	Split Completions always have a single address phase both for 64-bit and 32-bit initiators.	yes ✔ no___
XWP16.	A 64-bit initiator asserts REQ64# with the same timing as FRAME# to request a 64-bit transfer. It deasserts REQ64# with FRAME# at the end of the transaction (same as conventional PCI).	yes ✔ no___
XWP17.	The device includes a configuration bit in the PCI-X Status register to indicate the width of its interface. If installed on an add-in card the bit indicates the narrower of the widths of the device or the card interfaces.	yes ✔ no___
XWP18.	Allowable disconnect boundaries are unaffected by the width of the data transfer. A 32-bit transfer has twice as many data phases between two ADBs.	yes ✔ no___
XWP19.	If the transfer is 64-bits wide (REQ64# asserted) and AD[2] of the starting byte address is 1 for memory write and Split Completion transactions, the initiator duplicates the upper 32-bits of data on AD[63:32] and AD[31:0], and the upper 4-bits of byte enables on C/BE#[7:4] and C/BE#[3:0] for the first data phase.	yes ✔ no___
XWP20.	If the transfer is 64 bits wide (REQ64# asserted), and AD[2] of the starting byte address is 1 for memory write and Split Completion transactions, and the target asserts ACK64# when it asserts DEVSEL#, and the target signals Wait State, the initiator must toggle between the first and second QWORD data phases.	yes ✔ no___

Table 13-8: **PCI-X Bus Width Checklist *(Continued)***

Item	Description	Pass
XWP21.	If the transfer is 64 bits wide (REQ64# asserted), and AD[2] of the starting byte address is 1 for memory write and Split Completion transactions, and the target does not assert ACK64# when it asserts DEVSEL#, and the target signals Wait State, the initiator must toggle between the first and second DWORD data phases.	yes_✔ no___
XWP22.	The width of the transaction is established by the state of ACK64# on the first clock that DEVSEL# is asserted. Deassertion of ACK64# and DEVSEL# for Single Data Phase Disconnect does not change the data width established.	yes_✔ no___
XWP23.	The Split Completion address for a DWORD request and for a Split Completion Message is always 0, so the data or Split Completion Message always appears on AD[31:0].	yes_✔ no___

Split Transaction

Table 13-9: **PCI-X Split Transaction Checklist**

Item	Description	Pass
XST1.	If the completer signals Split Response, it initiates one or more Split Completion transactions with that Requester ID and Tag. **This is the responsibility of the user application.**	yes ✔ no___
XST2.	The completer never creates a Split Completion transaction containing both read data and a Split Completion Message. **This is the responsibility of the user application.**	yes ✔ no___
XST3.	If the completer returns read data, the Completer returns all the data (full byte count) unless an error occurs. **This is the responsibility of the user application.**	yes ✔ no___
XST4.	The completer never uses a byte count in the Split Completion other than the full remaining byte count for the Sequence, except to disconnect the Split Completion on the first ADB of the Sequence. **This is the responsibility of the user application.**	yes ✔ no___
XST5.	If the completer disconnects the Split Completion on the first ADB by adjusting the byte count, the completer sets the Byte Count Modified attribute for the affected transactions. **This is the responsibility of the user application.**	yes ✔ no___
XST6.	Each time the Split Completion resumes after a disconnection at an ADB, the initiator adjusts the byte count (and starting address) to indicate the number of bytes remaining in the Sequence. **This is the responsibility of the user application.**	yes ✔ no___
XST7.	The requester never terminates a Split Completion transaction with Split Response, Single Data Phase Disconnect, Retry, or Disconnect at Next ADB. **This is the responsibility of the user application.**	yes ✔ no___
XST8.	The requester correctly handles a Split Response termination and corresponding Split Completion for Memory Read Block commands.	yes ✔ no___
XST9.	The requester correctly handles a Split Response termination and corresponding Split Completion for Memory Read DWORD commands.	yes ✔ no___
XST10.	The requester correctly handles a Split Response termination and corresponding Split Completion for I/O Read commands.	yes ✔ no___
XST11.	The requester correctly handles a Split Response termination and corresponding Split Completion for I/O Write commands.	yes ✔ no___
XST12.	The requester correctly handles a Split Response termination and corresponding Split Completion for Configuration Read commands.	yes ✔ no___
XST13.	The requester correctly handles a Split Response termination and corresponding Split Completion for Configuration Write commands.	yes ✔ no___
XST14.	The requester correctly handles a Split Response termination and corresponding Split Completion for Interrupt Acknowledge commands.	yes ✔ no___
XST15.	Completers that signal Split Response return Split Completion even when their Bus Master bit is cleared in configuration space. **This is the responsibility of the user application.**	yes ✔ no___
XST16.	Requesters that have Split Completions outstanding for more than one Sequence at a time function correctly if Split Completions for different Sequences are returned in a different order than the Split Requests. **This is the responsibility of the user application.**	yes ✔ no___
XST17.	Completers generate the Split Completion address and Split Completion attributes according to the rules in Sections 2.10.3 and 2.10.4.	yes ✔ no___
XST18.	If the request is a DWORD write transaction, or if the completer encounters an error while executing the request, the completer sends a Split Completion Message to the requester. **This is the responsibility of the user application.**	yes ✔ no___
XST19.	Completers that signal Split Response detect transactions which exceed their address range and send a Split Completion Message indicating the error (Class 2h, Index 00h) after transferring data up to their device boundary. **This is the responsibility of the user application.**	yes ✔ no___

Table 13-9: **PCI-X Split Transaction Checklist *(Continued)***

Item	Description	Pass
XST20.	The requester accepts all split completions from its own split requests.	yes_✔ no___

Interoperability and Initialization

Table 13-10: **Interoperability and Initialization Checklist**

Item	Description	Pass
XIN1.	If placed into a bus with a 33 MHz conventional PCI device, the device will operate normally (includes the acceptance of configuration transactions) in conventional PCI mode.	yes_✔ no___
XIN2.	If the device is a PCI-X 133 device and it is placed in a bus with at least one PCI-X 66 device, the device operates normally (including accepting configuration transactions) in PCI-X 66 mode.	yes ✔ no___
XIN3.	If the device is a PCI-X 66 device, it operates in PCI-X mode at any frequency from 50 MHz to 66 MHz.	yes_✔ no___
XIN4.	If the device is a PCI-X 133 device, it operates in PCI-X mode at any frequency from 50 MHz to 133 MHz.	yes_✔ no___
XIN5.	A PCI-X 133 device operates properly in a hot-plug system if the system asserts RST# and changes the bus frequency to enable the addition of a PCI-X 66 card in an adjacent slot.	yes_✔ no___
XIN6.	A PCI-X 133 device operates properly in a hot-plug system if the system asserts RST# and changes the mode to conventional PCI to enable the addition of a conventional PCI card in an adjacent slot.	yes_✔ no___
XIN7.	A PCI-X system provides the PCI-X initialization pattern on the DEVSEL#, STOP#, TRDY# signals at the rising edge of RST#, indicating the operating frequency of the system.	yes__ no___ n/a_✔
XIN8.	If the device is installed on an add-in card, the 133 MHz Capable bit in the PCI-X Status register is set consistently with the wiring of the PCIXCAP pin on the card edge connector. That is, if the card is a PCI-X 133 card, the bit is set to 1 and PCIXCAP is connected to ground through a capacitor. If the card is a PCI-X 66 card, the bit is cleared to 0 and PCIXCAP is connected to ground through a resistor and a capacitor.	yes_✔ no___

Configuration Organization

Table 13-11: **Configuration Organization Checklist**

Item	Description	Pass
CO1.	Does each PCI resource have a configuration space based on the defined 256 byte template, with a predefined 64 byte header and a 192 byte device specific region?	yes ✔ no___
CO2.	Do all functions in the device support the Vendor ID, Device ID, Command, Status, Header Type and Class Code fields in the header?	yes_✔ no___
CO3.	Is the configuration space available for access at all times?	yes ✔ no___
CO4.	Are writes to reserved registers or read only bits completed normally and the data discarded?	yes_✔ no___
CO5.	Are reads to reserved or unimplemented registers, or bits, completed normally and a data value of 0 returned?	yes ✔ no___
CO6.	Is the vendor ID a number allocated by the PCI-SIG?	yes_✔ no___
CO7.	Does the Header Type field have a valid encoding?	yes ✔ no___
CO8.	Do multi-byte transactions access the appropriate registers and are the registers in “little endian” order?	yes_✔ no___
CO9.	Are all read only register values within legal ranges?	yes ✔ no___

Table 13-11: **Configuration Organization Checklist *(Continued)***

Item	Description	Pass
CO10.	Is the class code in compliance with the definition in Appendix D?	yes ✔ no
CO11.	Is the predefined header portion of configuration space accessible as bytes, words, and dwords?	yes ✔ no
CO12.	Is the device a multifunction device?	yes no ✔
CO13.	If the device is multifunction, are configuration space accesses to unimplemented functions ignored?	yes no n/a ✔
CO14.	Are Subsystem ID and Subsystem Vendor ID fields are loaded and valid prior to any system software accessing these fields including after boot and resuming from a sleeping state and are not initialized by Expansion ROM code?	yes ✔ no
CO15.	If the function uses extended Capabilities is bit 4 of the status register hardwired to 1 and is the Capabilities List pointer implemented?	yes ✔ no
CO16.	If the function implements Message Signaled Interrupts, is a capability list used to indicate support and does the function not use INTx# pins when MSI is enabled?	yes ✔ no
CO17.	If the function supports MSI-X, the MSI-X Capability Structure points to an MSI-X Table structure and an MSI-X Pending Bit Array (PBA) structure. The value in the CAP_ID field (bits 7:0) of the MSI-X Capability Structure is 11h and identifies the function as being MSI-X capable. This field is read only.	yes no n/a ✔

Indicate either N/A (Not Applicable) or Implemented by placing a check in the appropriate box. Grayed areas indicate invalid selections. This table should be completed for each function in a multifunction device.

Table 13-12: **Functions Checklist**

Location	Name	Required/Optional	N/A	Implemented
00h-01h	Vendor ID	Required		✔
02h-03h	Device ID	Required		✔
04h-05h	Command	Required		✔
06h-07h	Status	Required		✔
08h	Revision ID	Required		✔
09h-0Bh	Class Code	Required		✔
0Ch	Cache Line Size	Required by master devices/functions that can generate Memory Write and Invalidate.		✔
0Dh	Latency Timer	Required by master devices/functions that can burst more than two data phases.		✔
0Eh	Header Type	If the device is multi-functional, then bit 7 must be set to a 1. The remaining bits are required to have a defined value.		✔
0Fh	Built In Self Test	Optional	✔	
10h-27h	Base Address Registers	One or more required for any address allocation.		✔
28h-2Bh	Cardbus CIS Pointer	Optional		✔
2CH-2Dh	Subsystem Vendor ID	Required		✔
2Eh-2Fh	Subsystem ID	Required		✔
30h-33h	Expansion ROM Base Address Register	Required for devices/functions that have expansion ROM.		✔
34h	Capabilities Pointer	Optional		✔
35h-3Bh	Reserved	Required		✔
3Ch	Interrupt Line	Required by devices/functions that use an interrupt pin.		✔
3Dh	Interrupt Pin	Required by devices/functions that use an interrupt pin.		✔
3Eh	Minimum Grant	Optional		✔
3Fh	Maximum Latency	Optional		✔

Configuration Device Control

Table 13-13: **Configuration Device Control**

Item	Description	Pass
DC1.	When the command register is loaded with a 0000h is the device/function logically disconnected from the PCI bus, with the exception of configuration accesses? (Devices in boot code path are exempt).	yes_✔_ no___
DC2.	Is the device/function disabled after the assertion of RST#? (Devices in boot code path are exempt).	yes_✔ no___

In the following tables for Command and Status Registers, an "3" in the Target or Master columns indicates implementation; N/A indicates that the bit is not applicable, but must return a 0 when read.

Table 13-14: **Command and Status Registers Checklist**

Bit	Name	Required/Optional	N/A	Target	Master
0	I/O Space	Required if device/function has registers mapped into I/O space.		✔	N/A

Table 13-14: **Command and Status Registers Checklist *(Continued)***

Bit	Name	Required/Optional	N/A	Target	Master
1	Memory Space	Required if device/function responds to memory space accesses.		3v	N/A
2	Bus Master	Required		N/A	✔
3	Special Cycles	Required for devices/functions that can respond to Special Cycles.	✔		N/A
4	Memory Write and Invalidate	Required for devices/functions that generate Memory Write and Invalidate cycles.		N/A	✔
5	VGA Palette Snoop	Required for VGA or graphical devices/functions that snoop VGA color palette.	✔		N/A
6	Parity Error Response	Required unless exempted per section 3.7.2.		✔	✔
7	Wait Cycle Control	Optional		✔	✔
8	System Error	Required if device/function has SERR# pin.		✔	✔
9	Fast Back-to-Back	Required if Master device/function can support fast back-to-back cycles among different targets.	✔	N/A	
10	Interrupt Disable	Required		✔	✔
11-15	Reserved	Required		✔	✔

Configuration Device Status

Table 13-15: **Configuration Device Status Checklist**

Item	Description	Pass
DS1.	Do all implemented read/write bits in the Status reset to 0?	yes_✔ no___
DS2.	Are read/write bits set to a 1 exclusively by the device/function?	yes_✔ no___
DS3.	Are read/write bits reset to a 0 when RST# is asserted?	yes_✔ no___
DS4.	Are read/write bits reset to a 0 by writing a 1 to the bit?	yes_✔ no___

Table 13-16: **Required/Optional Checklist**

Bit	Name	Required/Optional	N/A	Target	Master
0-2	Reserved	Required		✔	✔
3	Interrupt Status	Required		✔	✔
4	Capabilities List	Optional		✔	✔
5	66 Mhz Capable	Required for 66Mhz capable devices.		✔	✔
6	Reserved	Required		✔	✔
7	Fast Back-to-Back Capable	Optional	3		N/A
8	Data Parity Detected	Required		N/A	✔
9-10	DEVSEL# Timing	Required		✔	N/A
11	Signaled Target Abort	Required for devices/functions that are capable of signaling target abort.		✔	N/A

Table 13-16: **Required/Optional Checklist *(Continued)***

Bit	Name	Required/Optional	N/A	Target	Master
12	Received Target Abort	Required		N/A	✔
13	Received Master Abort	Required		N/A	✔
14	Signaled System Error	Required for devices/functions that are capable of asserting SERR#.		✔	✔
15	Detected Parity Error	Required unless exempted per section 3.7.2		✔	✔

Configuration Base Addresses

Table 13-17: **Configuration Base Addresses Checklist**

Item	Description	Pass
BA1.	If the device/function uses Expansion ROM, does it implement the Expansion ROM Base Address Register?	yes_ ✔ no___
BA2.	Do all Base Address registers asking for IO space request 256 bytes or less? **Depends on user configuration of interface.**	yes__ no___ n/a_ ✔
BA3.	If the device/function has an Expansion ROM Base Address register, does the memory enable bit in the Command register have precedence over the enable bit in the Expansion ROM base Address register?	yes_ ✔ no___
BA4.	Does the device/function use any address space (memory or IO) other than that assigned using Base Address registers? (that is, does the device/function hard-decode any addresses?) Note: If the answer is yes, you must list decoded addresses as explanations at the end of this section. **Depends on user configuration of interface.**	yes__ no___ n/a_ ✔
BA5.	Does the device/function decode all 32-bits of IO space? **The upper 28 bits of address are decoded by base address registers.** **The lower 4 bits must be decoded by the user application.**	yes__ no_ ✔
BA6.	If the device/function has an Expansion ROM Base Address register, is the size of the memory space requested 16MB or smaller?	yes_ ✔ no___

VGA Devices

Table 13-18: **VGA Devices Checklist**

Item	Description	Pass
VG1.	Is palette snoop implemented, including bit in Command register?	yes__ no___ n/a_ ✔
VG2.	Is Expansion ROM Base Address register implemented and provide full relocatability of the expansion ROM? (The device must NOT do a hard decode of 0C0000h).	yes__ no___ n/a_ ✔
VG3.	Does the device come up disabled? (Bottom three bits of Command register must be initialized to zero on power-up and RST#).	yes__ no___ n/a_ ✔
VG4.	Does Class Code field indicate VGA device? (value of 030000h).	yes__ no___ n/a_ ✔
VG5.	Does the device hard-decode only standard ISA VGA addresses and their aliases? (IO addresses 3B0h through 3BBh, 3C0h through 3DFh, Memory addresses 0A0000h through 0BFFFFh)	yes__ no___ n/a_ ✔
VG6.	Does the device use Base Address Registers to allocate needed space other than standard ISA VGA Addresses? (for example, for a linear FRAME buffer)	yes__ no___ n/a ✔

General Component Protocol Checklist (PCI Master)

The following checklist is to filled out as a general verification of the protocol compliance of the IUT. This checklist applies to all master operations.

Table 13-19: **General Component Protocol Checklist (PCI Master) Checklist**

Item	Description	Pass
MP1.	All sustained tri-state signals are driven high for one clock before being tri-stated. (2.1)	yes_ ✔ no___

Table 13-19: **General Component Protocol Checklist (PCI Master) Checklist *(Continued)***

Item	Description	Pass
MP2.	IUT always asserts all byte enables during each data phase of a Memory Write and Invalidate cycle. (3.1.1)	yes_✔ no___
MP3.	IUT always uses Linear Burst Ordering for Memory Write and Invalidate cycles. (3.1.1)	yes_✔ no___
MP4.	IUT always drives IRDY# when data is valid during a write transaction. (3.2.1)	yes_✔ no___
MP5.	IUT only transfers data when both IRDY# and TRDY# are asserted on the same rising clock edge. (3.2.1)	yes_✔ no___
MP6.	Once the IUT asserts IRDY# it never changes FRAME# until the current data phase completes. (3.2.1)	yes_✔ no___
MP7.	Once the IUT asserts IRDY# it never changes IRDY# until the current data phase completes. (3.2.1)	yes_✔ no___
MP8.	IUT never uses reserved burst ordering (AD[1:0] = “01”). (3.2.2)	yes_✔ no___
MP9.	IUT never uses reserved burst ordering (AD[1:0] = “11”). (3.2.2)	yes_✔ no___
MP10.	IUT always ignores configuration command unless IDSEL is asserted and AD[1:0] are “00”. (3.2.2)	yes_✔ no___
MP11.	The IUT's AD lines are driven to stable values during every address and data phase. (3.2.4)	yes_✔ no___
MP12.	The IUT's C/BE# output buffers remain enabled from the first clock of the data phase through the end of the transaction. (3.3.1)	yes_✔ no___
MP13.	IUT drives C/BE# lines with valid byte enable information during the entire data phase. (3.3.1)	yes_✔ no___
MP14.	IUT never deasserts FRAME# unless IRDY# is asserted or will be asserted (3.3.3.1)	yes_✔ no___
MP15.	IUT never deasserts IRDY# until at least one clock after FRAME# is deasserted. (3.3.3.1)	yes_✔ no___
MP16.	Once the IUT deasserts FRAME# it never reasserts FRAME# during the same transaction. (3.3.3.1)	yes_✔ no___
MP17.	IUT never terminates with master abort once target has asserted DEVSEL#. (3.3.3.1)	yes_✔ no___
MP18.	IUT never signals master abort earlier than 5 clocks after FRAME# was first sampled asserted. (3.3.3.1)	yes_✔ no___
MP19.	IUT always repeats an access exactly as the original when terminated by retry. (3.3.3.2.2) **The retry process is controlled by logic in the user application.**	yes___ no___ n/a_✔
MP20.	IUT never starts cycle unless GNT# is asserted. (3.4.1)	yes_✔ no___
MP21.	IUT always tri-states C/BE# and AD within one clock after GNT# negation when bus is idle and FRAME# is negated. (3.4.3)	yes_✔ no___
MP22.	IUT always drives C/BE# and AD within eight clocks of GNT# assertion when bus is idle. (3.4.3)	yes_✔ no___
MP23.	IUT always asserts IRDY# within eight clocks on all data phases. (3.5.2) **The initial latency is controlled by logic in the user application.**	yes___ no___ n/a_✔
MP24.	IUT always begins locked operations with a read transaction. (3.6) **LOCK# function not supported.**	yes___ no___ n/a_✔
MP25.	IUT always releases LOCK# when access is terminated by target-abort or master-abort. (3.6) **LOCK# function not supported.**	yes___ no___ n/a_✔

Table 13-19: **General Component Protocol Checklist (PCI Master) Checklist *(Continued)***

Item	Description	Pass
MP26.	IUT always deasserts LOCK# for minimum of one idle cycle between consecutive lock operations. (3.6) **LOCK# function not supported.**	yes___ no___ n/a_ ✔
MP27.	IUT always uses Linear Burst Ordering for configuration cycles. (3.7.4)	yes_ ✔ no___
MP28.	IUT always drives PAR within one clock of C/BE# and AD being driven. (3.8.1)	yes_ ✔ no___
MP29.	IUT always drives PAR such that the number of “1”s on AD[31:0],C/BE[3:0], and PAR equals an even number. (3.8.1)	yes_ ✔ no___
MP30.	IUT always drives PERR# (when enabled) active two clocks after data when data parity error is detected. (3.8.2.1)	yes_ ✔ no___
MP31.	IUT always drives PERR# (when enabled) for a minimum of 1 clock for each data phase that a parity error is detected. (3.8.2.1)	yes_ ✔ no___
MP32.	IUT always holds FRAME# asserted for cycle following the dual address command. (3.10.1)	yes_ ✔ no___
MP33.	IUT never generates dual address command when upper 32-bits of address are zero. (3.10.1)	yes ✔ no___
MP34.	IUT deasserts REQ# for at least two clocks after having a request terminated with either Retry or Disconnect (3.3.3.2.2)	yes_ ✔ no___

General Component Protocol Checklist (PCI Target)

The following checklist is to filled out as a general verification of the protocol compliance of the IUT. This checklist applies to all target operations.

Table 13-20: **General Component Protocol Checklist (PCI Target) Checklist**

Item	Description	Pass
TP1.	All sustained tri-state signals are driven high for one clock before being tri-stated. (2.1)	yes_ ✔ no___
TP2.	IUT never reports PERR# until it has claimed the cycle and completed a data phase. (2.2.5)	yes_ ✔ no___
TP3.	IUT never aliases reserved commands with other commands. (3.1.1)	yes_ ✔ no___
TP4.	32-bit addressable IUT treats dual address command as reserved. (3.1.1)	yes_ ✔ no___
TP5.	Once IUT has asserted TRDY# it never changes TRDY# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP6.	Once IUT has asserted TRDY# it never changes DEVSEL# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP7.	Once IUT has asserted TRDY# it never changes STOP# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP8.	Once IUT has asserted STOP# it never changes STOP# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP9.	Once IUT has asserted STOP# it never changes TRDY# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP10.	Once IUT has asserted STOP# it never changes DEVSEL# until the data phase completes. (3.2.1)	yes_ ✔ no___
TP11.	IUT only transfers data when both IRDY# and TRDY# are asserted on the same rising clock edge. (3.2.1)	yes_ ✔ no___
TP12.	IUT always asserts TRDY# when data is valid on a read cycle. (3.2.1)	yes_ ✔ no___

Table 13-20: **General Component Protocol Checklist (PCI Target) Checklist *(Continued)***

Item	Description	Pass
TP13.	IUT always signals target abort when unable to complete the entire IO access as defined by the byte enables. (3.2.2) **This function is implemented in the user application.**	yes__ no___ n/a_ ✔
TP14.	IUT never responds to reserved encodings. (3.2.2)	yes_ ✔ no___
TP15.	IUT always ignores configuration command unless IDSEL is asserted and AD[1:0] are “00”. (3.2.2)	yes_ ✔ no___
TP16.	IUT always disconnects after the first data phase when reserved burst mode is detected. (3.2.2)	yes_ ✔ no___
TP17.	The IUT's AD lines are driven to stable values during every address and data phase. (3.2.4)	yes_ ✔ no___
TP18.	The IUT's C/BE# output buffers remain enabled from the first clock of the data phase through the end of the transaction. (3.3.1)	yes_ ✔ no___
TP19.	IUT never asserts TRDY# during turnaround cycle on a read. (3.3.1)	yes_ ✔ no___
TP20.	IUT always deasserts TRDY#,STOP#, and DEVSEL# the clock following the completion of the last data phase. (3.3.3.2)	yes_ ✔ no___
TP21.	IUT always signals disconnect when burst crosses resource boundary. (3.3.3.2) **This function is implemented in the user application.**	yes__ no___ n/a_ ✔
TP22.	IUT always deasserts STOP# the cycle immediately following FRAME# being deasserted. (3.3.3.2.1)	yes_ ✔ no___
TP23.	Once the IUT has asserted STOP# it never deasserts STOP# until FRAME# is negated. (3.3.3.2.1)	yes_ ✔ no___
TP24.	IUT always deasserts TRDY# before signaling target-abort. (3.3.3.2.1)	yes_ ✔ no___
TP25.	IUT never deasserts STOP# and continues the transaction. (3.3.3.2.1)	yes_ ✔ no___
TP26.	IUT always completes initial data phase within 16 clocks. (3.5.1.1) **The initial latency is controlled by logic in the user application.**	yes__ no___ n/a_ ✔
TP27.	IUT always locks minimum of 16 bytes. (3.6) **LOCK# function not supported.**	yes__ no___ n/a_ ✔
TP28.	IUT always issues DEVSEL# before any other response. (3.7.1)	yes_ ✔ no___
TP29.	Once IUT has asserted DEVSEL# it never deasserts DEVSEL# until the last data phase has competed except to signal target-abort. (3.7.1)	yes_ ✔ no___
TP30.	IUT never responds to special cycles. (3.7.2)	yes_ ✔ no___
TP31.	IUT always drives PAR within one clock of C/BE# and AD being driven. (3.8.1)	yes_ ✔ no___
TP32.	IUT always drives PAR such that the number of “1”s on AD[31:0],C/BE#[3:0], and PAR equals an even number. (3.8.1)	yes_ ✔ no___
TP33.	If IUT is accessed during initialization time (time from RST# is deasserted and 2**25 clocks later), IUT responds to the access by (3.5.1.1): Completing the initial data phase within 16 clocks Ignoring the access Claim the access and hold in wait states Claim the access and terminate with Retry **This function is implemented in the user application. One or more of the above options may be implemented.**	yes_ ✔ no___

Table 13-20: **General Component Protocol Checklist (PCI Target) Checklist *(Continued)***

Item	Description	Pass
TP34.	After terminating a memory write transaction with retry, IUT will be ready to complete at least one data phase of a memory write within 334 clocks for 33 MHz devices and 668 clocks for 66 MHz devices. **This function is implemented in the user application.**	yes___ no___ n/a_ ✔

Component Protocol Checklist for a PCI Master Device

Test Scenario: 1.1. PCI Device Speed (as indicated by DEVSEL#) Tests

If IUT does not implement memory transactions mark 1 through 10 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 10 N/A.

Table 13-21: **PCI Device Speed Tests — Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	
9	Master abort bit set after write to slower than subtractive memory slave.	✔	
10	Master abort bit set after read from slower than subtractive memory slave.	✔	

If IUT does not implement I/O transactions mark 11 through 20 N/A. Else if IUT supports both read and write transactions DO NOT mark 11 through 20 N/A.

Table 13-22: **PCI Device Speed Tests — Checklist #2**

Test	Description	Pass	N/A
11	Data transfer after write to fast I/O slave.	✔	
12	Data transfer after read from fast I/O slave.	✔	
13	Data transfer after write to medium I/O slave.	✔	
14	Data transfer after read from medium I/O slave.	✔	
15	Data transfer after write to slow I/O slave.	✔	
16	Data transfer after read from slow I/O slave.	✔	
17	Data transfer after write to subtractive I/O slave.	✔	
18	Data transfer after read from subtractive I/O slave.	✔	
19	Master abort bit set after write to slower than subtractive I/O slave.	✔	
20	Master abort bit set after read from slower than subtractive I/O slave.	✔	

If IUT does not implement Configuration transactions mark 21 through 30 N/A. Else if IUT supports both read and write transactions DO NOT mark 21 through 30 N/A.

Table 13-23: **PCI Device Speed Tests — Checklist #3**

Test	Description	Pass	N/A
21	Data transfer after write to fast config slave.	✔	
22	Data transfer after read from fast config slave.	✔	

Table 13-23: **PCI Device Speed Tests — Checklist #3 *(Continued)***

Test	Description	Pass	N/A
23	Data transfer after write to medium config slave.	✔	
24	Data transfer after read from medium config slave.	✔	
25	Data transfer after write to slow config slave.	✔	
26	Data transfer after read from slow config slave.	✔	
27	Data transfer after write to subtractive config slave.	✔	
28	Data transfer after read from subtractive config slave.	✔	
29	Master abort bit set after write to slower than subtractive config slave.	✔	
30	Master abort bit set after read from slower than subtractive config slave.	✔	

If IUT does not implement Interrupt transactions mark 31 through 35 N/A.

Table 13-24: **PCI Device Speed Tests — Checklist #4**

Test	Description	Pass	N/A
31	Data transfer after interrupt from fast memory slave.	✔	
32	Data transfer after interrupt from medium memory slave.	✔	
33	Data transfer after interrupt from slow memory slave.	✔	
34	Data transfer after interrupt from subtractive memory slave.	✔	
35	Master abort bit set for interrupt from slower than subtractive memory slave.	✔	

If IUT does not implement Special transactions mark 36 and 37 N/A.

Table 13-25: **PCI Device Speed Tests — Checklist #5**

Test	Description	Pass	N/A
36	Data transfer after Special transaction to slave.	✔	
37	Master abort bit is not set after Special transaction.	✔	

Table 13-26: **PCI Device Speed Tests Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.1 Checklist

Test Scenario: 1.2. PCI Bus Target Abort Cycles

If IUT does not implement memory transactions mark 1 through 16 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 16 N/A.

Table 13-27: **PCI Bus Target Abort Cycles —Checklist #1**

Test	Description	Pass	N/A
1	Target Abort bit set after write to fast memory slave.	✔	
2	IUT does not repeat the write transaction.	✔	
3	IUT's Target Abort bit set after read from fast memory slave.	✔	

Table 13-27: **PCI Bus Target Abort Cycles —Checklist #1 *(Continued)***

Test	Description	Pass	N/A
4	IUT does not repeat the read transaction.	✔	
5	Target Abort bit set after write to medium memory slave.	✔	
6	IUT does not repeat the write transaction.	✔	
7	IUT's Target Abort bit set after read from medium memory slave.	✔	
8	IUT does not repeat the read transaction.	✔	
9	Target Abort bit set after write to slow memory slave.	✔	
10	IUT does not repeat the write transaction.	✔	
11	IUT's Target Abort bit set after read from slow memory slave.	✔	
12	IUT does not repeat the read transaction.	✔	
13	Target Abort bit set after write to subtractive memory slave.	✔	
14	IUT does not repeat the write transaction.	✔	
15	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
16	IUT does not repeat the read transaction.	✔	

If IUT does not implement I/O transactions mark 17 through 32 N/A. Else if IUT supports both read and write transactions DO NOT mark 17 through 32 N/A.

Table 13-28: **PCI Bus Target Abort Cycles —Checklist #2**

Test	Description	Pass	N/A
17	Target Abort bit set after write to fast I/O slave.	✔	
18	IUT does not repeat the write transaction.	✔	
19	IUT's Target Abort bit set after read from fast I/O slave.	✔	
20	IUT does not repeat the read transaction.	✔	
21	Target Abort bit set after write to medium I/O slave.	✔	
22	IUT does not repeat the write transaction.	✔	
23	IUT's Target Abort bit set after read from medium I/O slave.	✔	
24	IUT does not repeat the read transaction.	✔	
25	Target Abort bit set after write to slow I/O slave.	✔	
26	IUT does not repeat the write transaction.	✔	
27	IUT's Target Abort bit set after read from slow I/O slave.	✔	
28	IUT does not repeat the read transaction.	✔	
29	Target Abort bit set after write to subtractive I/O slave.	✔	
30	IUT does not repeat the write transaction.	✔	
31	IUT's Target Abort bit set after read from subtractive I/O slave.	✔	
32	IUT does not repeat the read transaction.	✔	

If IUT does not implement configuration transactions mark 33 through 48 N/A. Else if IUT supports both read and write transactions DO NOT mark 33 through 48 N/A.

Table 13-29: **PCI Bus Target Abort Cycles —Checklist #3**

Test	Description	Pass	N/A
33	Target Abort bit set after write to fast config slave	✔	
34	IUT does not repeat the write transaction.	✔	
35	IUT's Target Abort bit set after read from fast config slave.	✔	
36	IUT does not repeat the read transaction.	✔	
37	Target Abort bit set after write to medium config slave.	✔	

Table 13-29: **PCI Bus Target Abort Cycles —Checklist #3 *(Continued)***

Test	Description	Pass	N/A
38	IUT does not repeat the write transaction.	✔	
39	IUT's Target Abort bit set after read from medium config slave.	✔	
40	IUT does not repeat the read transaction.	✔	
41	Target Abort bit set after write to slow config slave.	✔	
42	IUT does not repeat the write transaction.	✔	
43	IUT's Target Abort bit set after read from slow config slave.	✔	
44	IUT does not repeat the read transaction.	✔	
45	Target Abort bit set after write to subtractive config slave.	✔	
46	IUT does not repeat the write transaction.	✔	
47	IUT's Target Abort bit set after read from subtractive config slave.	✔	
48	IUT does not repeat the read transaction.	✔	

If IUT does not implement interrupt transactions mark 49 through 56 N/A.

Table 13-30: **PCI Bus Target Abort Cycles —Checklist #4**

Test	Description	Pass	N/A
49	IUT's Target Abort bit set after interrupt acknowledge from fast slave.	✔	
50	IUT does not repeat the interrupt acknowledge transaction.	✔	
51	IUT's Target Abort bit set after interrupt acknowledge from medium slave.	✔	
52	IUT does not repeat the interrupt acknowledge transaction.	✔	
53	IUT's Target Abort bit set after interrupt acknowledge from slow slave.	✔	
54	IUT does not repeat the interrupt acknowledge transaction.	✔	
55	IUT's Target Abort bit set after interrupt acknowledge from subtractive slave.	✔	
56	IUT does not repeat the interrupt acknowledge transaction.	✔	

Table 13-31: **PCI Bus Target Abort Cycles Explanations**

Explanations
The user application is responsible for not repeating terminated transactions.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.2 Checklist

Test Scenario: 1.3. PCI Bus Target Retry Cycles

If IUT does not implement memory transactions mark 1 through 8 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 8 N/A.

Table 13-32: **PCI Bus Target Retry Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	

Table 13-32: **PCI Bus Target Retry Cycles — Checklist #1 *(Continued)***

Test	Description	Pass	N/A
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions mark 9 through 16 N/A. Else if IUT supports both read and write transactions DO NOT mark 9 through 16 N/A.

Table 13-33: **PCI Bus Target Retry Cycles — Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions mark 17 through 24 N/A. Else if IUT supports both read and write transactions DO NOT mark 17 through 24 N/A.

Table 13-34: **PCI Bus Target Retry Cycles — Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config slave.	✔	
18	Data transfer after read from fast config slave.	✔	
19	Data transfer after write to medium config slave.	✔	
20	Data transfer after read from medium config slave.	✔	
21	Data transfer after write to slow config slave.	✔	
22	Data transfer after read from slow config slave.	✔	
23	Data transfer after write to subtractive config slave.	✔	
24	Data transfer after read from subtractive config slave.	✔	

If IUT does not implement interrupt transactions mark 25 through 28 N/A else do not mark 25 through 28 N/A.

Table 13-35: **PCI Bus Target Retry Cycles — Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after interrupt acknowledge from fast slave.	✔	
26	Data transfer after interrupt acknowledge from medium slave.	✔	
27	Data transfer after interrupt acknowledge from slow slave.	✔	
28	Data transfer after interrupt acknowledge from subtractive slave.	✔	

Table 13-36: **PCI Bus Target Retry Cycles Explanations**

Explanations

Table 13-36: **PCI Bus Target Retry Cycles Explanations *(Continued)***

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.3 Checklist

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

If IUT does not implement memory transactions mark 1 through 8 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 8 N/A.

Table 13-37: **PCI Bus Single Data Phase Disconnect Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions mark 9 through 16 N/A. Else if IUT supports both read and write transactions DO NOT mark 9 through 16 N/A.

Table 13-38: **PCI Bus Single Data Phase Disconnect Cycles — Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions mark 17 through 24 N/A. Else if IUT supports both read and write transactions DO NOT mark 17 through 24 N/A.

Table 13-39: **PCI Bus Single Data Phase Disconnect Cycles — Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config slave.	✔	
18	Data transfer after read from fast config slave.	✔	
19	Data transfer after write to medium config slave.	✔	
20	Data transfer after read from medium config slave.	✔	
21	Data transfer after write to slow config slave.	✔	

Table 13-39: **PCI Bus Single Data Phase Disconnect Cycles — Checklist #3 *(Continued)***

Test	Description	Pass	N/A
22	Data transfer after read from slow config slave.	✔	
23	Data transfer after write to subtractive config slave.	✔	
24	Data transfer after read from subtractive config slave.	✔	

If IUT does not implement interrupt transactions mark 25 through 28 N/A else do not mark 25 through 28 N/A.

Table 13-40: **PCI Bus Single Data Phase Disconnect Cycles — Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after interrupt acknowledge from fast slave.	✔	
26	Data transfer after interrupt acknowledge from medium slave.	✔	
27	Data transfer after interrupt acknowledge from slow slave.	✔	
28	Data transfer after interrupt acknowledge from subtractive slave.	✔	

Table 13-41: **PCI Bus Single Data Phase Disconnect Cycles Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.4 Checklist

Test Scenario: 1.5. PCI Bus Multi-Data Phase Target Abort Cycles

If IUT does not implement memory transactions mark 1 through 16 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 16 N/A.

Table 13-42: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Target Abort bit set after write to fast memory slave.	✔	
2	IUT does not repeat the write transaction.	✔	
3	IUT's Target Abort bit set after read from fast memory slave.	✔	
4	IUT does not repeat the read transaction.	✔	
5	Target Abort bit set after write to medium memory slave.	✔	
6	IUT does not repeat the write transaction.	✔	
7	IUT's Target Abort bit set after read from medium memory slave.	✔	
8	IUT does not repeat the read transaction.	✔	
9	Target Abort bit set after write to slow memory slave.	✔	
10	IUT does not repeat the write transaction.	✔	
11	IUT's Target Abort bit set after read from slow memory slave.	✔	
12	IUT does not repeat the read transaction.	✔	
13	Target Abort bit set after write to subtractive memory slave.	✔	
14	IUT does not repeat the write transaction.	✔	
15	IUT's Target Abort bit set after read from subtractive memory slave.	✔	

Table 13-42: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #1 *(Continued)***

Test	Description	Pass	N/A
16	IUT does not repeat the read transaction.	✔	

If IUT does not implement dual address transactions mark 17 through 32 N/A. Else if IUT supports both read and write dual address transactions DO NOT mark 17 through 32 N/A.

Table 13-43: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #2**

Test	Description	Pass	N/A
17	Target Abort bit set after write to fast memory slave.	✔	
18	IUT does not repeat the write transaction.	✔	
19	IUT's Target Abort bit set after read from fast memory slave.	✔	
20	IUT does not repeat the read transaction.	✔	
21	Target Abort bit set after write to medium memory slave.	✔	
22	IUT does not repeat the write transaction.	✔	
23	IUT's Target Abort bit set after read from medium memory slave.	✔	
24	IUT does not repeat the read transaction.	✔	
25	Target Abort bit set after write to slow memory slave.	✔	
26	IUT does not repeat the write transaction.	✔	
27	IUT's Target Abort bit set after read from slow memory slave.	✔	
28	IUT does not repeat the read transaction.	✔	
29	Target Abort bit set after write to subtractive memory slave.	✔	
30	IUT does not repeat the write transaction.	✔	
31	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
32	IUT does not repeat the read transaction.	✔	

If IUT does not implement configuration transactions mark 33 through 48 N/A. Else if IUT supports both read and write transactions DO NOT mark 33 through 48 N/A.

Table 13-44: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #3**

Test	Description	Pass	N/A
33	Target Abort bit set after write to fast config. slave.	✔	
34	IUT does not repeat the write transaction.	✔	
35	IUT's Target Abort bit set after read from fast config. slave.	✔	
36	IUT does not repeat the read transaction.	✔	
37	Target Abort bit set after write to medium config. slave.	✔	
38	IUT does not repeat the write transaction.	✔	
39	IUT's Target Abort bit set after read from medium config. slave.	✔	
40	IUT does not repeat the read transaction.	✔	
41	Target Abort bit set after write to slow config. slave.	✔	
42	IUT does not repeat the write transaction.	✔	
43	IUT's Target Abort bit set after read from slow config. slave.	✔	
44	IUT does not repeat the read transaction.	✔	
45	Target Abort bit set after write to subtractive config. slave.	✔	
46	IUT does not repeat the write transaction.	✔	
47	IUT's Target Abort bit set after read from subtractive config. slave.	✔	
48	IUT does not repeat the read transaction.	✔	

If IUT does not implement memory read multiple transactions mark 49 through 56 N/A. Else DO NOT mark 49 through 56 N/A.

Table 13-45: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #4**

Test	Description	Pass	N/A
49	IUT's Target Abort bit set after read from fast memory slave.	✔	
50	IUT does not repeat the read transaction.	✔	
51	IUT's Target Abort bit set after read from medium memory slave.	✔	
52	IUT does not repeat the read transaction.	✔	
53	IUT's Target Abort bit set after read from slow memory slave.	✔	
54	IUT does not repeat the read transaction.	✔	
55	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
56	IUT does not repeat the read transaction.	✔	

If IUT does not implement memory read line transactions mark 57 through 64 N/A. Else DO NOT mark 57 through 64 N/A.

Table 13-46: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #5**

Test	Description	Pass	N/A
57	IUT's Target Abort bit set after read from fast memory slave.	✔	
58	IUT does not repeat the read transaction.	✔	
59	IUT's Target Abort bit set after read from medium memory slave.	✔	
60	IUT does not repeat the read transaction.	✔	
61	IUT's Target Abort bit set after read from slow memory slave.	✔	
62	IUT does not repeat the read transaction.	✔	
63	IUT's Target Abort bit set after read from subtractive memory slave.	✔	
64	IUT does not repeat the read transaction.	✔	

If IUT does not implement memory write and invalidate transactions mark 65 through 72 N/A. Else DO NOT mark 65 through 72 N/A.

Table 13-47: **PCI Bus Multi-Data Phase Target Abort Cycles — Checklist #6**

Test	Description	Pass	N/A
65	Target Abort bit set after write to fast memory slave.	✔	
66	IUT does not repeat the write transaction.	✔	
67	Target Abort bit set after write to medium memory slave.	✔	
68	IUT does not repeat the write transaction.	✔	
69	Target Abort bit set after write to slow memory slave.	✔	
70	IUT does not repeat the write transaction.	✔	
71	IUT's Target Abort bit set after read from slow memory slave.	✔	
72	IUT does not repeat the write transaction.	✔	

Table 13-48: **PCI Bus Multi-Data Phase Target Abort Cycles Explanations**

Explanations
The user application is responsible for not repeating terminated transactions.

Table 13-48: **PCI Bus Multi-Data Phase Target Abort Cycles Explanations *(Continued)***

Explanations
This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.5 Checklist

Test Scenario: 1.6. PCI Bus Multi-Data Phase Retry Cycles

If IUT does not implement memory transactions mark 1 through 8 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 8 N/A.

Table 13-49: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions mark 9 through 16 N/A. Else if IUT supports both read and write transactions DO NOT mark 9 through 16 N/A.

Table 13-50: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions mark 17 through 24 N/A. Else if IUT supports both read and write transactions DO NOT mark 17 through 24 N/A.

Table 13-51: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config. slave.	✔	
18	Data transfer after read from fast config. slave.	✔	
19	Data transfer after write to medium config. slave.	✔	
20	Data transfer after read from medium config. slave.	✔	
21	Data transfer after write to slow config. slave.	✔	
22	Data transfer after read from slow config. slave.	✔	
23	Data transfer after write to subtractive config. slave.	✔	

Table 13-51: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #3 *(Continued)***

Test	Description	Pass	N/A
24	Data transfer after read from subtractive config. slave.	✔	

If IUT does not implement memory read multiple transactions mark 25 through 28 N/A else do not mark 25 through 28 N/A.

Table 13-52: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after memory read multiple from fast slave.	✔	
26	Data transfer after memory read multiple from medium slave.	✔	
27	Data transfer after memory read multiple from slow slave.	✔	
28	Data transfer after memory read multiple from subtractive slave.	✔	

If IUT does not implement memory read line transactions mark 29 through 32 N/A. Else do not mark 29 through 32 N/A.

Table 13-53: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #5**

Test	Description	Pass	N/A
29	Data transfer after memory read line from fast slave.	✔	
30	Data transfer after memory read line from medium slave.	✔	
31	Data transfer after memory read line from slow slave.	✔	
32	Data transfer after memory read line from subtractive slave.	✔	

If IUT does not implement memory write and invalidate transactions mark 33 through 36 N/A. Else do not mark 33 through 36 N/A.

Table 13-54: **PCI Bus Multi-Data Phase Retry Cycles — Checklist #6**

Test	Description	Pass	N/A
33	Data transfer after memory write and invalidate to fast slave.	✔	
34	Data transfer after memory write and invalidate to medium slave.	✔	
35	Data transfer after memory write and invalidate to slow slave.	✔	
36	Data transfer after memory write and invalidate to subtractive slave.	✔	

Table 13-55: **PCI Bus Multi-Data Phase Retry Cycles Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.6 Checklist

Test Scenario: 1.7. PCI Bus Multi-Data Phase Disconnect Cycles

If IUT does not implement multi-data phase memory transactions mark 1 through 8 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 8 N/A.

Table 13-56: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Data transfer after write to fast memory slave.	✔	
2	Data transfer after read from fast memory slave.	✔	
3	Data transfer after write to medium memory slave.	✔	
4	Data transfer after read from medium memory slave.	✔	
5	Data transfer after write to slow memory slave.	✔	
6	Data transfer after read from slow memory slave.	✔	
7	Data transfer after write to subtractive memory slave.	✔	
8	Data transfer after read from subtractive memory slave.	✔	

If IUT does not implement I/O transactions mark 9 through 16 N/A. Else if IUT supports both read and write transactions DO NOT mark 9 through 16 N/A.

Table 13-57: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #2**

Test	Description	Pass	N/A
9	Data transfer after write to fast I/O slave.	✔	
10	Data transfer after read from fast I/O slave.	✔	
11	Data transfer after write to medium I/O slave.	✔	
12	Data transfer after read from medium I/O slave.	✔	
13	Data transfer after write to slow I/O slave.	✔	
14	Data transfer after read from slow I/O slave.	✔	
15	Data transfer after write to subtractive I/O slave.	✔	
16	Data transfer after read from subtractive I/O slave.	✔	

If IUT does not implement configuration transactions mark 17 through 24 N/A. Else if IUT supports both read and write transactions DO NOT mark 17 through 24 N/A.

Table 13-58: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #3**

Test	Description	Pass	N/A
17	Data transfer after write to fast config. slave.	✔	
18	Data transfer after read from fast config. slave.	✔	
19	Data transfer after write to medium config. slave.	✔	
20	Data transfer after read from medium config. slave.	✔	
21	Data transfer after write to slow config. slave.	✔	
22	Data transfer after read from slow config. slave.	✔	
23	Data transfer after write to subtractive config. slave.	✔	
24	Data transfer after read from subtractive config. slave.	✔	

If IUT does not implement memory read multiple transactions mark 25 through 28 N/A else do not mark 25 through 28 N/A.

Table 13-59: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #4**

Test	Description	Pass	N/A
25	Data transfer after memory read multiple from fast slave.	✔	
26	Data transfer after memory read multiple from medium slave.	✔	

Table 13-59: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #4 *(Continued)***

Test	Description	Pass	N/A
27	Data transfer after memory read multiple from slow slave.	✔	
28	Data transfer after memory read multiple from subtractive slave.	✔	

If IUT does not implement memory read line transactions mark 29 through 32 N/A else do not mark 29 through 32 N/A.

Table 13-60: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #5**

Test	Description	Pass	N/A
29	Data transfer after memory read line from fast slave.	✔	
30	Data transfer after memory read line from medium slave.	✔	
31	Data transfer after memory read line from slow slave.	✔	
32	Data transfer after memory read line from subtractive slave.	✔	

If IUT does not implement memory write and invalidate transactions mark 33 through 36 N/A else do not mark 33 through 36 N/A.

Table 13-61: **PCI Bus Multi-Data Phase Disconnect Cycles — Checklist #6**

Test	Description	Pass	N/A
33	Data transfer after memory write and invalidate to fast slave.	✔	
34	Data transfer after memory write and invalidate to medium slave.	✔	
35	Data transfer after memory write and invalidate to slow slave.	✔	
36	Data transfer after memory write and invalidate to subtractive slave.	✔	

Table 13-62: **PCI Bus Multi-Data Phase Disconnect Cycles Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.7 Checklist

Test Scenario: 1.8. Multi-Data Phase & TRDY# Cycles

If IUT does not implement multi-data phase memory transactions mark 1 through 12 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 12 N/A.

Table 13-63: **Multi-Data Phase & TRDY# Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Verify that data is written to primary target when TRDY# is released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
2	Verify that data is read from primary target when TRDY# is released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
3	Verify that data is written to primary target when TRDY# is released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	

Table 13-63: **Multi-Data Phase & TRDY# Cycles — Checklist #1 *(Continued)***

Test	Description	Pass	N/A
4	Verify that data is read from primary target when TRDY# is released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
5	Verify that data is written to primary target when TRDY# is released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
6	Verify that data is read from primary target when TRDY# is released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
7	Verify that data is written to primary target when TRDY# is released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
8	Verify that data is read from primary target when TRDY# is released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
9	Verify that data is written to primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
10	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
11	Verify that data is written to primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	
12	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement dual address transactions mark 13 through 24 N/A. Else if IUT supports both read and write transactions DO NOT mark 13 through 24 N/A.

Table 13-64: **Multi-Data Phase & TRDY# Cycles — Checklist #2**

Test	Description	Pass	N/A
13	Verify that data is written to primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
14	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
15	Verify that data is written to primary target when TRDY# released after 4th rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
16	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
17	Verify that data is written to primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
18	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
19	Verify that data is written to primary target when TRDY# released after 5th rising clock edge and asserted on 7th rising clock edge after FRAME#	✔	
20	Verify that data is read from primary target when TRDY# released after 5th rising clock edge and asserted on 7th rising clock edge after FRAME#	✔	
21	Verify that data is written to primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
22	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
23	Verify that data is written to primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	
24	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement memory read multiple transactions mark 25 through 30 N/A else do not mark 25 through 30 N/A.

Table 13-65: **Multi-Data Phase & TRDY# Cycles — Checklist #3**

Test	Description	Pass	N/A
25	Verify that data is read from primary target when TRDY# released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
26	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
27	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
28	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
29	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
30	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement memory read line transactions mark 31 through 36 N/A else do not mark 31 through 36 N/A.

Table 13-66: **Multi-Data Phase & TRDY# Cycles — Checklist #4**

Test	Description	Pass	N/A
31	Verify that data is read from primary target when TRDY# released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
32	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
33	Verify that data is read from primary target when TRDY# released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
34	Verify that data is read from primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
35	Verify that data is read from primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
36	Verify that data is read from primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

If IUT does not implement memory write and invalidate transactions mark 37 through 42 N/A else do not mark 37 through 42 N/A.

Table 13-67: **Multi-Data Phase & TRDY# Cycles — Checklist #5**

Test	Description	Pass	N/A
37	Verify that data is written to primary target when TRDY# released after 2nd rising clock edge and asserted on 3rd rising clock edge after FRAME#	✔	
38	Verify that data is written to primary target when TRDY# released after 3rd rising clock edge and asserted on 4th rising clock edge after FRAME#	✔	
39	Verify that data is written to primary target when TRDY# released after 3rd rising clock edge and asserted on 5th rising clock edge after FRAME#	✔	
40	Verify that data is written to primary target when TRDY# released after 4th rising clock edge and asserted on 6th rising clock edge after FRAME#	✔	
41	Verify that data is written to primary target when TRDY# alternately released for one clock cycle and asserted for one clock cycle after FRAME#	✔	
42	Verify that data is written to primary target when TRDY# alternately released for two clock cycles and asserted for two clock cycles after FRAME#	✔	

Table 13-68: **Multi-Data Phase & TRDY# Cycles Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.8 Checklist

Test Scenario: 1.9. Bus Data Parity Error Single Cycles

If IUT is exempted from reporting parity errors per the exemptions listed in subsection 3.7.2 of the specification then mark the following N/A. If IUT does not implement memory transactions mark 1 through 3 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 3 N/A

Table 13-69: **Bus Data Parity Error Single Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Verify the IUT sets Data Parity Error Detected bit when Primary Target asserts PERR# on IUT Memory Write	✔	
2	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT Memory Read	✔	
3	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Memory read	✔	

If IUT does not implement I/O transactions mark 4 through 6 N/A. Else if IUT supports both read and write transactions DO NOT mark 4 through 6 N/A.

Table 13-70: **Bus Data Parity Error Single Cycles — Checklist #2**

Test	Description	Pass	N/A
4	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT I/O Write	✔	
5	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT I/O Read	✔	
6	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT I/O read	✔	

If IUT does not implement configuration transactions mark 7 through 9 N/A. Else if IUT supports both read and write transactions DO NOT mark 7 through 9 N/A.

Table 13-71: **Bus Data Parity Error Single Cycles — Checklist #3**

Test	Description	Pass	N/A
7	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT Config Write	✔	
8	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT Config Read	✔	
9	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Config read	✔	

Table 13-72: **Bus Data Parity Error Single Cycles Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.9 Checklist

Test Scenario: 1.10. Bus Data Parity Error Multi-Data Phase Cycles

If IUT is exempted from reporting parity errors per the exemptions listed in subsection 3.7.2 of the specification then mark the following N/A. If IUT does not implement memory transactions mark 1 through 3 N/A. Else if IUT supports both read and write transactions DO NOT mark 1 through 3 N/A.

Table 13-73: **Bus Data Parity Error Multi-Data Phase Cycles — Checklist #1**

Test	Description	Pass	N/A
1	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT multi data phase Memory Write	✔	
2	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT multi data phase Memory Read	✔	
3	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Memory multi data phase read	✔	

If IUT does not implement dual address transactions mark 4 through 6 N/A. Else if IUT supports both read and write transactions DO NOT mark 4 through 6 N/A.

Table 13-74: **Bus Data Parity Error Multi-Data Phase Cycles — Checklist #2**

Test	Description	✔	N/A
4	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT dual address multi data phase Write	✔	
5	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT dual address multi data phase Read	✔	
6	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT dual address multi data phase read	✔	

If IUT does not implement configuration transactions mark 7 through 9 N/A. Else if IUT supports both read and write transactions DO NOT mark 7 through 9 N/A.

Table 13-75: **Bus Data Parity Error Multi-Data Phase Cycles — Checklist #3**

Test	Description	Pass	N/A
7	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT Config multi-data phase Write	✔	
8	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT Config multi-data phase Read	✔	
9	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT Config multi-data phase read	✔	

If IUT does not implement memory read multiple transactions mark 10 through 11 N/A. Else DO NOT mark 10 through 11 N/A.

Table 13-76: **Bus Data Parity Error Multi-Data Phase Cycles — Checklist #4**

Test	Description	Pass	N/A
10	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT mem. rd. multiple data phase.	✔	
11	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT mem. rd. multiple data phase.	✔	

If IUT does not implement memory read line transactions mark 12 through 13 N/A. Else DO NOT mark 12 through 13 N/A.

Table 13-77: **Bus Data Parity Error Multi-Data Phase Cycles — Checklist #5**

Test	Description	Pass	N/A
12	Verify that PERR# is active two clocks after the first data phase (which had odd parity) on IUT mem. rd. line data phase.	✔	
13	Verify the IUT sets Parity Error Detected bit when odd parity is detected on IUT mem. rd. line data phase.	✔	

If IUT does not implement memory write and invalidate transactions mark 14 N/A. Else DO NOT mark 14 N/A.

Table 13-78: **Bus Data Parity Error Multi-Data Phase Cycles — Checklist #6**

Test	Description	Pass	N/A
14	Verify the IUT sets Parity Error Detected bit when Primary Target asserts PERR# on IUT Memory Write and Invalidate data phase.	✔	

Table 13-79: **Bus Data Parity Error Multi-Data Phase Cycles Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.10 Checklist

Test Scenario: 1.11. Bus Master Timeout

If IUT does not support dual address cycles, mark 7 and 8 N/A

Table 13-80: **Bus Master Timeout — Checklist #1**

Test	Description	Pass	N/A
1	Memory write transaction terminates before 4 data phases completed	✔	
2	Memory read transaction terminates before 4 data phases completed	✔	
3	Config write transaction terminates before 4 data phases completed	✔	
4	Config read transaction terminates before 4 data phases completed	✔	
5	Memory read multiple transaction terminates before 4 data phases	✔	
6	Memory read line transaction terminates before 4 data phases	✔	
7	Dual Address write transaction terminates before 4 data phases completed	✔	

Table 13-80: **Bus Master Timeout — Checklist #1 *(Continued)***

Test	Description	Pass	N/A
8	Dual Address read transaction terminates before 4 data phases completed	✔	

If IUT does not support cache coherent transactions mark 9 N/A Else if IUT supports cache coherent transactions and has implemented the configuration register that specifies cache line length DO NOT mark 9 N/A.

Table 13-81: **Bus Master Timeout — Checklist #2**

Test	Description	Pass	N/A
9	Memory write invalidate terminates on line boundary	✔	

Table 13-82: **Bus Master Timeout Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End Scenario 1.11 Checklist

Test Scenario: 1.12. PCI Bus Master Parking

Verify that the IUT is able to drive PCI bus to stable conditions if it is idle and GNT# is asserted.

Table 13-83: **PCI Bus Master Parking — Checklist #1**

Test	Description	Pass	N/A
1	IUT drives AD[31::00] to stable values within eight PCI Clocks of GNT#.	✔	
2	IUT drives C/BE#[3:0] to stable values within eight PCI Clocks of GNT#.	✔	
3	IUT drives PAR one clock cycle after IUT drives AD[31:0]	✔	

Verify that the IUT will Tri-state the bus when GNT# is not asserted.

Table 13-84: **PCI Bus Master Parking — Checklist #2**

Test	Description	Pass	N/A
4	IUT Tri-states AD[31:00] and C/BE[3:0] and PAR when GNT# is released.	✔	

Table 13-85: **PCI Bus Master Parking Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.12 Checklist

Test Scenario: 1.13. PCI Bus Master Arbitration

Verify that the IUT is able to complete bus transaction when GNT# is deasserted coincident with FRAME# asserted.

Table 13-86: **PCI Bus Master Arbitration — Checklist #1**

Test	Description	Pass	N/A
1	IUT completes transaction when de-asserting GNT# is coincident with asserting FRAME#.	✔	

Table 13-87: **PCI Bus Master Arbitration Explanations**

Explanations

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.

End of Scenario 1.13 Checklist

Component Protocol Checklist for a PCI Target Device

Test Scenario: 2.1. Target Reception of an Interrupt Cycle

If IUT does not respond to Interrupt Acknowledge bus transactions mark 1 through 2 N/A.

Table 13-88: **Target Reception of an Interrupt Cycle Checklist**

Test	Description	Pass	N/A
1	IUT generates Interrupts when programmed	✔	
2	IUT clears Interrupts when serviced (may include driver specific actions)	✔	

Test Scenario: 2.2. Target Reception of Special Cycle

If IUT does not implement Special Cycles mark 1 through 2 N/A.

Table 13-89: **Target Reception of Special Cycle Checklist**

Test	Description	Pass	N/A
1	No DEVSEL# Assertion by IUT after Special Cycle		✔
2	IUT receives encoded special cycle		✔

Test Scenario: 2.3. Target Detection of Address and Data Parity Error for Special Cycle

If IUT does not implement Special Cycles mark 2 N/A.

Table 13-90: **Target Detection of Address and Data Parity Error for Special Cycle Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR#	✔	
2	IUT reports data parity error via SERR#		✔
3	IUT keeps SERR# active for at least one clock	✔	

Test Scenario: 2.4. Target Reception of I/O Cycles With Legal and Illegal Byte Enables

IF IUT does not support I/O cycles mark 1 through 4 N/A or if IUT claims all 32 bits during an I/O cycle mark 1 and 2 N/A

Table 13-91: **Target Reception of I/O Cycles With Legal and Illegal Byte Enables Checklist**

Test	Description	Pass	N/A
1	IUT asserts TRDY# following 2nd rising edge from FRAME on all legal BE		✔
2	IUT terminates with target abort for each illegal BE		✔
3	IUT asserts STOP#	✔	
4	IUT de-asserts STOP# after FRAME# de-assertion	✔	

Test Scenario: 2.5. Target Ignores Reserved Commands

Table 13-92: **Target Ignores Reserved Commands Checklist**

Test	Description	Pass	N/A
1	IUT does not respond to reserved commands	✔	
2	Initiator detects master abort for each transfer	✔	
3	IUT does not respond to 64bit cycle (dual address)	✔	

Test Scenario: 2.6. Target Receives Configuration Cycles

If IUT does not support configuration type 1 mark 3 NA.

Table 13-93: **Target Receives Configuration Cycles Checklist**

Test	Description	Pass	N/A
1	IUT responds to all configuration cycles type 0 read/write cycles appropriately	✔	
2	IUT does not respond to configuration cycles type 0 with IDSEL inactive	✔	
3	IUT responds to all configuration cycles type 1 read/write cycles appropriately		✔

Table 13-93: **Target Receives Configuration Cycles Checklist *(Continued)***

4	IUT responds to all configuration cycles type 0 read/write cycles appropriately	✔	
5	IUT does not respond (master abort) on illegal configuration cycle types	✔	

Test Scenario: 2.7. Target Receives I/O Cycles With Address and Data Parity Errors

If IUT does not support I/O cycles mark 1 through 2 N/A.

Table 13-94: **Target Receives I/O Cycles With Address and Data Parity Errors Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR# during I/O read/write cycles	✔	
2	IUT reports data parity error via PERR# during I/O write cycles	✔	

Test Scenario: 2.8. Target Gets Config Cycles With Address and Data Parity Errors

Table 13-95: **Target Gets Configuration Cycles With Address and Data Parity Errors Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR# during configuration read/write cycles	✔	
2	IUT reports data parity error via PERR# during configuration write cycles	✔	

Test Scenario: 2.9. Target Receives Memory Cycles

If IUT does not interface to a memory subsystem mark all N/A. If IUT does not interface to main system memory or memory is not cacheable mark 2 through 4 N/A.

Table 13-96: **Target Receives Memory Cycles Checklist**

Test	Description	Pass	N/A
1	IUT completes single memory read and write cycles appropriately	✔	
2	IUT completes memory read line cycles appropriately	✔	
3	IUT completes memory read multiple cycles appropriately	✔	
4	IUT completes memory write and invalidate cycles appropriately	✔	
5	IUT completes one cycle and disconnects on reserved memory operations	✔	
6	IUT disconnects on burst transactions that cross its address boundary	✔	

Test Scenario: 2.10. Target Gets Memory Cycles With Address and Data Parity Errors

If IUT does not interface to a memory subsystem mark 1 to 2 N/A.

Table 13-97: **Target Gets Memory Cycles With Address and Data Parity Errors Checklist**

Test	Description	Pass	N/A
1	IUT reports address parity error via SERR# during all memory read and write cycles	✔	
2	IUT reports data parity error via PERR# during all memory write cycles	✔	

Test Scenario: 2.11. Target Gets Fast Back to Back Cycles

If IUT does not implement the "fast back to back" bit then mark 3 and 4 N/A.

Table 13-98: **Target Gets Fast Back to Back Cycles Checklist**

Test	Description	Pass	N/A
1	IUT responds to back to back memory writes appropriately	✔	
2	IUT responds to memory write followed by memory read appropriately	✔	
3	IUT responds to back to back memory writes with 2nd write selecting IUT		✔
4	IUT responds to memory write followed by memory read with read selecting IUT		✔

Table 13-99: **Target Gets Fast Back to Back Cycles Explanations**

Explanations
Test 2.6: The implementation under test does not support configuration bursts as a target.
Test 2.9: The user application is responsible for terminating the transfer.

This section should be used to clarify any answers on checklist items above. Please key explanation to item number.