



PCI Master/Target MegaCore Function with DMA

April 1997, ver. 1

Data Sheet

Features

- pci_a MegaCore™ function implementing a 32-bit peripheral component interconnect (PCI) interface
- Optimized for FLEX® 10K architecture
- Hardware tested
- Dramatically shortens design cycles
- Simulation circuit board included
- Extensive simulation testing
- Includes test vectors
- OpenCore™ feature allows designers to instantiate and simulate designs in MAX+PLUS® II prior to purchase
- Uses approximately 950 FLEX logic elements (LEs), which is about 50% the capacity of an EPF10K30 device
- Fully-compliant design (see [“Compliance Summary” on page 3](#)), including
 - 32-bit, 33-MHz operation
 - Simplified local side (DMA control engine) interface
 - Fully simulated with required protocol test scenarios
 - Hardware tested
- PCI master features:
 - Memory read/write
 - Bus parking
 - Fully integrated DMA engine including address counter register, byte counter register, control & status register, interrupt status register
 - Configurable interrupt source, including DMA terminal count, master abort, target abort, local side interrupt
 - 64-byte (16 double words or DWORDS) RAM buffer implemented in FLEX 10K embedded array blocks (EABs)
 - Zero wait state PCI read (107 Mbytes per second); zero wait state PCI write (102 Mbytes per second)
- PCI target features:
 - Type zero configuration space
 - Parity error detection
 - Memory read/write and configuration read/write
 - Target abort, retry, and disconnect
 - 1 Mbyte of target space
- Configuration registers: device ID, vendor ID, status, command, class code, revision ID, header type, latency timer, one memory base address, subsystem ID, subsystem vendor ID, maximum latency, minimum grant, interrupt pin, and interrupt line

Introduction

This data sheet provides operating information for the pci_a MegaCore function and includes the following topics:

Ordering Information.....	3
General Description.....	3
Compliance Summary.....	3
PCI Bus Signals.....	4
Local Side Signals.....	7
Function Prototype.....	9
Parameters.....	9
Functional Description.....	10
Sustained Tri-State Signal Operation.....	11
Master Device Signals & Signal Assertion.....	12
Target Device Signals & Signal Assertion.....	13
Parity Signal Operation.....	14
Bus Master Commands.....	15
Configuration Registers.....	15
Vendor ID Register (Offset = 00 Hex).....	17
Device ID Register (Offset = 02 Hex).....	18
Command Register (Offset = 04 Hex).....	18
Status Register: (Offset = 06 Hex).....	19
Revision ID Register (Offset = 08 Hex).....	20
Class Code Register (Offset = 09 Hex).....	20
Latency Timer Register (Offset = 0D Hex).....	20
Header Type Register (Offset = 0E Hex).....	21
Base Address Register Zero (Offset = 10 Hex).....	21
Subsystem Vendor ID Register (Offset = 2C Hex).....	22
Subsystem ID Register (Offset = 2E Hex).....	22
Interrupt Line Register (Offset = 3C Hex).....	22
Interrupt Pin Register (Offset = 3D Hex).....	23
Minimum Grant Register (Offset = 3E Hex).....	23
Maximum Latency Register (Offset = 3F Hex).....	23
PCI Bus Transactions.....	24
Target Transactions.....	24
Configuration Transactions.....	30
Master Transactions.....	32
DMA Operation.....	39
Target Address Space.....	39
Internal Target Registers Memory Map.....	39
DMA Registers.....	40
DMA Transactions.....	43
General Programming Guidelines.....	47
Applications.....	50
PCI SIG Protocol Checklists.....	52
PCI SIG Test Bench Summary.....	59
References.....	65

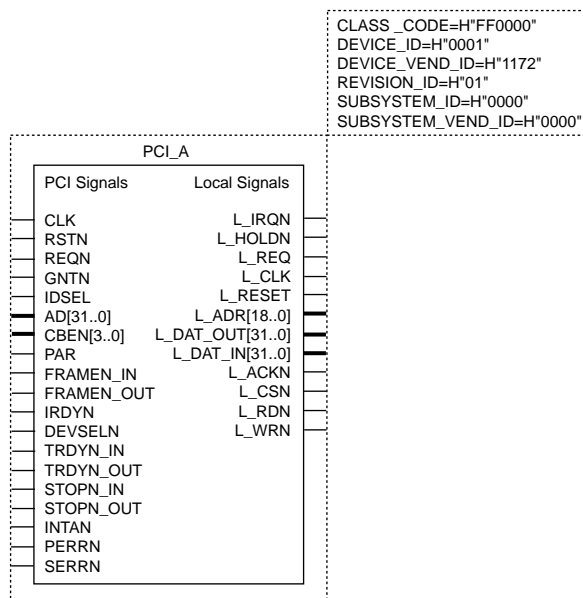
Ordering Information

The Altera® `pci_a` ordering code is PLSM-PCI/A. The maintenance agreement ordering code is PLAESW-MCORE.

General Description

The `pci_a` MegaCore function provides a timely solution for integrating 32-bit PCI peripheral devices, and is fully tested to meet the requirements of the PCI specification. The `pci_a` is optimized for the EPF10K30RC240-3 device, reducing the design task and enabling designers to focus efforts on the custom logic surrounding the PCI interface. The `pci_a` function is intended for use in Altera FLEX 10K devices with remaining logic resources available for user-defined local side (DMA control engine) customization. See [Figure 1](#).

Figure 1. `pci_a` Symbol



Compliance Summary

The `pci_a` function is compliant with the requirements specified in the PCI Special Interest Group’s (SIG) *PCI Local Bus Specification, Rev. 2.1*, and *Compliance Checklist, Rev. 2.1*. The `pci_a` function was verified using the applicable scenarios listed in [Table 1](#). For a detailed listing of tests performed, see “[PCI SIG Test Bench Summary](#)” on page 59.

Table 1. PCI Bus Tests Performed on the pci_a		
Test Scenario #	Test Scenario Description	Simulation File Name <i>Note (1)</i>
1.1	PCI bus device speed	pcicc101
1.2	PCI bus single data phase target abort cycles	pcicc102
1.3	PCI bus single data phase target retry cycles	pcicc103
1.4	PCI bus single data phase target disconnect cycles	pcicc104
1.5	PCI bus multi-data phase target abort cycles	pcicc105
1.6	PCI bus multi-data phase target retry cycles	pcicc106
1.7	PCI bus multi-data phase target disconnect cycles	pcicc107
1.8	PCI bus multi-data phase & tr_{dyn} cycles	pcicc108
1.9	PCI bus data parity error single cycles	pcicc109
1.10	PCI bus data parity error multi-data phase cycles	pcicc110
1.11	PCI bus master time-out	pcicc111
1.13	PCI bus master parking	pcicc113
1.14	PCI bus master arbitration	pcicc114
2.5	Target ignores reserved commands (including dual address)	pcicc205
2.6	Target reception of configuration cycles	pcicc206
2.8	Target receives configuration cycles with address and data parity errors	pcicc208
2.9	Target receives memory cycles	pcicc209
2.10	Target receives memory cycles with address and data parity errors	pcicc210

Note:

- (1) The file extension depends on the type of simulation file used, e.g., simulation channel file (.scf), vector file (.vec), or VHDL file.

PCI Bus Signals

The following PCI bus signals are used by the pci_a function:

- *Input*—Standard input-only signal.
- *Output*—Standard output-only signal.
- *Bidirectional*—Tri-state input/output signal.

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

Table 2 summarizes the PCI bus signals interfacing the `pci_a` to the PCI bus. See “Local Side Signals” on page 7 for information on local side signals.

Name	Type	Polarity	Description
<code>clk</code>	Input	–	Clock. The <code>clk</code> input provides the reference signal for all other PCI interface signals, except <code>rstn</code> and <code>intan</code> .
<code>rstn</code>	Input	Low	Reset. The <code>rstn</code> input initializes the FLEX 10K PCI interface circuitry, and can be asserted asynchronously to the PCI bus <code>clk</code> edge. When active, the PCI output signals are tri-stated and the open-drain signals, such as <code>serrn</code> , <code>float</code> .
<code>gntn</code>	Input	Low	Grant. The <code>gntn</code> input indicates to the master device that it has control of the PCI bus. Every master device has a pair of arbitration lines (<code>gntn</code> and <code>reqn</code>) that connect directly to the arbiter.
<code>reqn</code>	Output	Low	Request. The <code>reqn</code> output indicates to the arbiter that the master wants to gain control of the PCI bus to perform a transaction.
<code>ad[31..0]</code>	Tri-State	–	Address/data bus. The <code>ad[31..0]</code> bus is a time-multiplexed address/data bus; each bus transaction consists of an address phase followed by one or more data phases. The data phases occur when <code>irdyn</code> and <code>trdyn</code> are both asserted.
<code>cben[3..0]</code>	Tri-State Master: Output Target: Input	Low	Command/byte enable. The <code>cben[3..0]</code> bus is a time-multiplexed command/byte enable bus. During the address phase this bus indicates the command; during the data phase this bus indicates byte enables.
<code>par</code>	Tri-State	–	Parity. The <code>par</code> signal is a tri-stated output of even parity. The number of 1s on <code>ad[31..0]</code> , <code>cben[3..0]</code> , and <code>par</code> is an even number.

Table 2. PCI Signals Interfacing the pci_a to the PCI Bus (Part 2 of 2)

Name	Type	Polarity	Description
framen <i>Note (1)</i>	Sustained Tri-State Master: Output Target: Input	Low	Frame. The <code>framen</code> is an output from the current bus master that indicates the beginning and duration of a bus operation. When <code>framen</code> is initially asserted, the address and command signals are present on the <code>ad[31..0]</code> and <code>cben[3..0]</code> buses. The <code>framen</code> signal remains asserted during the data operation and is deasserted to identify the end of a transaction.
irdyn	Sustained Tri-State Master: Output Target: Input	Low	Initiator ready. The <code>irdyn</code> signal is an output from a bus master to its target and indicates that the bus master can complete a data transaction. In a write transaction, <code>irdyn</code> indicates that valid data is on the <code>ad[31..0]</code> bus. In a read transaction, <code>irdyn</code> indicates that the master is ready to accept the data on the <code>ad[31..0]</code> bus.
devseln	Sustained Tri-State Master: Input Target: Output	Low	Device select. Target asserts <code>devseln</code> to indicate that the target has decoded its own address.
trdyn <i>Note (1)</i>	Sustained Tri-State Master: Input Target: Output	Low	Target ready. The <code>trdyn</code> is a target output, indicating that the target can complete the current data transaction. In a read operation, <code>trdyn</code> indicates that the target is providing data on the <code>ad[31..0]</code> bus. In a write operation, <code>trdyn</code> indicates that the target is ready to accept data on the <code>ad[31..0]</code> bus.
stopn <i>Note (1)</i>	Sustained Tri-State Master: Input Target: Output	Low	Stop. The <code>stopn</code> signal is a target device request that indicates to the bus master to stop the current transaction.
idsel	Input	High	Initialization device select. The <code>idsel</code> input is a chip select for configuration read or write operations.
perrn	Sustained Tri-State	Low	Parity error. The <code>perrn</code> signal indicates a data parity error.
serrn	Open-Drain	Low	System error. The <code>serrn</code> signal indicates system error and address parity error.
intan	Open-Drain	Low	Interrupt A. The <code>intan</code> signal is an active-low interrupt to the host, and must be used for any single-function device requiring an interrupt capability.

Note:

- (1) To allow the `pci_a` to pass the PCI set-up time requirement, the `framen`, `trdyn`, and `stopn` signals are split into two unidirectional (input, output) signals. For example, the PCI signal `trdyn` is connected to the input `trdyn_in` and the output `trdyn_out`. The input `trdyn_in` is connected to a dedicated input on the FLEX 10K device, and the output `trdyn_out` is connected to an I/O pin on the FLEX 10K device.

The PCI bus and FLEX 10K devices allow Joint Test Action Group (JTAG) boundary-scan testing (BST). To use JTAG BST, designers should connect the PCI bus JTAG pins with the FLEX 10K device JTAG pins. See [Table 3](#).

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

Local Side Signals

[Table 4](#) summarizes the `pci_a` signals that interface the `pci_a` to the local side peripheral device(s).

Name	Type	Polarity	Description
<code>l_irqn</code>	Input	Low	Local side interrupt request. The local side peripheral device asserts <code>l_irqn</code> to signal a PCI bus interrupt. For example, when the local side peripheral device requires a DMA transfer, it could use the <code>l_irqn</code> input to request servicing from the host.
<code>l_holdn</code>	Input	Low	Local hold. When asserted, <code>l_holdn</code> suspends the current DMA transfer. As long as <code>l_holdn</code> is active, data transfers cannot occur between the <code>pci_a</code> and the local side peripheral device.
<code>l_req</code>	Input	High	Local DMA request. The local side peripheral device asserts <code>l_req</code> , which signals the <code>pci_a</code> to request permission for a PCI DMA operation.
<code>l_dat_in[31..0]</code>	Input	–	Local data bus input. The <code>l_dat_in[31..0]</code> input is driven active by the local side peripheral device during <code>pci_a</code> -initiated DMA write transactions (i.e., local side DMA read transactions) and PCI bus target read transactions.

Table 4. *pci_a* Signals Interfacing the *pci_a* to the Local Side (Part 2 of 3)

Name	Type	Polarity	Description
l_addr[18..0]	Output	–	Local target address. The l_addr[18..0] output represents the 128K DWORDS of target register space available to the local side peripheral device.
l_csn	Output	Low	Local target chip select. When active, l_csn notifies the peripheral device of an impending target transaction. Any PCI master device can read or write to a local side peripheral device through the pci_a. The l_ackn and the l_csn outputs are never asserted at the same time.
l_rdn	Output	Low	Read. The pci_a asserts l_rdn to signal a read access to the local side peripheral device. The pci_a uses the l_rdn for reading from peripheral device target registers and for PCI DMA write transactions. A PCI DMA write transaction begins when the pci_a reads data from the peripheral device. For target read operations, the pci_a asserts the l_csn and l_rdn signals. For DMA read operations, the pci_a asserts the l_ackn and l_rdn signals.
l_wrn	Output	Low	Write. The pci_a asserts l_wrn to signal a write access to the local side peripheral device. The pci_a uses the l_wrn output for writing to peripheral device target registers and for PCI DMA read transactions. A PCI DMA read operation begins when the pci_a reads data from memory over the PCI bus and into the internal EAB RAM buffer, and then writes the data out to the peripheral device. For a write operation, the pci_a also asserts either l_csn for target accesses to the local side, or l_ackn for DMA accesses to the local side.
l_ackn	Output	Low	Local DMA acknowledge. When low, l_ackn notifies the local side peripheral device that it has been granted a DMA read or write transaction. The peripheral device can then transfer data to or from the PCI bus through the pci_a.

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

Name	Type	Polarity	Description
l_clk	Output	–	Local PCI clock. The l_clk is a buffered version of the PCI bus clock and is used by the local side peripheral device to synchronize all control logic to the pci_a.
l_reset	Output	High	Local reset. The pci_a asserts the l_reset output to reset the local side peripheral device. The l_reset output is active during a PCI master reset and follows the state of the l_rst bit (bit 2 of the DMA control status register).
l_dat_out[31..0]	Output	–	Local data bus output. The pci_a drives the l_dat_out[31..0] output during pci_a-initiated DMA read transactions (i.e., local side DMA write transactions) and PCI target write transactions.

Function Prototype

The Altera Hardware Description Language (AHDL™) Function Prototype of the pci_a is shown below:

```
FUNCTION pci_a (clk, framen_in, gntn, idsel,
               l_dat_in[31..0], l_holdn, l_irqn, l_req, rstn,
               stopn_in, trdyn_in)

    WITH (SUBSYSTEM_ID, SUBSYSTEM_VEND_ID, DEVICE_ID,
         DEVICE_VEND_ID, CLASS_CODE, REVISION_ID)

    RETURNS (framen_out, l_ackn, l_adr[18..0], l_clk,
            l_csn, l_dat_out[31..0], l_rdn, l_reset, l_wrn,
            stopn_out, trdyn_out, ad[31..0], cben[3..0], devseln,
            intan, irdyn, par, perrn, reqn, serrn);
```

Parameters

The pci_a parameters set read-only PCI bus configuration registers in the pci_a; these registers are called device identification registers. See [“Configuration Registers” on page 15](#) for more information on device ID registers. [Table 5](#) describes the parameters of the pci_a function.

Table 5. Parameters

Name	Value (Hexadecimal)	Default Value (Hexadecimal)	Description
CLASS_CODE	24-bit	FF0000	Class code register
DEVICE_ID	16-bit	0001	Device ID register
DEVICE_VEND_ID	16-bit	1172	Device vendor ID register
REVISION_ID	8-bit	01	Revision ID register
SUBSYSTEM_ID	16-bit	0000	Subsystem ID register
SUBSYSTEM_VEND_ID	16-bit	0000	Subsystem vendor ID register

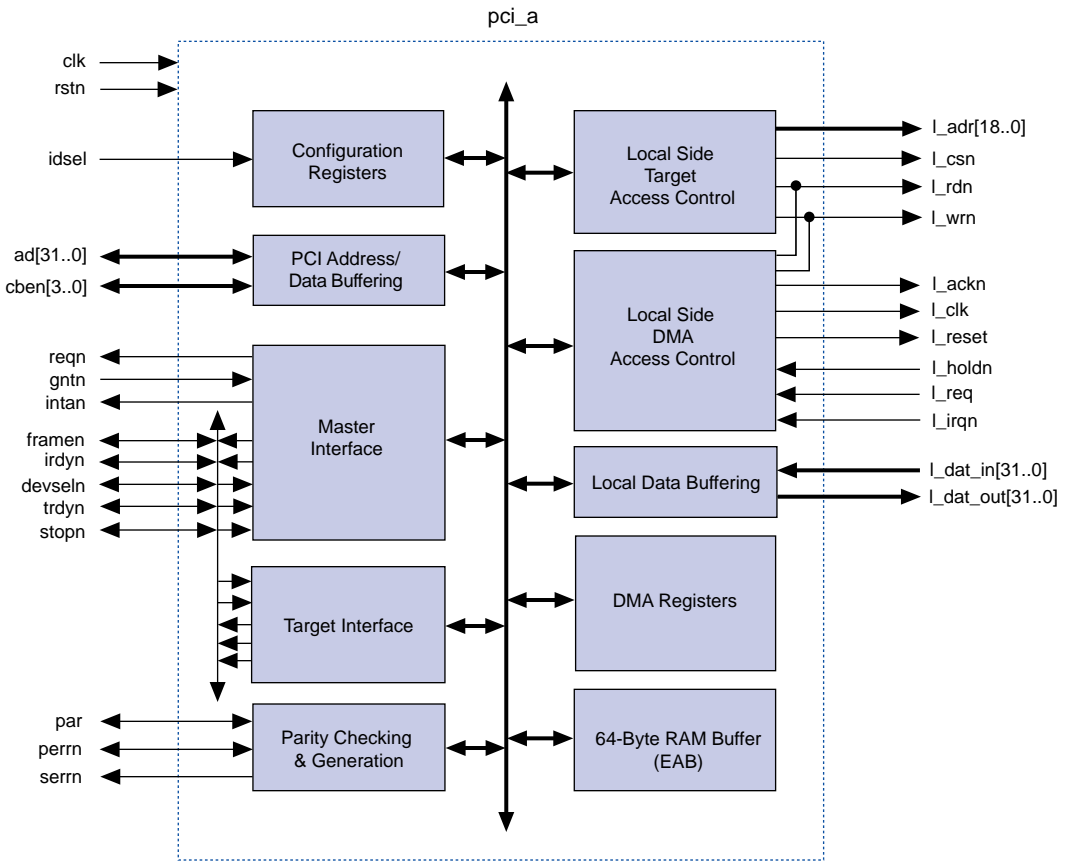
Functional Description

The `pci_a` function consists of three main components:

- A defined 64-byte PCI bus configuration register space and master control logic
- PCI bus target interface control logic, including target decode and register read/write signals
- Embedded DMA control engine, which operates with four registers and includes a 64-byte (16 DWORD) RAM buffer, and local side interface DMA control logic, including read/write control and PCI bus arbitration for master/target accesses

Figure 2 shows the `pci_a` block diagram.

Figure 2. pci_a Block Diagram



Sustained Tri-State Signal Operation

The PCI specification defines signals that are constantly sampled by different bus agents yet driven by one agent at a time, as sustained tri-state signals. For example, `framen` is constantly sampled by different PCI bus targets (to detect the start of a transaction), and yet driven by one PCI bus master at a time.

For sustained tri-state signals, the PCI specification requires using one clock cycle to drive the signals inactive before being tri-stated. The PCI specification also requires that any sustained tri-state signal being released, such as the master device releasing `ad[31..0]` after asserting the address on a read operation, be given a full clock cycle to tri-state before another device can drive it.

The PCI specification defines a turn-around cycle as the clock cycle where a sustained tri-state signal is being tri-stated so that another bus agent can drive it. Turn-around cycles prevent contention on the bus.

Master Device Signals & Signal Assertion

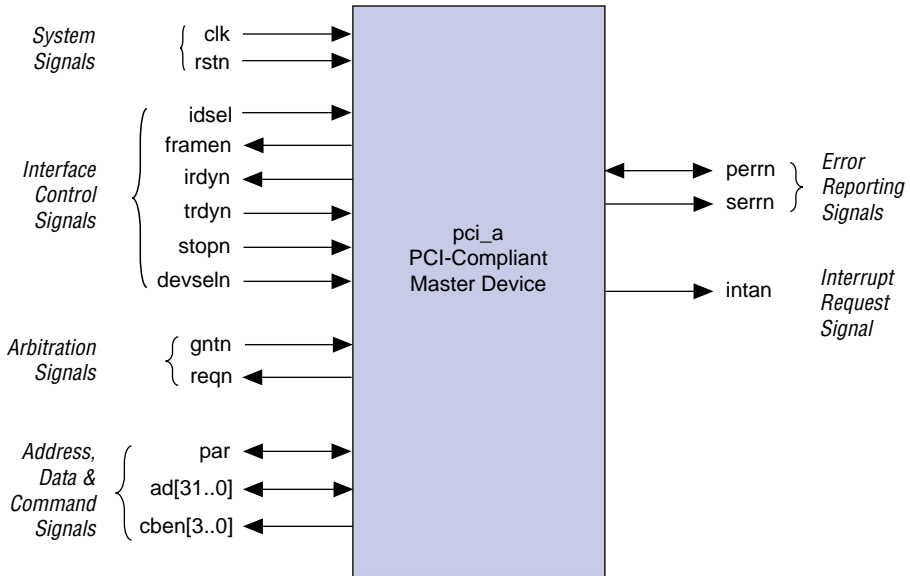
Figure 3 illustrates the PCI-compliant master device signals interfacing the `pci_a` with the PCI bus. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the `pci_a` operating as a master on the PCI bus.

A `pci_a` master sequence begins with the assertion of `reqn` to request mastership of the PCI bus. After receiving `gntn` from the arbiter (usually the PCI host bridge) and after the bus idle state is detected, the `pci_a` initiates the address phase by asserting `framen` and driving both the PCI address on `ad[31..0]` and the bus command on `cben[3..0]` for one clock cycle.

When the `pci_a` is ready to present data on the bus, it asserts `irdyn`. At this point, the `pci_a` master logic monitors the control signals driven by the target device. (A target device is determined by the decoding of the address and command signals presented on the PCI bus during the address phase of the transaction.) The target device drives the control signals `devseln`, `trdyn`, and `stopn` to indicate one of the following:

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

Figure 3. pci_a Master Device Signals



Target Device Signals & Signal Assertion

Figure 4 illustrates the PCI-compliant target device signals interfacing the pci_a with the PCI bus. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the pci_a operating as a target on the PCI bus.

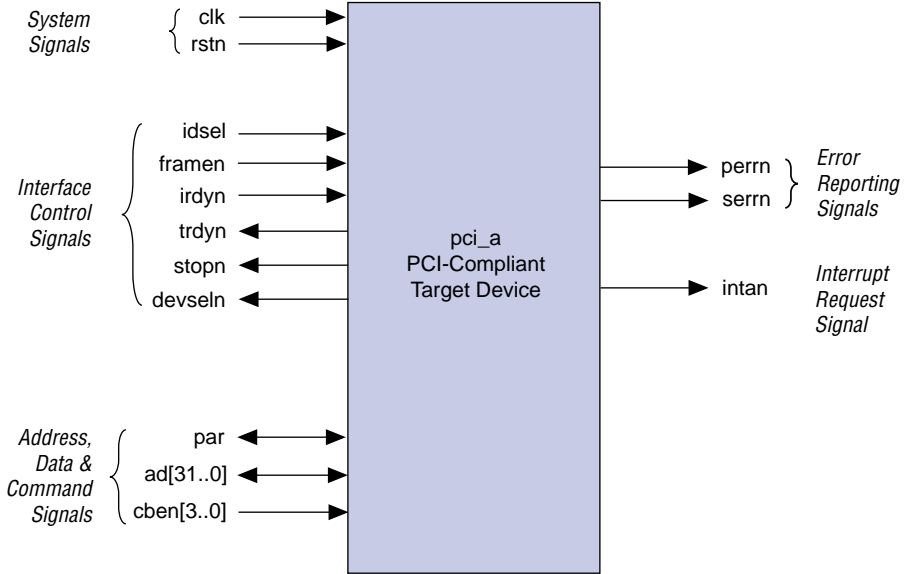
A pci_a target sequence begins when the master device asserts `framen` and drives the address of the target and the command on the PCI bus. When the target device decodes its address on the PCI bus, it asserts `devseln` to indicate to the master that it has accepted the transaction. The master will then assert `irdyn` to indicate to the target device that:

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

When the pci_a functions as the selected target device, it will drive the control signals `devseln`, `trdyn`, and `stopn` as discussed in "Master Device Signals & Signal Assertion" on page 12.

As a target device, the `pci_a` only supports single-cycle accesses; therefore, the `pci_a` simultaneously drives `stopn` and `trdyn` active. When qualified by an active `irdyn` signal, a data word is clocked from the sending to the receiving device.

Figure 4. `pci_a` Target Device Signals



Parity Signal Operation

All bus cycles include parity. Every device that transmits on the `ad[31..0]` bus must also drive the `par` signal, including master devices outputting the address. Because parity on the PCI bus is even, the number of logic 1s on `ad[31..0]`, `cben[3..0]`, and `par` must be even. Parity checking is not required, but can be enabled through the agent's PCI command register. Address parity errors are presented on the `serrn` output, and data parity errors are presented on the `perrn` output. The `par` bit lags the `ad[31..0]` bus by one clock cycle, and parity error signals lag the `par` bit by one clock cycle; thus, parity error signals lag the address or data by two clock cycles.

Bus Master Commands

When the `pci_a` acquires mastership of the PCI bus, it can initiate a memory read or memory write command. During the address phase of a transaction, the `cben[3..0]` bus is used to indicate the transaction type. See [Table 6](#).

The `pci_a` supports memory read/write and configuration read/write commands. When operating as a master device, the `pci_a` executes standard memory read and write operations. When operating as a target, the `pci_a` responds to standard memory read and write transactions. The `pci_a` also responds to configuration read and write operations.

[Table 6](#) summarizes the PCI bus commands that are supported by the `pci_a`.

cben[3..0] Value	Bus Command Cycle	Target Support	Master Support
0110	Memory read	Yes	Yes
0111	Memory write	Yes	Yes
1010	Configuration read	Yes	No
1011	Configuration write	Yes	No

Configuration Registers

Each logical PCI bus device includes a block of 64 configuration DWORDS reserved for the implementation of its configuration registers. The format of the first 16 DWORDS is defined by the PCI SIG's *PCI Local Bus Specification* compliance checklist, revision 2.1, which defines two header formats, type one and type zero. Header type one is used for PCI-to-PCI bridges; header type zero is used for all other devices, including the `pci_a`.

[Table 7](#) displays the defined 64-byte configuration space. The registers within this range are used to identify the device, control PCI bus functions, and provide PCI bus status. The shaded areas indicate registers that are supported by the `pci_a`.

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

Table 8 summarizes the `pci_a`-supported configuration registers address map. Read/write refers to the status at run time, i.e., from the perspective of other PCI bus agents. Designers can set some of the read-only registers at design time by setting the parameters when the `pci_a` is instantiated in MAX+PLUS II. For example, the device ID register value can be modified from its default value by changing the `DEVICE_ID` parameter in MAX+PLUS II. The specified default state is defined as the state of the register when the PCI bus is reset.

Table 8. pci_a-Supported Configuration Registers Address Map

Address Offset (Hexadecimal)	Range Reserved (Hexadecimal)	Bytes Used/ Reserved	Read/Write	Mnemonic	Register Name
00	00-01	2/2	Read	ven_id	Vendor ID
02	02-03	2/2	Read	dev_id	Device ID
04	04-05	2/2	Read/Write	comd	Command
06	06-07	2/2	Read/Write	status	Status
08	08-08	1/1	Read	rev_id	Revision ID
09	09-0B	3/3	Read	class	Class code
0D	0D-0D	1/1	Read/Write	lat_tmr	Latency timer
0E	0E-0E	1/1	Read	header	Header type
10	10-13	4/4	Read/Write	bar0	Base address register zero
2C	2C-2D	2/2	Read	sub_ven_id	Subsystem vendor ID
2E	2E-2F	2/2	Read	sub_id	Subsystem ID
3C	3C-3C	1/1	Read/Write	int_ln	Interrupt line
3D	3D-3D	1/1	Read	int_pin	Interrupt pin
3E	3E-3E	1/1	Read	min_gnt	Minimum grant
3F	3F-3F	1/1	Read	max_lat	Maximum latency

Vendor ID Register (Offset = 00 Hex)

Vendor ID is a 16-bit read-only register that identifies the manufacturer of the device (e.g., Altera for the pci_a). The value of this register is assigned by the PCI SIG; the default value of this register is the Altera vendor ID value, which is 1172 hex. However, by setting the DEVICE_VEND parameter (see Table 5), designers can change the value of the vendor ID register to their PCI SIG-assigned vendor ID value. See Table 9.

Table 9. Vendor ID Register Format

Data Bit	Mnemonic	Read/Write	Definition
15..0	ven_id	Read	PCI vendor ID

Device ID Register (Offset = 02 Hex)

Device ID is a 16-bit read-only register that identifies the type of device. The value of this register is assigned by the manufacturer (e.g., Altera assigned the value of the device ID register for the `pci_a`). The default value of the device ID register is 0001 hex; however, designers can change the value of the device ID register by setting the parameter `DEVICE_ID` (see [Table 5](#)).

Command Register (Offset = 04 Hex)

Command is a 16-bit read and write register that provides basic control over the ability of the `pci_a` to respond to and/or perform PCI bus accesses. See [Table 10](#).

Table 10. Command Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	Unused	–	–
1	<code>mem_ena</code>	Read/Write	Memory access enable. When high, <code>mem_ena</code> enables the <code>pci_a</code> to respond to the PCI bus memory accesses as a target. Because the DMA registers are set via memory target accesses, the <code>mem_ena</code> bit must be set as part of the initialization operation for the <code>pci_a</code> to perform DMA transfers.
2	<code>mstr_ena</code>	Read/Write	Master enable. When high, <code>mstr_ena</code> enables the <code>pci_a</code> to acquire mastership of the PCI bus. For the <code>pci_a</code> to perform DMA transfers, the <code>mstr_ena</code> bit must be set as a part of the initialization operation.
5..3	Unused	–	–
6	<code>perr_ena</code>	Read/Write	Parity error enable. When high, <code>perr_ena</code> enables the <code>pci_a</code> to report parity errors via the <code>perrn</code> output.
7	Unused	–	–
8	<code>serr_ena</code>		System error enable. When high, <code>serr_ena</code> enables the <code>pci_a</code> to report address parity errors via the <code>serrn</code> output. However, to signal a system error, the <code>perr_ena</code> bit must also be high.
15..9	Unused	–	–

Status Register: (Offset = 06 Hex)

Status is a 16-bit register that provides the status of bus-related events. Read transactions to the status register behave normally. However, write transactions are different from typical write transactions in that bits in the status register can be cleared but not set. A bit in the status register is cleared by writing a logic one to that bit. For example, writing the value 4000 hex to the status register clears bit number 14 and leaves the rest of the bits unchanged. The default value of the status register is 0400 hex. See [Table 11](#).

Data Bit	Mnemonic	Read/Write	Definition
7..0	Unused	–	–
8	dat_par_rep	Read/Write	Data parity reported. When high, <code>dat_par_rep</code> indicates that during a read transaction the <code>pci_a</code> asserted the <code>perrn</code> output as a master device, or that during a write transaction the <code>perrn</code> was asserted by a target device. This bit is high only when the <code>perr_ena</code> bit (bit 6 of the command register) is also high.
10..9	devsel_tim	Read	Device select timing. The <code>devsel_tim</code> bits indicate target access timing of the <code>pci_a</code> via the <code>devseln</code> output. The <code>pci_a</code> is designed to be a slow target device.
11	Unused	–	–
12	tar_abrt	Read/Write	Target abort. When high, <code>tar_abrt</code> indicates that the current target device transaction has been terminated.
13	mstr_abrt	Read/Write	Master abort. When high, <code>mstr_abrt</code> indicates that the current master device transaction has been terminated.
14	serr_set	Read/Write	Signaled system error. When high, <code>serr_set</code> indicates that the <code>pci_a</code> drove the <code>serrn</code> output active, i.e., an address phase parity error has occurred.
15	det_par_err	Read/Write	Detected parity error. When high, <code>det_par_err</code> indicates that the <code>pci_a</code> detected either an address or data parity error. Even if parity error reporting is disabled (via <code>perr_ena</code>), the <code>pci_a</code> will set the <code>det_par_err</code> bit.

Revision ID Register (Offset = 08 Hex)

Revision ID is an 8-bit read-only register that identifies the revision number of the device. The value of this register is assigned by the manufacturer (e.g., Altera for the `pci_a`). Therefore, the default value of the revision ID register is set as the revision number of the `pci_a`. See [Table 12](#). However, designers can change the value of the revision ID register by setting the `REVISION_ID` parameter (see [Table 5](#)).

Table 12. Revision ID Register Format

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

Class Code Register (Offset = 09 Hex)

Class code is a 24-bit read-only register divided into three sub-registers: base class, sub-class, and programming interface. Refer to the *PCI Local Bus Specification*, rev. 2.1 for detailed bit information. See [Table 13](#). The default value of the class code register is `FF0000` hex; however, designers can change the value by setting the `CLASS_CODE` parameter (see [Table 5](#)).

Table 13. Class Code Register Format

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

Latency Timer Register (Offset = 0D Hex)

The latency timer register is an 8-bit register with bits 2, 1, and 0 tied to `GND`. The register defines the maximum amount of time, in PCI bus clock cycles, that the `pci_a` can retain ownership of the PCI bus. After initiating a transaction, the `pci_a` decrements its latency timer by one on the rising edge of each clock. The default value of the latency timer register is `00` hex. See [Table 14](#).

Table 14. Latency Timer Register Format

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

Header Type Register (Offset = 0E Hex)

Header type is an 8-bit read-only register that identifies the `pci_a` as a single-function device. The default value of the header type register is 00 hex. See [Table 15](#).

Table 15. Header Type Register Format

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

Base Address Register Zero (Offset = 10 Hex)

Base address register zero (BAR0) consists of a 12-bit register (bits 31 through 20) that determines the base memory address of the `pci_a` target space. The default value of the base address register is 0000.0000 hex. See [Table 16](#).

Table 16. Base Address Register Format

Data Bit	Mnemonic	Read/Write	Definition
0	mem_ind	Read	Memory indicator. The <code>mem_ind</code> bit indicates whether the register is I/O or a memory address decoder. In the <code>pci_a</code> , the <code>mem_ind</code> bit is tied to GND, which indicates a memory address decoder.
2..1	mem_type	Read	Memory type. The <code>mem_type</code> bits indicate the type of memory that can be implemented in the <code>pci_a</code> memory address space. These bits are tied to GND, which indicates that the memory block can be located anywhere in the 32-bit address space.
3	pre_fetch	Read	Memory prefetchable. The <code>pre_fetch</code> bit indicates whether the block of memory defined by BAR0 is prefetchable by the host bridge. In the <code>pci_a</code> , the address space is not prefetchable, i.e., it reads as low.
19..4	Unused	–	–
31..20	bar0	Read/write	Base address register 0.

Subsystem Vendor ID Register (Offset = 2C Hex)

Subsystem vendor ID is a 16-bit read-only register that identifies add-in cards designed by different vendors but with the same functional device on the card. The value of this register is assigned by the PCI SIG. See [Table 17](#). The default value of the subsystem vendor ID register is 0000 hex; however, designers can change the value by setting the SUBSYSTEM_VEND_ID parameter (see [Table 5](#)).

<i>Table 17. Subsystem Vendor ID Register Format</i>			
Data Bit	Mnemonic	Read/Write	Definition
15..0	sub_vend_id	Read	PCI subsystem/vendor ID

Subsystem ID Register (Offset = 2E Hex)

Subsystem ID register identifies the subsystem; the value of this register is defined by the subsystem vendor, i.e., the designer. See [Table 18](#). The default value of the subsystem ID register is 0000 hex; however, designers can change the value by setting the SUBSYSTEM_ID parameter (see [Table 5](#)).

<i>Table 18. Subsystem ID Register Format</i>			
Data Bit	Mnemonic	Read/Write	Definition
15..0	sub_id	Read	PCI subsystem ID

Interrupt Line Register (Offset = 3C Hex)

The interrupt line register consists of an 8-bit register that defines to which system interrupt request line (on the system interrupt controller) the intan output is routed. The interrupt line register is written to by the system software on power-up; the default value is FF hex. See [Table 19](#).

<i>Table 19. Interrupt Line Register Format</i>			
Data Bit	Mnemonic	Read/Write	Definition
7..0	int_ln	Read/write	Interrupt line register

Interrupt Pin Register (Offset = 3D Hex)

The interrupt pin register consists of an 8-bit read-only register that defines the `pci_a` PCI bus interrupt request line to be `intan`. The default value of the interrupt pin register is 01 hex. See [Table 20](#).

Table 20. Interrupt Pin Register Format

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

Minimum Grant Register (Offset = 3E Hex)

Minimum grant register consists of an 8-bit read-only register that defines the length of time the `pci_a` would like to retain mastership of the PCI bus. The value set in this register indicates the required burst period length in 250-ns increments. The `pci_a` requests a timeslice of 4 microseconds. The default state of the minimum grant register is 10 hex. See [Table 21](#).

Table 21. Minimum Grant Register Format

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

Maximum Latency Register (Offset = 3F Hex)

The maximum latency register is an 8-bit read-only register that defines the frequency in which the `pci_a` would like to gain access to the PCI bus. The value of the maximum latency register is set to 00 hex, which indicates that the `pci_a` has no major requirements for maximum latency. See [Table 22](#).

Table 22. Maximum Latency Register Format

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

PCI Bus Transactions

This section describes `pci_a` PCI bus transactions. The following should be considered when reading the diagrams in this section:

- All `pci_a` accesses to the PCI bus are quad-byte, or 32-bit transfers; therefore, all byte enables are active for the duration of data transfers.
- Although [Figures 5 through 14](#) show PCI bus signals as tri-stated when not driven by the `pci_a`, they are actually high due to the pull-up resistors used to keep sustained tri-state signals at a logic high while the signals are not being driven by a PCI bus agent.

The `pci_a` accesses the PCI bus for three types of transactions:

- Target
- Device configuration
- Master

Target Transactions

A target read or write transaction begins after the master acquires mastership of the PCI bus and asserts `framen` to indicate the beginning of a bus transaction. The `pci_a` latches the address and command signals on the first clock edge when `framen` is asserted and starts the address decode phase.

Target Read Transactions

The `pci_a` supports two types of target read transactions:

- *Internal target read*—Target read transaction from the internal DMA registers
- *External target read*—Target read transaction from the local side target memory space

The sequence of events in both target read transactions is identical; however, the timing is not. A target read transaction from the local side target memory space requires more time because the `pci_a` must wait for the local side to supply it with data.

Internal Target Read Transaction

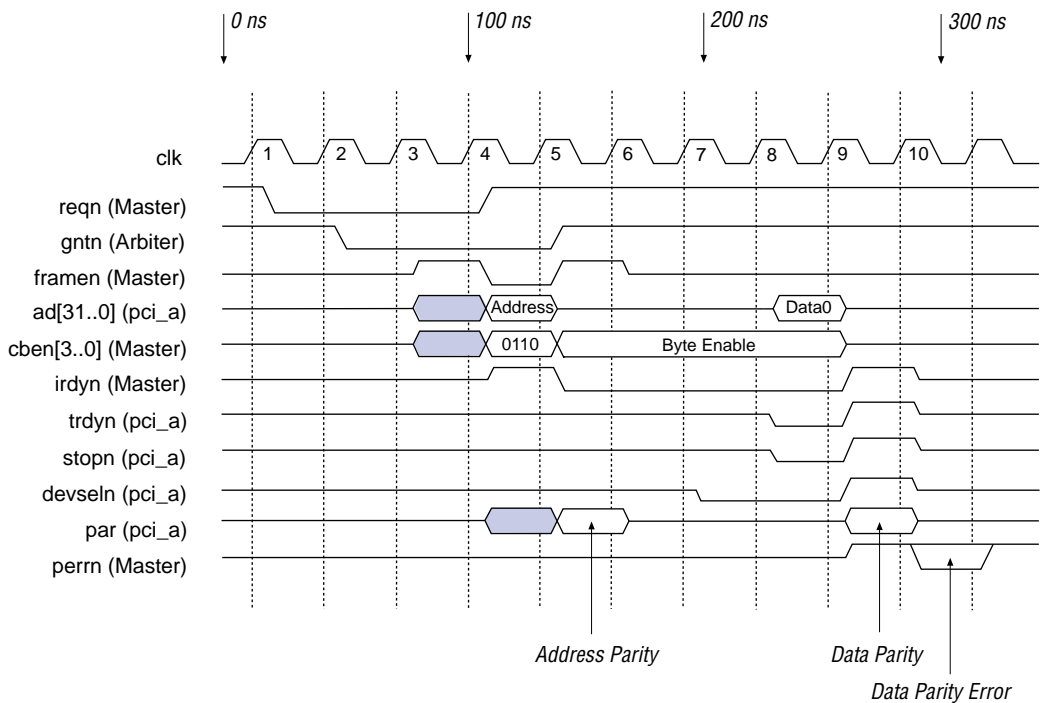
Immediately after the address phase (clock 4), the master deasserts `framen` and asserts `irdyn`, indicating both of the following:

- The transaction contains a single data phase.
- The master device is ready to read the data that the `pci_a` has presented on the `ad[31..0]` bus.

The master device tri-states the $ad[31..0]$ bus in clock five after the pci_a latches the address. The pci_a can drive the $ad[31..0]$ bus beginning in clock six. If the master is attempting a burst access, it will keep both $framen$ and $irdyn$ signals asserted. However, because the pci_a does not support target bursts, it will assert $stopn$ to indicate a disconnect to the master. The master will subsequently end the transaction by deasserting $framen$ and asserting $irdyn$ for one clock cycle.

In Figure 5, the pci_a asserts $devseln$ in clock seven, which indicates to the master device that the pci_a has accepted the transaction. The $devseln$ is then sampled by the master device on the rising-edge of clock eight, which is slow decode timing. The pci_a is a slow decode device during target and configuration read or write transactions. Figure 5 shows the timing of a pci_a internal target read transaction.

Figure 5. Internal Target Read Transaction



In [Figure 5](#), the `pci_a` asserts `trdyn` and `stopn` in clock eight to indicate that valid data is on the `ad[31..0]` bus and a disconnect is desired. Data is transferred during clock eight when `irdyn` and `trdyn` are active and latched by the master device on the rising-edge of clock nine. In the case of an attempted burst transfer, the PCI specification requires that a target device that does not support burst transfers must issue a disconnect during the first data phase. Because of the PCI specification, the `pci_a` always asserts `stopn` and `trdyn` at the same time.

The master drives the `par` active in clock five for address parity, and the `pci_a` drives `par` active in clock nine for data parity. In a target read transaction, the master device drives the `perrn` signal to indicate data parity errors.

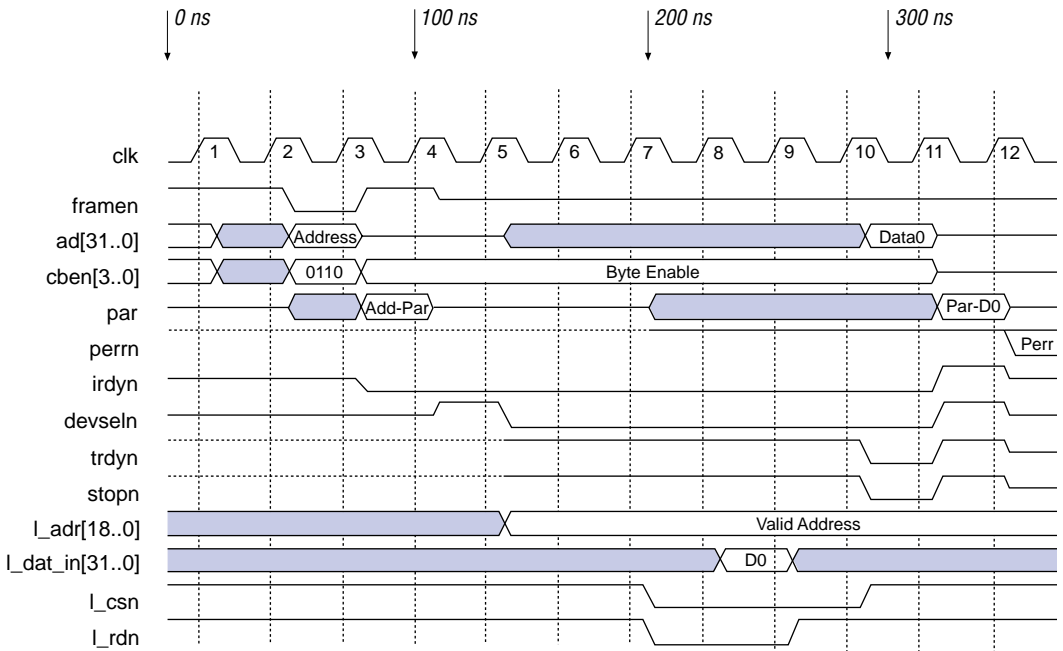
In clock nine, because the data has been sampled, the `pci_a` releases the `ad[31..0]` bus and the master releases `cben[3..0]`. The `devseln`, `trdyn`, and `stopn` signals are driven high in clock nine and released by the `pci_a` one clock later. Thus, the sustained tri-state signal requirement is met, i.e., driving the signal high for one clock cycle before releasing it.

External Target Read Transaction

Again, the sequence of events in an external target read transaction is identical to an internal target read transaction. However, because the `pci_a` must wait for the local side to supply it with data, a target read transaction from the local side target memory space requires more time. Also, because a DMA access to the local side takes precedence over any other access to the local side, an external target read transaction is allowed to complete only when the DMA is in an idle state. If an external target read transaction is received by the `pci_a` while the DMA is not in an idle state, the `pci_a` signals a retry.

[Figure 6](#) shows the timing of a `pci_a` external target read transaction.

Figure 6. External Target Read Transaction



Target Write Transactions

The pci_a supports two types of target write transactions:

- Internal target write: Target write to internal DMA registers
- External target write: Target write to the local side target memory space

The sequence of events in both target write transactions is identical; however, the timing may not be.

Internal Target Write Transaction

Immediately after the address phase, the master deasserts framen and asserts irdyn, indicating the following:

- The transaction contains a single data phase.
- The master device is ready to write data on the ad[31..0] bus for the target device to receive.

If the master device is not ready for the data phase to begin, `irdyn` is delayed and `framen` is not deasserted until the clock where `irdyn` goes active. If the master is attempting a burst access, it will keep both `framen` and `irdyn` signals asserted. However, because the `pci_a` does not support target bursts, it will assert `stopn` to indicate a disconnect to the master. The master will subsequently end the transaction by deasserting `framen` and asserting `irdyn` for one clock cycle.

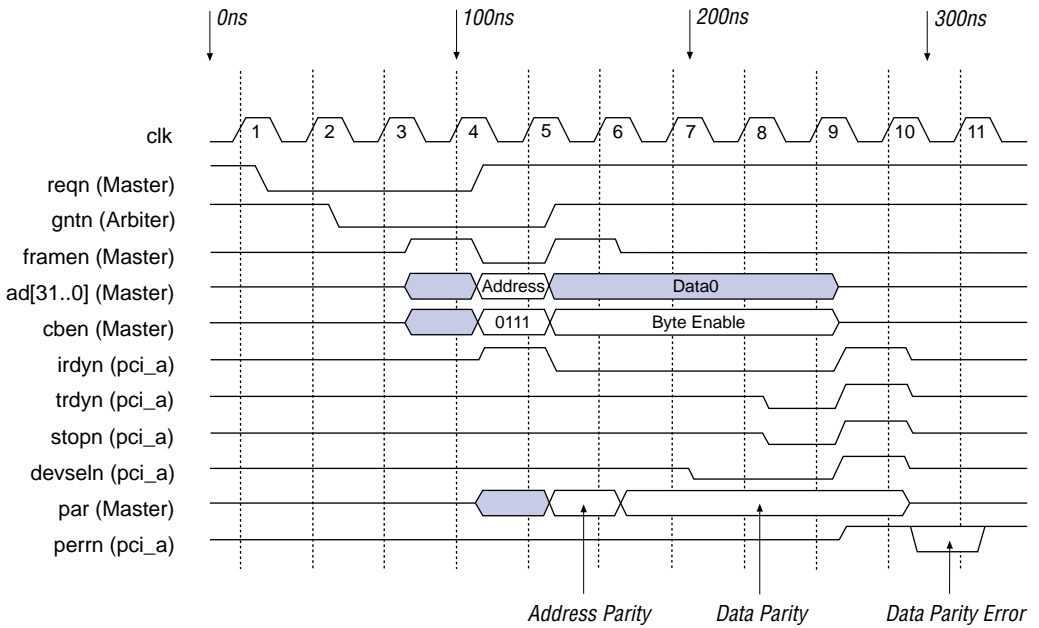
In [Figure 7](#), the `pci_a` asserts `devseln` in clock seven, which indicates to the master device that the `pci_a` has accepted the transaction. The `devseln` signal is then sampled by the master device on the rising edge of clock eight, which is slow decode timing.

In [Figure 7](#), where the data phase occurs in clocks five through eight, the master device drives data on the `ad[31..0]` bus simultaneously with the assertion of `irdyn` in clock five. To indicate completion of the data phase and to request a bus disconnect, the `pci_a` asserts `trdyn` and `stopn` in clock eight. Data is transferred during clock eight when `irdyn` and `trdyn` are active, and latched by the `pci_a` on the rising edge of clock nine. In the case of an attempted burst write transfer, the PCI specification requires that a target device that does not support burst transfers must issue a disconnect during the first data phase. Because of the PCI specification, the `pci_a` always asserts `stopn` and `trdyn` at the same time.

The master device drives `par` active in clock five for parity of the address bits, and clock six for parity of the data bits. In a target write transaction, the `pci_a` drives `perrn` to indicate data parity errors.

In clock nine, because the data has been sampled, the `pci_a` releases the `ad[31..0]` and `cben[3..0]` buses. One clock later `par` is released by the master device and `devseln`, `trdyn`, and `stopn` are release by the `pci_a`. The `devseln`, `trdyn`, and `stopn` are driven high in clock nine and released one clock later. [Figure 7](#) shows the timing of an internal target write transaction.

Figure 7. Internal Target Write Transaction

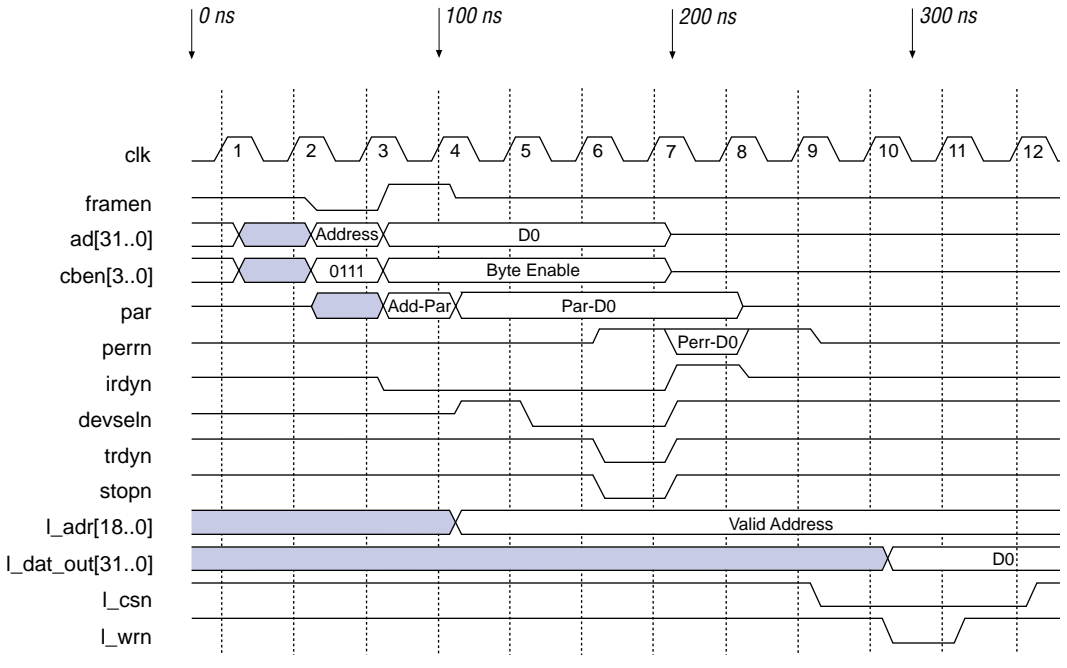


External Target Write Transaction

Again, the sequence of events in an external target write transaction is identical to an internal target write transaction. However, the timing may be different.

To allow an external target write transaction to complete faster, the `pci_a` provides a single address and a single data holding register. When an external target write access takes place, the `pci_a` stores the address and data in its internal holding registers and completes the transfer on the PCI bus. The `pci_a` will subsequently assert its `l_csn` signal to indicate to the local side that there is a pending target access; once clock later (clock 10), the `l_wrn` and `l_dat_out` signals are also asserted. Figure 8 shows the timing of an external target write transaction.

Figure 8. External Target Write Transaction



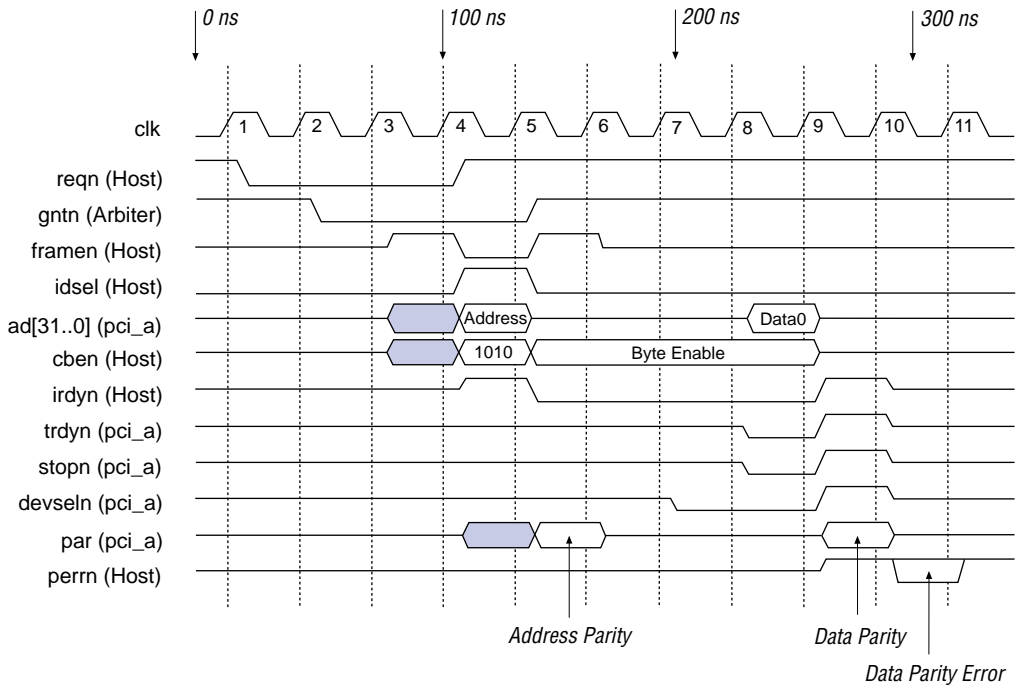
Configuration Transactions

A configuration transaction is generated by either a host-to-PCI bridge or PCI-to-PCI bridge access. In the address phase of a configuration transaction, the PCI bridge will drive the `idse1` signal of the PCI bus agent that it wants to access. If a PCI bus agent decodes the configuration command and detects its `idse1` to be high, the agent will claim the configuration access and assert `devsel`.

PCI Configuration Read Transaction

Figure 9 shows the timing of a `pci_a` configuration read transaction. The protocol is identical to the protocol discussed in the “[Target Read Transactions](#)” on page 24 except for the `idse1` signal, which is active during the address phase of a configuration transaction.

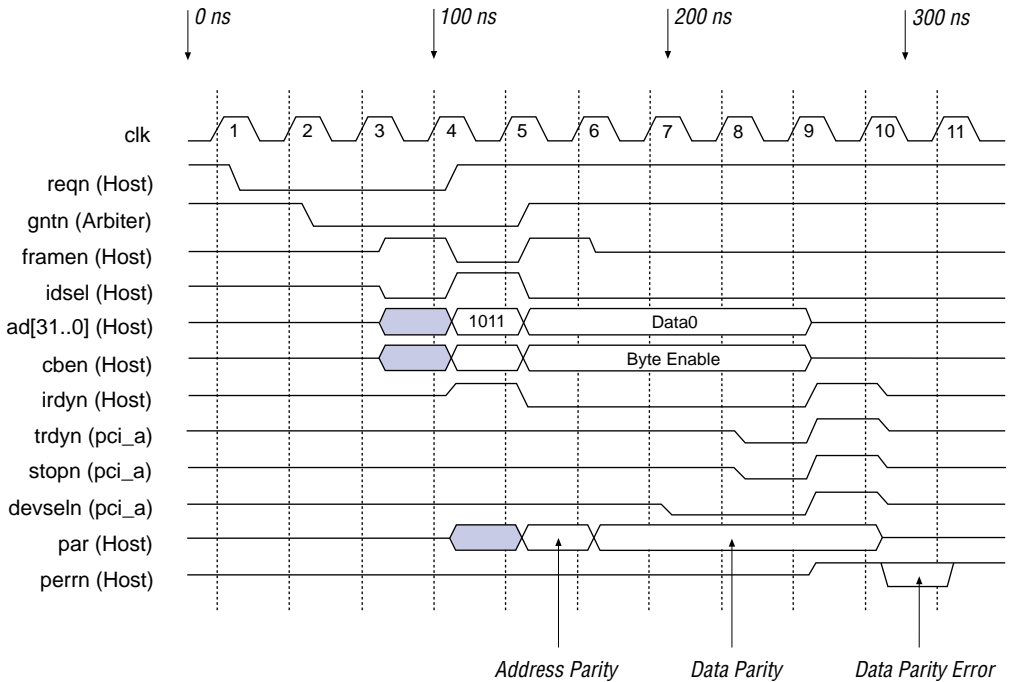
Figure 9. Configuration Read Transaction



PCI Configuration Write Transaction

Figure 10 shows the timing of a pci_a configuration write transaction. The protocol is identical to the protocol discussed in the “Target Write Transactions” on page 27 except for the idsel signal, which is active during the address phase of a configuration transactions.

Figure 10. Configuration Write Transaction



Master Transactions

Master transactions in the `pci_a` are controlled by the DMA engine. A `pci_a` master transaction begins after the user loads the appropriate values in the DMA register (see [“General Programming Guidelines” on page 47](#) for more detailed information on DMA register loading). The `pci_a` waits for the local side to assert `l_req`, which indicates to the `pci_a` that it can begin the DMA operation.

In a DMA read (PCI to local side) transaction, the `pci_a` immediately asserts `reqn` to acquire mastership of the PCI bus. After the arbiter asserts `gntn`, the `pci_a` begins the address phase by asserting `framen` and driving the address on the `ad[31..0]` bus and the command on the `cben[3..0]` bus.

In a DMA write (local side to PCI) transaction, the `pci_a` first reads up to 16 DWORDS from the local side and stores them in its internal RAM buffer. At this point, the DMA asserts `reqn` to acquire mastership of the PCI bus. After the arbiter asserts `gntn`, the `pci_a` begins the address phase.

Master Read Transactions

The `pci_a` supports two types of master read transactions:

- Single-cycle master read
- Master burst read

Single-Cycle Master Read Transaction

In a master read transaction, data is being transferred from the PCI side to the local side. Assuming the `pci_a` has acquired mastership of the PCI bus, the start of a master read transaction is indicated when the `pci_a` asserts `framem`.

After the master read transaction is initiated, the target devices latch the address and command on the clock edge when `framem` is active and start the address decode. The `pci_a` is not ready to read data until clock five; therefore, `framem` is not deasserted and `irdyn` is not asserted until clock five.

The selected target device asserts `devseln` in clock three, and `devseln` is sampled by the `pci_a` on the rising-edge of clock four, which depicts a fast decode target device.

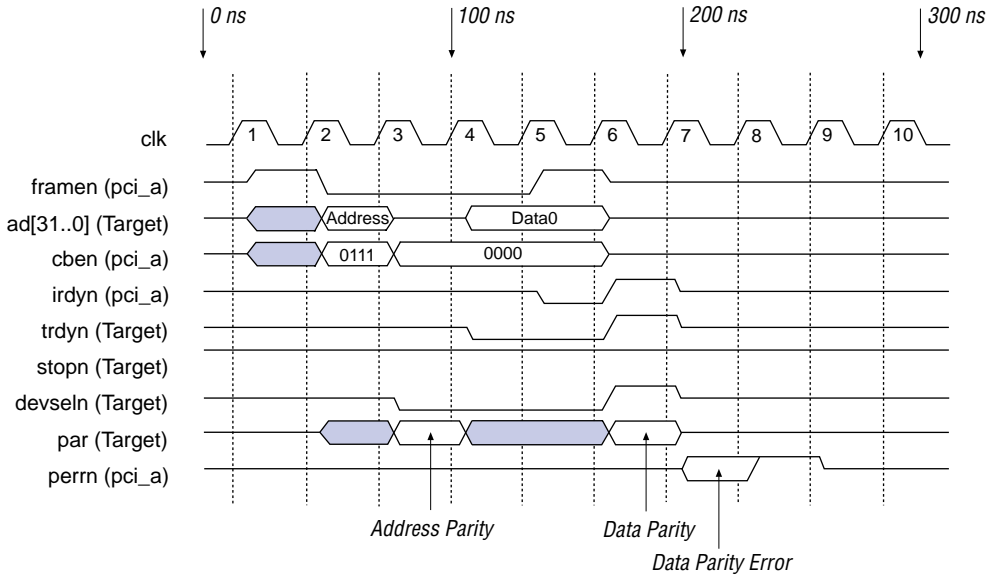
To indicate that it is ready to send data, the target device simultaneously asserts `trdyn` and drives data on the `ad[31..0]` bus beginning in clock four. The data phase begins in clock five when `irdyn` and `trdyn` are active and finishes on the rising edge of clock six with data latched by the `pci_a`.

The `pci_a` drives the `par` signal active in clock three for parity of the address and command bits, and the selected target drives `par` active in clock six for parity of the data and byte enable bits.

The `pci_a` releases the `ad[31..0]` bus in clock three, the `cben[3..0]` bus in clock six, and the `par` signal in clock four.

Figure 11 shows the timing of a `pci_a` master read transaction. The figure assumes the `pci_a` has already acquired mastership of the PCI bus.

Figure 11. Single-Cycle Master Read Transaction



Master Burst Read Transaction

The protocol for the address phase of a master burst read transaction is identical to “Single-Cycle Master Read Transaction” on page 33. After the address phase, the protocol changes to reflect the additional read transactions.

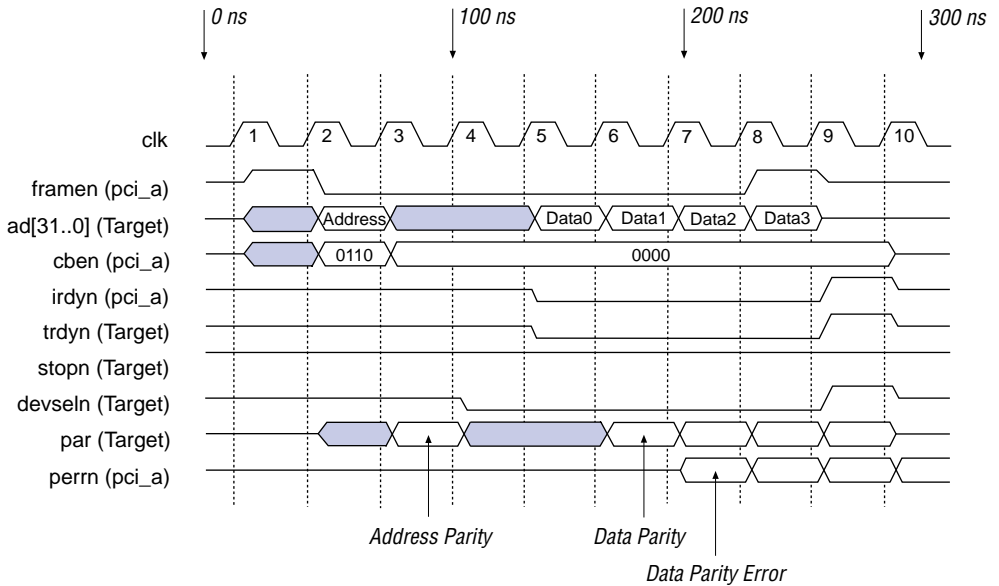
After the master burst read transaction is initiated, the selected target device asserts `devseln` in clock three, and the `pci_a` samples `devseln` on the rising edge of clock five. This example displays a fast decode target. The target device then signals to the `pci_a` that it is ready to send data by driving `trdyn` and the `ad[31..0]` bus active in clock four.

The `pci_a` drives `par` active in clock three for parity of the address and command bits. In clock six the target device drives `par` active for parity of the first data phase (Data0). The target device also drives `par` active in clocks 7, 8, and 9 for parity of the second, third and fourth data phases.

Figure 12 shows a 16-byte data transaction, with the data phases occurring in four consecutive clock cycles. The data phase begins in clock five and ends in clock eight when the `pci_a` releases `framen`, which indicates the start of the final data phase.

Because the data has been read, the target device simultaneously releases `devseln`, `trdyn`, and the `ad[31..0]` bus when the `pci_a` releases `irdyn` in clock nine.

Figure 12. Master Burst Read Transaction



Master Write Transactions

The `pci_a` supports two types of master write transactions:

- Single-cycle master write
- Master burst write

Single-Cycle Master Write Transaction

In a master write transaction, data is transferred from the local side to the PCI side. Assuming the `pci_a` has acquired mastership of the PCI bus, the start of a master device write transaction is indicated when the `pci_a` asserts `framen`.

After the master device write transaction is initiated, the target devices latch the address and command on the clock edge when `framen` is active and start the address decode. Data from `pci_a` master device write transactions is not available until clock five; therefore, `framen` is not deasserted and `irdy` is not asserted until clock five.

The selected target device asserts `devseln` in clock four and is sampled by the `pci_a` in clock five, which depicts a medium decode target device.

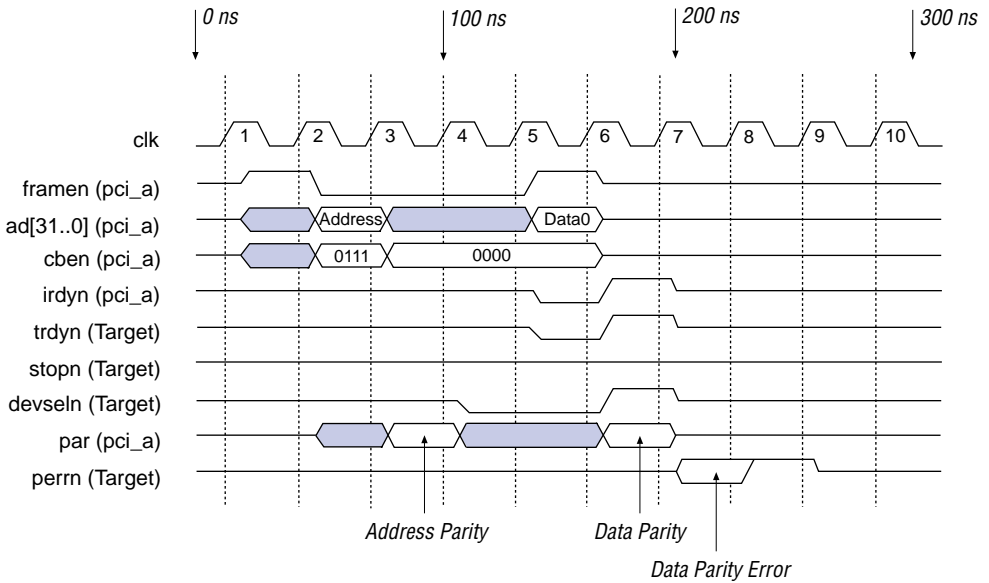
To indicate that it is ready to receive data, the target device drives `trdyn` active in clock five. Then, the `pci_a` drives data on the `ad[31..0]` bus beginning in clock five and simultaneously with the assertion of `irdyn`. The data phase begins in clock five when `irdyn` and `trdyn` are active, and ends on the rising-edge of clock six with data latched by the selected target device.

The `pci_a` drives `par` active in clock three for parity of the address and command bits and clock six for parity of the data and byte enable bits.

Because the data phase is complete, the `pci_a` releases the `ad[31..0]` bus and `cben[3..0]` in clock six. One clock later, `par` is released by the `pci_a`, and `devseln` and `trdyn` are released by the target device. To meet the requirement of driving a sustained tri-state signal high for one clock cycle before releasing it, the `pci_a` drives `irdyn` high in clock six before releasing it in clock seven.

Figure 13 shows the timing of a `pci_a` master write transaction. The figure assumes the `pci_a` has already acquired mastership of the PCI bus.

Figure 13. Single-Cycle Master Write Transaction



Master Burst Write Transaction

The protocol for master burst write transactions from the address phase to data phase one is identical to “Single-Cycle Master Write Transaction” on page 35. From data phase two, the protocol changes to reflect the additional write transactions.

After the master burst write transaction is initiated, the selected target device asserts devseln in clock four, and the pci_a samples devseln on the rising edge of clock five. This example depicts a medium decode target. The target device signals to the master device that it is ready to receive data by driving trdyn active in clock five.

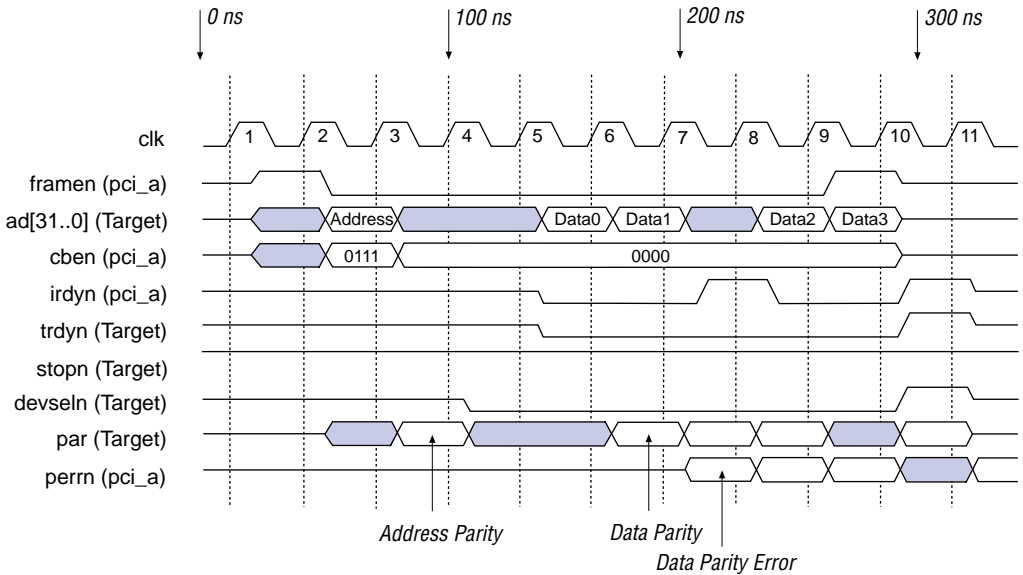
The master burst write transaction example in Figure 14 shows the data phases occurring in clocks five, six, seven, and nine when irdyn and trdyn are both active.

To ensure data synchronization on the `pci_a` internal data path pipeline, a wait state for master burst write transactions is inserted by the `pci_a` in clock eight. If the target does not insert a wait state during the burst write transaction, the `pci_a` will insert only one wait state for the entire burst transfer. However, if the target inserts additional wait states during the burst write transaction, the `pci_a` will insert additional wait states. The final data transfer occurs when the `pci_a` simultaneously asserts `irdyn` and deasserts `framen` in clock nine.

The `pci_a` drives the `par` active in clock three for parity of the address bits and clock six for parity of the data bits.

Figure 14 shows the timing of a `pci_a` burst write transaction, which depicts a 16-byte data transfer.

Figure 14. Master Burst Write Transaction



DMA Operation

This section provides operating details of the DMA engine, and is divided into the following sub-sections:

- Target address space
- Internal target registers memory map
- DMA registers
- DMA Transactions
- General programming guidelines

Target Address Space

The `pci_a` memory-mapped target registers (internal and external) are read and/or written over the PCI bus in BAR0 memory space. Accesses to or from BAR0 memory space occur in 32-bit transfers. [Table 23](#) lists the `pci_a` memory space address map. The `pci_a` BAR0 address space is 1 Mbyte of contiguous address divided into two 512-kbyte spaces. The lower 512-kbyte region (internal target address space) contains the `pci_a` DMA control registers, and the upper region (external target address space) contains user-defined memory space.

Table 23. Memory Space Address Map

Memory Space	Block Size (DWORDS)	Address Offset	Words Used/Reserved	Read/ Write	Description
BAR0	128k	00000h-7FFFFh	4/128k	Read/Write	DMA registers
BAR0	128k	80000h-FFFFFFh	128k/128k	Read/Write	User-defined memory space (128k DWORDS)

Internal Target Registers Memory Map

Internal `pci_a` target address space is used for the DMA registers, including the DMA control/status register, DMA address counter register, DMA byte counter register and the interrupt status register. [Table 24](#) lists the `pci_a` DMA registers memory map.

Table 24. Internal Target Registers Memory Map

Range Reserved	Bytes Used/Reserved	Read/Write	Mnemonic	Default State (Hexadecimal)	Register Name
00000h-00003h	1/4	Read/write	dma_csr	00000000	DMA control/status
00004h-00007h	4/4	Read/write	dma_acr	00000000	DMA address counter
00008h-0000Bh	2/4	Read/write	dma_bcr	00000000	DMA byte counter
0000Ch-0000Fh	1/4	Read	dma_isr	00000000	DMA interrupt status

DMA Registers

This section describes the DMA registers. The specified default state is defined as the state of the storage element when the PCI bus is reset. The pci_a contains the following DMA registers:

- Control and status
- Address counter
- Byte counter
- Interrupt status

Control & Status Register (Offset = 00000 Hex)

The DMA control and status register (dma_csr) configures the pci_a DMA engine, directs the pci_a DMA operation, and provides status of the current memory transfer. See Table 25.

Table 25. DMA Control & Status Register Format (Part 1 of 2)

Data Bit	Mnemonic	Read/Write	Definition
0	int_ena	Read/write	PCI interrupt enable. The int_ena bit enables the intan output when either the err_pend or dma_tc bits are driven high from the dma_isr, or when the l_irqn signal is active.
1	flush	Write	Flush buffer. When high, flush marks all bytes in the internal EAB RAM queue as invalid and resets dma_tc and ad_loaded (bits 3 and 4 of the interrupt status register). The flush bit also resets itself; therefore, it always reads as zero. The flush bit should never be set while dma_on is set, because a DMA transfer is in progress.
2	l_rst	Read/write	Local reset. This bit serves as a software reset to the local side add-on logic (see “Local Side Signals” on page 7). The l_reset output of the pci_a is active as long as the l_rst bit is high. (The l_reset output is also active for PCI bus resets.)

Table 25. DMA Control & Status Register Format (Part 2 of 2)

Data Bit	Mnemonic	Read/Write	Definition
3	<code>write</code>	Read/write	Memory read/write. The <code>write</code> bit determines the direction of the <code>pci_a</code> DMA transfer. When <code>write</code> is high, the data flows to local memory from the PCI bus (PCI bus write); when <code>write</code> is low, the data flows from the local memory to the PCI bus (PCI bus read).
4	<code>dma_ena</code>	Read/write	DMA enable. When high, <code>dma_ena</code> allows the <code>pci_a</code> to respond to DMA requests from the local side (<code>int_req</code>) as long as the PCI bus activity is not stopped due to a pending interrupt, etc.
5	<code>tci_dis</code>	Read/write	Transfer complete interrupt disable. When high, <code>tci_dis</code> disables <code>dma_tc</code> (bit 3 of the DMA interrupt status register) from generating PCI bus interrupts.
6	<code>dma_on</code>	Read	DMA on. When high, <code>dma_on</code> indicates that the <code>pci_a</code> can request mastership of the PCI bus (<code>reqn</code>) if prompted by the local side (i.e., an active <code>l_req</code>). The <code>dma_on</code> bit is high when the address is loaded (<code>ad_loaded</code>), the DMA is enabled, and there are no pending errors. The DMA transfer sequence actually begins when the <code>dma_on</code> bit becomes set. Under normal conditions (i.e., DMA is enabled and no errors are pending) the <code>dma_on</code> bit becomes set when a write transaction to the DMA address counter register occurs.
31..7	Unused	–	–

Address Counter Register (Offset = 00004 Hex)

The DMA address counter register (`dma_acr`) is a 32-bit register consisting of a 30-bit counter (bits 31..2) and 2 bits (bits 1..0) tied to GND. The `dma_acr` contains the PCI bus address for the current memory transfer and is incremented after every data transfer on the PCI bus. PCI bus memory transfers initiated by the `pci_a` must begin on DWORD boundaries. Reading the `dma_acr` during a memory transfer is permitted and can be used for monitoring progress, but is not recommended for maximum efficiency. See [Table 26](#).

Table 26. DMA Address Counter Register Format

Data Bit	Name	Read/Write	Definition
1..0	<code>dma_acr</code>	Read	Bits are tied to GND.
31..2	<code>dma_acr</code>	Read/write	30-bit counter.

Byte Counter Register (Offset = 00008 Hex)

The DMA byte counter register (`dma_bcr`) is a 16-bit register consisting of a 14-bit counter (bits 15..2) and 2 bits (bits 1..0) tied to GND. The `dma_bcr` holds the byte count for the current `pci_a`-initiated memory transfer and decrements (by 4 bytes) after every data transfer on the PCI bus. PCI bus memory transfers initiated by the `pci_a` must be DWORD transfers. Reading the `dma_bcr` during a memory transfer is permitted and can be used for monitoring progress, but is not recommended for maximum efficiency. See [Table 27](#).

Table 27. DMA Byte Counter Register Format

Data Bit	Name	Read/Write	Definition
1..0	<code>byte_cntr</code>	Read	Bits are tied to GND.
15..2	<code>byte_cntr</code>	Read/write	14-bit counter.
31..16	Unused	–	–

Interrupt Status Register (Offset = 0000C Hex)

The DMA interrupt status register (`dma_isr`) provides all interrupt source status signals to the interrupt handler. See [Table 28](#).

Table 28. DMA Interrupt Status Register Format (Part 1 of 2)

Data Bit	Mnemonic	Read/Write	Definition
0	<code>int_pend</code>	Read	The <code>pci_a</code> automatically asserts <code>int_pend</code> to indicate that a <code>pci_a</code> interrupt is pending. The three possible interrupt signals from the <code>pci_a</code> are <code>err_pend</code> , <code>dma_tc</code> , and <code>int_irq</code> .
1	<code>err_pend</code>	Read	When high, <code>err_pend</code> indicates that an error occurred during a <code>pci_a</code> -initiated PCI bus memory transfer, and that the interrupt handler must read the PCI configuration status register and clear the appropriate bits. Any one of the following three PCI status register bits can assert <code>err_pend</code> : <code>mstr_abrt</code> , <code>tar_abrt</code> , and <code>det_par_err</code> . See “ Control & Status Register (Offset = 00000 Hex) ” on page 40.
2	<code>int_irq</code>	Read	When high, <code>int_irq</code> indicates that the local side is requesting an interrupt, i.e., the <code>l_irqn</code> input is asserted.

Table 28. DMA Interrupt Status Register Format (Part 2 of 2)

Data Bit	Mnemonic	Read/Write	Definition
3	<code>dma_tc</code>	Read	When high, <code>dma_tc</code> indicates that the <code>pci_a</code> -initiated DMA transfer is complete. When the <code>pci_a</code> sets the <code>dma_tc</code> bit, an interrupt will be generated on the <code>intan</code> output as long as interrupts are enabled by the <code>int_ena</code> bit (bit 0 of the <code>dma_csr</code>) and not disabled by the <code>tci_dis</code> bit (bit 5 of the <code>dma_csr</code>). The <code>dma_tc</code> bit is reset in one of three ways: a read transaction to the <code>dma_csr</code> ; a write transaction to the <code>dma_csr</code> , which sets the flush bit (bit 1 of the <code>dma_csr</code>); or by asserting the <code>rstn</code> input from the PCI bus.
4	<code>ad_loaded</code>	Read	When high, <code>ad_loaded</code> indicates that the address has been loaded via the <code>dma_acr</code> . This bit is cleared in one of three ways: when the DMA operation is complete and the <code>dma_tc</code> bit is set; when the flush bit is set; or when the <code>rstn</code> input is asserted from the PCI bus. The <code>ad_loaded</code> bit triggers the beginning of a DMA operation and is automatically set by the <code>pci_a</code> when a write operation to the <code>dma_csr</code> is performed. Therefore, the <code>dma_acr</code> should be written to last when a DMA operation is being loaded into the DMA registers.
31..5	Unused	–	–

DMA Transactions

The `pci_a` DMA engine, which consists of a 64-byte RAM buffer and four programmable registers, is the control channel when the `pci_a` acquires mastership of the PCI bus.

As a master device, the `pci_a` performs DMA read and write transactions to system memory (via the host bridge), or to another PCI bus agent capable of accepting burst target data transfers.

A DMA read transaction from memory to the local side consists of two separate transfers:

- A PCI bus burst read from the PCI bus to the RAM buffer
- An equivalent number of DWORD transfers to the local side

All DMA read transactions from the `pci_a` use the memory read command.

Similarly, a DMA write transaction from the `pci_a` to system memory consists of two separate transfers:

- One to sixteen DWORD transfers from the local side to the DMA buffer
- A PCI burst write from the RAM buffer to a PCI agent.

All DMA (PCI bus) write transactions from the `pci_a` use the memory write command.

PCI Bus DMA Read Transaction & Signal Sequence

In a PCI bus internal DMA read transaction, data is transferred from the system memory to the local side buffer. Specifically, a PCI bus DMA read consists of:

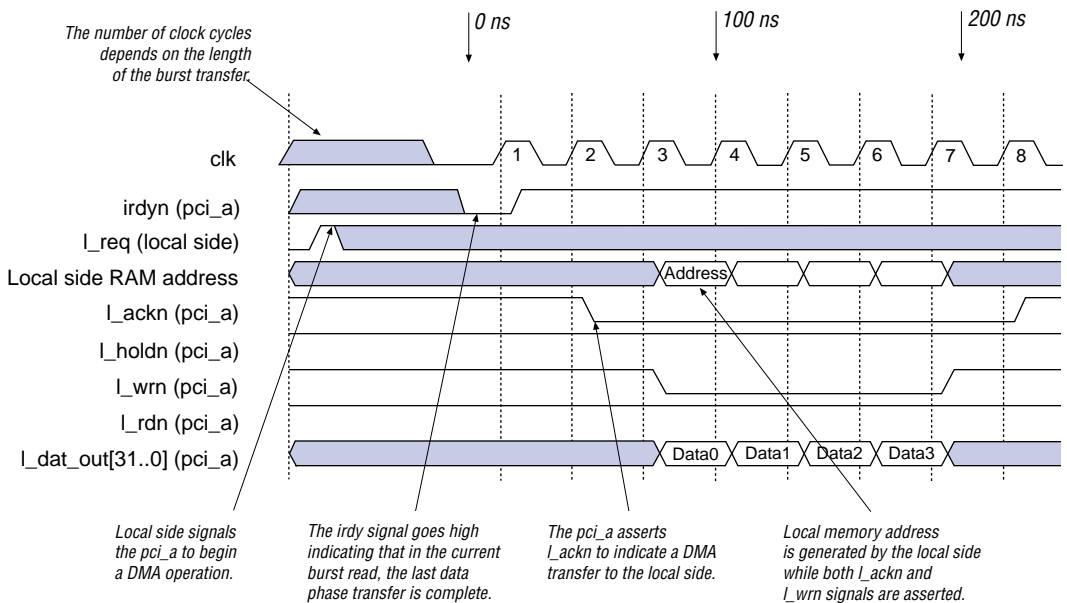
- A `pci_a` master device read from a PCI agent to the `pci_a` RAM buffer.
- A write from the `pci_a` RAM buffer to the local side peripheral device.

The following is the signal sequence of a PCI bus DMA read transaction:

1. The host sets up a DMA read transfer by writing appropriate values to the DMA registers. The DMA transfer sequence actually begins when the `dma_on` bit becomes set. Under normal conditions (i.e., DMA is enabled and no errors are pending) the `dma_on` bit becomes set when a write transaction to the DMA address counter register occurs.
2. The local side peripheral device asserts `l_req` to request a DMA transfer.
3. The `pci_a` asserts `reqn` and waits for `gntn` to become active before assuming mastership of the PCI bus.
4. The `pci_a` reads up to the 16 DWORDS from PCI bus system memory and loads the data into the `pci_a` RAM buffer.
5. Once the PCI transfer is complete, the `pci_a` asserts `l_ackn` and `l_wrn` to the local side peripheral device and transfers up to 16 DWORDS. Because the `pci_a` does not have the local side address location where data is to be written, the local side is responsible for generating the address during a local side DMA transfer. In [Figure 15](#) the address is not generated from the `pci_a`.

6. The pci_a writes the data from the pci_a RAM buffer onto the l_dat_out[31..0] bus. When the last data word is written, the pci_a disables l_ackn and l_wrn.
7. If the dma_bcr expiration occurs, the pci_a drives intan. Otherwise, steps 2 through 5 are repeated until dma_bcr expiration or until a DMA error occurs. See Figure 15.

Figure 15. PCI Bus DMA Read Transaction (Local Side DMA Write Transaction)



PCI Bus DMA Write Transaction & Signal Sequence

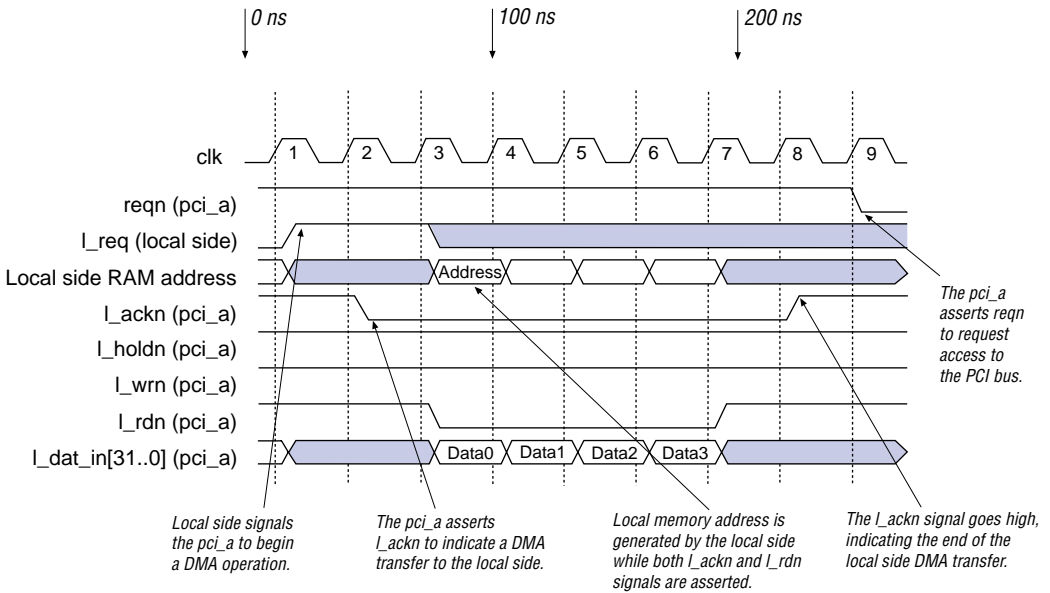
In a PCI bus internal DMA write transaction, data is transferred from the local side to system memory. Specifically, a PCI DMA write consists of:

- A transfer from the local side to the `pci_a` RAM buffer.
- A `pci_a` master write from the `pci_a` RAM buffer to a PCI bus agent.

The following is the signal sequence of a PCI DMA write transaction:

1. The host sets up a DMA write transfer by writing appropriate values to the DMA registers. The DMA transfer sequence actually begins when the `dma_on` bit becomes set. Under normal conditions (i.e., DMA is enabled and no errors are pending) the `dma_on` bit becomes set when a write transaction to the DMA address counter register occurs.
2. The local side peripheral device asserts `l_req` to request a DMA transfer.
3. The `pci_a` function asserts `l_ackn` and `l_rdn` in response to the DMA request and latches up to 16 DWORDS from the local side peripheral device.
4. The `pci_a` function reads the data from the `l_dat_in[31..0]` bus into the `pci_a` RAM buffer. When the last DWORD in the DMA transfer is read, or when the RAM buffer is full, the `pci_a` disables `l_ackn` and `l_rdn`.
5. The `pci_a` function asserts `reqn` and waits for `gntn` to become active before assuming mastership of the PCI bus.
6. The `pci_a` function transfers up to 16 DWORDS from its RAM buffer to the PCI bus target device.
7. If the `dma_bcr` expiration occurs (i.e., the specified number of data bytes have been transferred), the `pci_a` drives `intan`. Otherwise, steps 2 through 5 are repeated until the `dma_bcr` expiration or until a DMA error occurs. See [Figure 16](#).

Figure 16. PCI Bus DMA Write Transaction (Local Side DMA Read Transaction)



General Programming Guidelines

This section provides general programming guidelines and is divided into four tasks:

- Initializing the pci_a
- DMA operation
- Interrupt service operation
- Clearing error bits

Initializing the pci_a

1. Configure the pci_a-supported PCI bus configuration registers.
2. Configure the DMA dma_csr. See [Table 29](#).

Table 29. Initializing the pci_a

Step	Address (Hexadecimal)	Register Name	Data (Hexadecimal)	Definition
1	04	PCI bus command/status register	0000.0146	The value in the PCI bus command register enables memory transfers, master operations, the assertion of <code>perrn</code> in case of data parity errors, and the assertion of <code>serrn</code> in case of address parity errors.
2	BAR0: 0.0000	<code>dma_csr</code>	0000.0011	The value in the <code>dma_csr</code> enables both the interrupts and the DMA engine.

DMA Operation

3. Load the `dma_bcr`. (This step is optional if the byte count for the next block of data is the same as the current block.)
4. Load the `dma_acr`, (see [“Internal Target Registers Memory Map” on page 40](#)). Loading the `dma_acr` will set the `dma_on` bit, which enables the `pci_a` for the transfer to start as soon as the local side asserts `l_req`.
5. Configure the local side peripheral device. This step will set up the address generation process necessary on the local side and allow the local side to assert `l_req`. However, if an intelligent PCI agent is operating on the local side (i.e., a microprocessor), this step may not be necessary.
6. The `pci_a` function generates PCI interrupt (`intan`) to interrupt controller due to byte counter expiration. See [Table 30](#).

Table 30. DMA Operation

Step	Address (Hexadecimal)	Register Name	Data (Hexadecimal)	Definition
1	BAR0: 0.0008	dma_bcr	User-defined	The amount of data (in bytes) for a DMA transfer.
2	BAR0: 0.0004	dma_acr	User-defined	The PCI bus address where the transfer should begin. This address is automatically updated after every data transfer.
3	BAR0: 8.0000	External target register	User-defined	This may involve several steps, e.g., setting-up the local address generator; asserting <code>l_req</code> from the local side.

Interrupt Service Operation

7. Read the `dma_isr`.
 - a. If the `dma_tc` bit is high and `err_pend` bit is low, indicating that the DMA operation was successful and that the `pci_a` is ready for a new DMA transfer, go to step 3.
 - b. If the `err_pend` bit is high, indicating that the DMA operation was stopped due to an error, go to step 9. You should clear the error bit prior to continuing. See [Table 31](#).

Table 31. Interrupt Service Routine

Step	Address (Hexadecimal)	Register Name	Data (Hexadecimal)	Instructions
1	BAR0: 0.000C	dma_isr	User-defined	The value in this register indicates the progress of the DMA operation and the reason the operation was terminated.

Clearing Error Bits

8. Read the `dma_isr`, and if the `err_pend` bit is active, go to step 9.
9. Configure the `dma_csr` by asserting the `flush` bit to clear the `ad_loaded` bit (bit 4 of the `dma_isr`).

10. Read the PCI bus configuration status register and determine which error is asserted (i.e., bit 15, 12, or 13).
11. Configure the `pci_a`-supported PCI configuration status register and write a logic one to the appropriate error bit field. Writing a logic one to a bit in the status register will clear the bit; this feature allows the you to read the status register and write the same value to clear the error conditions.



For specific programming information, go to [Application Note 86 \(Implementing the `pci_a` Master/Target in FLEX 10K Devices\)](#).

Applications

The `pci_a` function is ideal for add-in applications. [Figure 17](#) shows a typical connection to an intelligent local-side host. In this example, a target and a DMA control block are needed for access to the local side. The local side data bus is a bidirectional bus controlled by the `l_holdn` output. The host asserts `l_holdn` whenever it is accessing the local bus. Because the PCI bus address is often different than the local side address, the host is responsible for generating the local side address during a DMA access.

Figure 17. Local Side Interface to an Intelligent Local-Side Host with a Shared Memory Bus

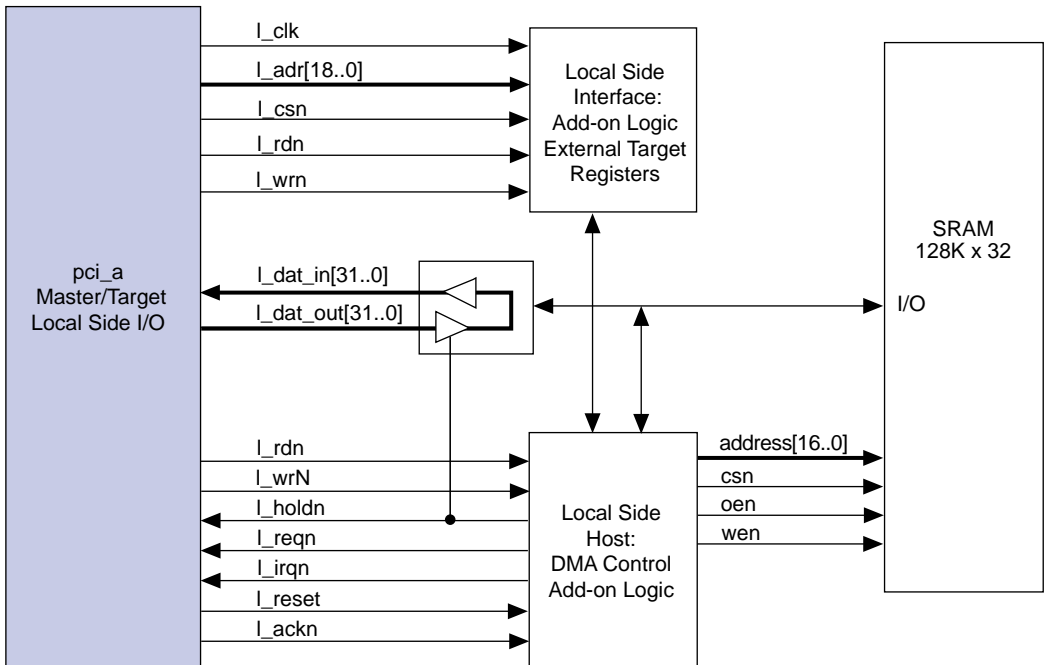
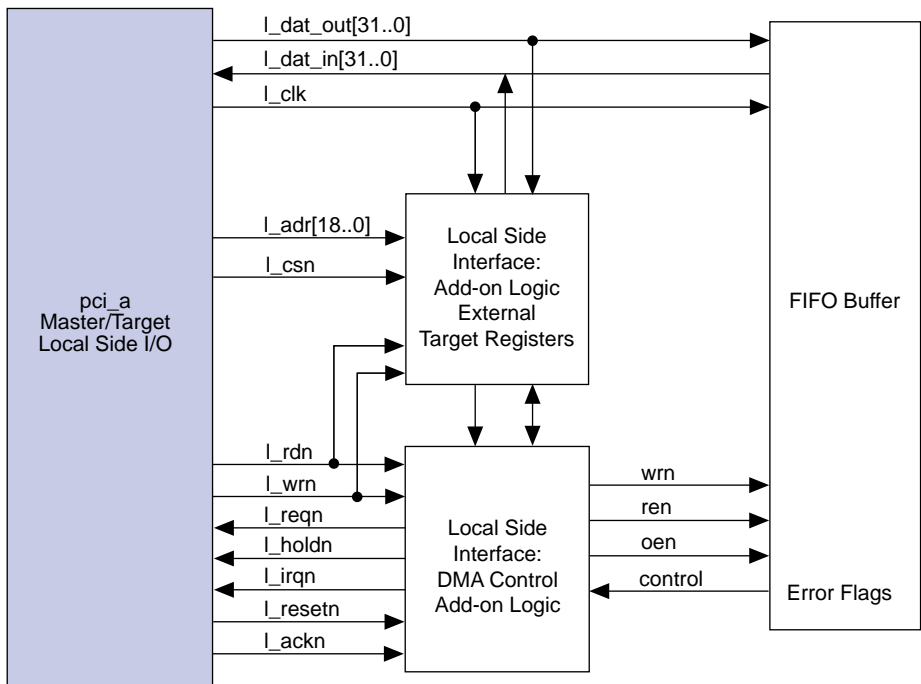


Figure 18 shows a typical `pci_a` connection to a dumb memory FIFO buffer. In this example, a target and a DMA control block are needed for access to the local side.

Because the local side does not have the intelligence to generate control and address signals during a DMA access, designers can set up the DMA control block to accept configuration and control data from the PCI bus via target access. Figure 18 illustrates the process via the bidirectional signals going between the two control blocks.

Figure 18. Local Side Interface to a Dumb FIFO buffer



PCI SIG Protocol Checklists

Tables 32 through 38 list the applicable PCI SIG protocol requirements from the *PCI Compliance Checklist, Rev. 2.1*. A check mark in the yes column indicates that the pci_a meets the requirement. Checklists not applicable to the Altera FLEX 10K pci_a function are not listed.

CO#	Requirement	Yes	No
1	Does each PCI resource have a configuration space based on the 256 byte template defined in section 6.1, with a predefined 64-byte header and a 192-byte device specific region?	✓	
2	Do all functions in the device support the vendor ID, device ID, command, status, header type and class code fields in the header?	✓	
3	Is the configuration space available for access at all times?	✓	
4	Are writes to reserved registers or read only bits completed normally and the data discarded?	✓	
5	Are reads to reserved or unimplemented registers, or bits, completed normally and a data value of 0 returned?	✓	
6	Is the vendor ID a number allocated by the PCI SIG?	✓	
7	Does the header type field have a valid encoding?	✓	
8	Do multi-byte transactions access the appropriate registers and are the registers in "little endian" order?	✓	
9	Are all read-only register values within legal ranges? For example, the interrupt pin register must only contain values 0-4.	✓	
10	Is the class code in compliance with the definition in appendix D?	✓	
11	Is the predefined header portion of configuration space accessible as bytes, words, and DWORDS?	✓	
12	Is the device a multi-function device?		✓

Table 33. Component Configuration Space Summary

Location	Name	Required/Optional	N/A	Support
00h-01h	Vendor ID	Required.		✓
02h-03h	Device ID	Required.		✓
04h-05h	Command	Required.		✓
06h-07h	Status	Required.		✓
08h	Revision ID	Required.		✓
09h-0Bh	Class code	Required.		✓
0Ch	Cache line size	Required by master devices/functions that can generate Memory Write and Invalidate.	✓	
0Dh	Latency timer	Required by master devices/functions that can burst more than two data phases.		✓
0Eh	Header type	If the device is multi-functional, then bit 7 must be set to a 1.		✓
0F	BIST	Optional.	✓	
10h-13h	BAR0	Optional.		✓
14h-27h	BAR1-BAR5	Optional.	✓	
28h-2Bh	Cardbus CIS pointer	Optional.	✓	
2Ch-2Dh	Subsystem vendor ID	Optional.		✓
2Eh-2Fh	Subsystem ID	Optional.		✓
30h-33h	Expansion ROM base address	Required for devices/functions that have expansion ROM.	✓	
34h-3Bh	Reserved			
3Ch	Interrupt line	Required by devices/functions that use an interrupt pin.		✓
3Dh	Interrupt pin	Required by devices/functions that use an interrupt pin.		✓
3Eh	Min_Gnt	Optional.		✓
3Fh	Max_Lat	Optional.	✓	

Table 34. Command Register Summary

Bit	Name	Required/Optional	N/A	Target	Master
0	I/O space	Required if device/function has registers mapped into I/O space.	✓		
1	Memory space	Required if device/function responds to memory space accesses.		✓	
2	Bus master	Required.			✓
3	Special cycles	Required for devices/functions that can respond to special cycles.	✓		
4	Memory write and invalidate	Required for devices/functions that generate Memory Write and Invalidate cycles.	✓		
5	VGA palette snoop	Required for VGA or graphical devices/functions that snoop VGA palette.	✓		
6	Parity error response	Required.			✓
7	Wait cycle control	Optional.	✓		
8	<code>serrn</code> enable	Required if device/function has <code>serrn</code> pin.			✓
9	Fast back-to-back enable	Required if master device/function can support fast back-to-back cycles among different targets.	✓		
10..15	Reserved				

Table 35. Device Status

DS#	Requirement	Yes	No
1	Do all implemented read/write bits in the status reset to 0?	✓	
2	Are read/write bits set to a 1 exclusively by the device/function?	✓	
3	Are read/write bits reset to a 0 when PCI <code>rstn</code> is asserted?	✓	
4	Are read/write bits reset to a 0 by writing a 1 to the bit?	✓	

Table 36. Status Register Summary

Bit	Name	Required/Optional	N/A	Target	Master
4..0	Reserved	Required.			
5	66-MHz capable	Required for 66-MHz capable devices.	✓		
6	UDF supported	Optional.	✓		
7	Fast back-to-back capable	Optional.	✓		
8	Data parity detected	Required.			✓
10..9	DEVSEL timing	Required.		✓	
11	Signaled target abort	Required for devices/functions that are capable of signaling target abort.	✓		
12	Received target abort	Required.			✓
13	Received master abort	Required.			✓
14	Signaled system error	Required for devices/functions that are capable of asserting <code>serrn</code> .			✓
15	Detected parity error	Required unless exempted per section 3.7.2.			✓

Table 37. Component Master Checklist (Part 1 of 3)

MP#	Requirement	Yes	No
1	All sustained tri-state signals are driven high for one clock before being tri-stated. (section 2.1)	✓	
2	Interface under test (IUT) always asserts all byte enables during each data phase of a memory write Invalidate cycle. (section 3.1.1)	✓	
3	IUT always uses linear burst ordering for memory write invalidate cycles. (section 3.1.1)	—	
4	IUT always drives <code>irdyn</code> when data is valid during a write transaction. (section 3.2.1)	✓	
5	IUT only transfers data when both <code>irdyn</code> and <code>trdyn</code> are asserted on the same rising clock edge. (section 3.2.1)	✓	
6	Once the IUT asserts <code>irdyn</code> it never changes <code>framen</code> until the current data phase completes. (section 3.2.1)	✓	
7	Once the IUT asserts <code>irdyn</code> it never changes <code>irdyn</code> until the current data phase completes. (section 3.2.1)	✓	
8	IUT never uses reserved burst ordering (<code>ad[1..0] = "01"</code>). (section 3.2.2)	✓	

Table 37. Component Master Checklist (Part 2 of 3)

MP#	Requirement	Yes	No
9	IUT never uses reserved burst ordering ($ad[1..0] = "11"$). (section 3.2.2)	✓	
10	IUT always ignores configuration command unless $idse1$ is asserted and $ad[1..0]$ are "00". (section 3.2.2)	✓	
11	The IUT's address lines are driven to stable values during every address and data phase. (section 3.2.4)	✓	
12	The IUT's $cbe[n][3..0]$ output buffers remain enabled from the first clock of the data phase through the end of the transaction. (section 3.3.1)	✓	
13	The IUT's $cbe[n][3..0]$ lines contain valid byte enable information during the entire data phase. (section 3.3.1)	✓	
14	IUT never deasserts $framen$ unless $irdyn$ is asserted or will be asserted (section 3.3.3.1)	✓	
15	IUT never deasserts $irdyn$ until at least one clock after $framen$ is deasserted. (section 3.3.3.1)	✓	
16	Once the IUT deasserts $framen$ it never reasserts $framen$ during the same transaction. (section 3.3.3.1)	✓	
17	IUT never terminates with master abort once target has asserted $devseln$.	✓	
18	IUT never signals master abort earlier than 5 clocks after $framen$ was first sampled asserted. (section 3.3.3.1)	✓	
19	IUT always repeats an access exactly as the original when terminated by retry. (section 3.3.3.2.2)	✓	
20	IUT never starts cycle unless $gntn$ is asserted. (section 3.4.1)	✓	
21	IUT always tri-states $cbe[n][3..0]$ and $ad[31..0]$ within one clock after $gntn$ negation when bus is idle and $framen$ is negated. (section 3.4.3)	✓	
22	IUT always drives $cbe[n][3..0]$ and $ad[31..0]$ within eight clocks of $gntn$ assertion when bus is idle. (section 3.4.3)	✓	
23	IUT always asserts $irdyn$ within eight clocks on all data phases. (section 3.5.2)	✓	
24	IUT always begins lock operation with a read transaction. (section 3.6)	—	
25	IUT always releases LOCK# when access is terminated by target-abort or master-abort. (section 3.6)	—	
26	IUT always deasserts LOCK# for minimum of one idle cycle between consecutive lock operations. (section 3.6)	—	
27	IUT always uses linear burst ordering for configuration cycles. (section 3.7.4)	✓	
28	IUT always drives par within one clock of $cbe[n][3..0]$ and $ad[31..0]$ being driven. (section 3.8.1)	✓	
29	IUT always drives par such that the number of "1"s on $ad[31..0]$, $cbe[n][3..0]$, and par equals an even number. (section 3.8.1)	✓	

Table 37. Component Master Checklist (Part 3 of 3)

MP#	Requirement	Yes	No
30	IUT always drives <code>perrn</code> (when enabled) active two clocks after data when data parity error is detected. (section 3.8.2.1)	✓	
31	IUT always drives PERR (when enabled) for a minimum of 1 clock for each data phase that a parity error is detected. (section 3.8.2.1)	✓	
32	IUT always holds <code>framen</code> asserted for cycle following DUAL command. (section 3.10.1)	—	
33	IUT never generates DUAL cycle when upper 32-bits of address are zero. (section 3.10.1)	—	

Table 38. Component Target Checklist (Part 1 of 2)

TP#	Requirement	Yes	No
1	All sustained tri-state signals are driven high for one clock before being tri-stated. (section 2.1)	✓	
2	IUT never reports <code>perrn</code> until it has claimed the cycle and completed a data phase. (section 2.2.5)	✓	
3	IUT never aliases reserved commands with other commands. (section 3.1.1)	—	
4	32-bit addressable IUT treats DUAL command as reserved. (section 3.1.1)	—	
5	Once IUT has asserted <code>trdyn</code> it never changes <code>trdyn</code> until the data phase completes. (section 3.2.1)	✓	
6	Once IUT has asserted <code>trdyn</code> it never changes <code>devseln</code> until the data phase completes. (section 3.2.1)	✓	
7	Once IUT has asserted <code>trdyn</code> it never changes <code>stopn</code> until the data phase completes. (section 3.2.1)	✓	
8	Once IUT has asserted <code>stopn</code> it never changes <code>stopn</code> until the data phase completes. (section 3.2.1)	✓	
9	Once IUT has asserted <code>stopn</code> it never changes <code>trdyn</code> until the data phase completes. (section 3.2.1)	✓	
10	Once IUT has asserted <code>stopn</code> it never changes <code>devseln</code> until the data phase completes. (section 3.2.1)	✓	
11	IUT only transfers data when both <code>irdyn</code> and <code>trdyn</code> are asserted on the same rising clock edge. (section 3.2.1)	✓	
12	IUT always asserts <code>trdyn</code> when data is valid on a read cycle. (section 3.2.1)	✓	
13	IUT always signals target-abort when unable to complete the entire I/O access as defined by the byte enables. (section 3.2.2)	—	
14	IUT never responds to reserved encodings. (section 3.2.2)	✓	
15	IUT always ignores configuration command unless <code>idse1</code> is asserted and <code>ad[31..0]</code> are "00". (section 3.2.2)	✓	

TP#	Requirement	Yes	No
16	IUT always disconnects after the first data phase when reserved burst mode is detected. (section 3.2.2)	—	
17	The IUT's <code>ad[31..0]</code> lines are driven to stable values during every address and data phase. (section 3.2.4)	✓	
18	The IUT's <code>cben[3..0]</code> output buffers remain enabled from the first clock of the data phase through the end of the transaction. (section 3.3.1)	✓	
19	IUT never asserts <code>trdyn</code> during turnaround cycle on a read. (section 3.3.1)	✓	
20	IUT always deasserts <code>trdyn</code> , <code>stopn</code> , and <code>devseln</code> the clock following the completion of the last data phase. (section 3.3.3.2)	✓	
21	IUT always signals disconnect when burst crosses resource boundary. (section 3.3.3.2)	—	
22	IUT always deasserts <code>stopn</code> the cycle immediately following <code>framen</code> being deasserted. (section 3.3.3.2.1)	✓	
23	Once the IUT has asserted <code>stopn</code> it never deasserts <code>stopn</code> until <code>framen</code> is negated. (section 3.3.3.2.1)	✓	
24	IUT always deasserts <code>trdyn</code> before signaling target-abort. (section 3.3.3.2.1)	—	
25	IUT never deasserts <code>stopn</code> and continues the transaction. (section 3.3.3.2.1)	✓	
26	IUT always completes initial data phase within 16 clocks. (section 3.5.1.1)	✓	
27	IUT always locks minimum of 16 bytes. (section 3.6)	—	
28	IUT always issues <code>devseln</code> before any other response. (section 3.7.1)	✓	
29	Once IUT has asserted <code>devseln</code> it never deasserts <code>devseln</code> until the last data phase has competed except to signal target-abort. (section 3.7.1)	✓	
30	IUT never responds to special cycles. (section 3.7.2)	✓	
31	IUT always drives <code>par</code> within one clock of <code>cben[3..0]</code> and <code>ad[31..0]</code> being driven. (section 3.8.1)	✓	
32	IUT always drives <code>par</code> such that the number of "1"s on <code>ad[31..0]</code> , <code>cben[3..0]</code> , and <code>par</code> equals an even number. (section 3.8.1)	✓	

PCI SIG Test Bench Summary

Tables 39 through 56 list the applicable PCI SIG test bench scenarios from the *PCI Compliance Checklist, Rev. 2.1*. A check mark in the yes column indicates that the pci_a meets the requirement. Checklists not applicable to the Altera FLEX 10K pci_a function are not listed.

Table 39. Test Scenario: 1.1 PCI Device Speed (as indicated by devsel) Tests

#	Requirement	Yes	No
1	Data transfer after write to fast memory slave.	✓	
2	Data transfer after read from fast memory slave.	✓	
3	Data transfer after write to medium memory slave.	✓	
4	Data transfer after read from medium memory slave.	✓	
5	Data transfer after write to slow memory slave.	✓	
6	Data transfer after read from slow memory slave.	✓	
7	Data transfer after write to subtractive memory slave.	✓	
8	Data transfer after read from subtractive memory slave.	✓	
9	Master abort bit set after write to slower than subtractive memory slave.	✓	
10	Master abort bit set after read from slower than subtractive memory slave.	✓	

Table 40. Test Scenario: 1.2 PCI Bus Target Abort Cycles (Part 1 of 2)

#	Requirement	Yes	No
1	Target abort bit set after write to fast memory slave.	✓	
2	IUT does not repeat the write transaction.	✓	
3	IUT's target abort bit set after read from fast memory slave.	✓	
4	IUT does not repeat the read transaction.	✓	
5	Target abort bit set after write to medium memory slave.	✓	
6	IUT does not repeat the write transaction.	✓	
7	IUT's target abort bit set after read from medium memory slave.	✓	
8	IUT does not repeat the read transaction.	✓	
9	Target abort bit set after write to slow memory slave.	✓	

Table 40. Test Scenario: 1.2 PCI Bus Target Abort Cycles (Part 2 of 2)			
#	Requirement	Yes	No
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.	✓	

Table 41. Test Scenario: 1.3 PCI Bus Target Retry Cycles			
#	Requirement	Yes	No
1	Data transfer after write to fast memory slave.	✓	
2	Data transfer after read from fast memory slave.	✓	
3	Data transfer after write to medium memory slave.	✓	
4	Data transfer after read from medium memory slave.	✓	
5	Data transfer after write to slow memory slave.	✓	
6	Data transfer after read from slow memory slave.	✓	
7	Data transfer after write to subtractive memory slave.	✓	
8	Data transfer after read from subtractive memory slave.	✓	

Table 42. Test Scenario: 1.4 PCI Bus Single Data Phase Retry Cycles (Part 1 of 2)			
#	Requirement	Yes	No
1	Data transfer after write to fast memory slave.	✓	
2	Data transfer after read from fast memory slave.	✓	
3	Data transfer after write to medium memory slave.	✓	
4	Data transfer after read from medium memory slave.	✓	
5	Data transfer after write to slow memory slave.	✓	

Table 42. Test Scenario: 1.4 PCI Bus Single Data Phase Retry Cycles (Part 2 of 2)

#	Requirement	Yes	No
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.	✓	

Table 43. Test Scenario: 1.5 PCI Bus Single Data Phase Disconnect Cycles

#	Requirement	Yes	No
1	Target abort bit set after write to fast memory slave.	✓	
2	IUT does not repeat the write transaction.	✓	
3	IUT's target abort bit set after read from fast memory slave.	✓	
4	IUT does not repeat the read transaction.	✓	
5	Target abort bit set after write to medium memory slave.	✓	
6	IUT does not repeat the write transaction.	✓	
7	IUT's target abort bit set after read from medium memory slave.	✓	
8	IUT does not repeat the read transaction.	✓	
9	Target abort bit set after write to slow memory slave.	✓	
10	IUT does not repeat the write transaction.	✓	
11	IUT's target abort bit set after read from slow memory slave.	✓	
12	IUT does not repeat the read transaction.	✓	
13	Target abort bit set after write to subtractive memory slave.	✓	
14	IUT does not repeat the write transaction.	✓	
15	IUT's target abort bit set after read from subtractive memory slave.	✓	
16	IUT does not repeat the read transaction.	✓	

Table 44. Test Scenario: 1.6 PCI Bus Multi-Data Phase Retry Cycles			
#	Requirement	Yes	No
1	Data transfer after write to fast memory slave.	✓	
2	Data transfer after read from fast memory slave.	✓	
3	Data transfer after write to medium memory slave.	✓	
4	Data transfer after read from medium memory slave.	✓	
5	Data transfer after write to slow memory slave.	✓	
6	Data transfer after read from slow memory slave.	✓	
7	Data transfer after write to subtractive memory slave.	✓	
8	Data transfer after read from subtractive memory slave.	✓	

Table 45. Test Scenario: 1.7 PCI Bus Multi-Data Phase Disconnect Cycles			
#	Requirement	Yes	No
1	Data transfer after write to fast memory slave.	✓	
2	Data transfer after read from fast memory slave.	✓	
3	Data transfer after write to medium memory slave.	✓	
4	Data transfer after read from medium memory slave.	✓	
5	Data transfer after write to slow memory slave.	✓	
6	Data transfer after read from slow memory slave.	✓	
7	Data transfer after write to subtractive memory slave.	✓	
8	Data transfer after read from subtractive memory slave.	✓	

Table 46. Test Scenario: 1.8 PCI Bus Multi-Data Phase & trdyn Cycles (Part 1 of 2)			
#	Requirement	Yes	No
1	Verify that data is written to primary target when <code>trdynis</code> released after second rising clock edge and asserted on third rising clock edge after <code>framen</code> .	✓	
2	Verify that data is read from primary target when <code>trdyn</code> is released after second rising clock edge and asserted on third rising clock edge after <code>framen</code> .	✓	
3	Verify that data is written to primary target when <code>trdyn</code> is released after third rising clock edge and asserted on fourth rising clock edge after <code>framen</code> .	✓	

Table 46. Test Scenario: 1.8 PCI Bus Multi-Data Phase & trdyn Cycles (Part 2 of 2)

#	Requirement	Yes	No
4	Verify that data is read from primary target when <code>trdyn</code> is released after third rising clock edge and asserted on fourth rising clock edge after <code>framen</code> .	✓	
5	Verify that data is written to primary target when <code>trdyn</code> is released after third rising clock edge and asserted on fifth rising clock edge after <code>framen</code> .	✓	
6	Verify that data is read from primary target when <code>trdyn</code> is released after third rising clock edge and asserted on fifth rising clock edge after <code>framen</code> .	✓	
7	Verify that data is written to primary target when <code>trdyn</code> is released after fourth rising clock edge and asserted on sixth rising clock edge after <code>framen</code> .	✓	
8	Verify that data is read from primary target when <code>trdyn</code> is released after fourth rising clock edge and asserted on sixth rising clock edge after <code>framen</code> .	✓	
9	Verify that data is written to primary target when <code>trdyn</code> alternately released for one clock cycle and asserted for one clock cycle after <code>framen</code> .	✓	
10	Verify that data is read from primary target when <code>trdyn</code> alternately released for one clock cycle and asserted for one clock cycle after <code>framen</code> .	✓	
11	Verify that data is written to primary target when <code>trdyn</code> alternately released for two clock cycles and asserted for two clock cycles after <code>framen</code> .	✓	
12	Verify that data is read from primary target when <code>trdyn</code> alternately released for two clock cycles and asserted for two clock cycles after <code>framen</code> .	✓	

Table 47. Test Scenario: 1.9 PCI Bus Data Parity Error Single Cycles

#	Requirement	Yes	No
1	Verify the IUT sets data parity error detected bit when primary target asserts <code>perrn</code> on IUT memory write.	✓	
2	Verify that <code>perrn</code> 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.	✓	

Table 48. Test Scenario: 1.10 PCI Bus Data Parity Error Multi-Data Phase Cycles

#	Requirement	Yes	No
1	Verify the IUT sets parity error detected bit when primary target asserts <code>perrn</code> on IUT multi-data phase memory write.	✓	
2	Verify that <code>perrn</code> 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.	✓	

Table 49. Test Scenario: 1.11 PCI Bus Master Time-Out			
#	Requirement	Yes	No
1	Memory write transaction terminates before 4 data phases completed.	✓	
2	Memory read transaction terminates before 4 data phases completed.	✓	

Table 50. Test Scenario: 1.13 PCI Bus Master Parking			
#	Requirement	Yes	No
1	IUT drives <code>ad[31..0]</code> to stable values within eight PCI clocks of <code>gntn</code> .	✓	
2	IUT drives <code>cben[3..0]</code> to stable values within eight PCI clocks of <code>gntn</code> .	✓	
3	IUT drives <code>par</code> one clock cycle after IUT drives <code>ad[31..0]</code>	✓	
4	IUT tri-states <code>ad[31..0]</code> and <code>cben[3..0]</code> and <code>par</code> when <code>gntn</code> is released.	✓	

Table 51. Test Scenario: 1.14 PCI Bus Master Arbitration			
#	Requirement	Yes	No
1	IUT completes transaction when deasserting <code>gntn</code> is coincident with asserting <code>framen</code> .	✓	

Table 52. Test Scenario: 2.5 Target Ignores Reserved Commands			
#	Requirement	Yes	No
1	IUT does not respond to RESERVED COMMANDS.	✓	
2	Initiator detects master abort for each transfer.	✓	
3	IUT does not respond to 64-bit cycle (dual address).	✓	

Table 53. Test Scenario: 2.6 Target Receives Configuration Cycles			
#	Requirement	Yes	No
1	IUT responds to all configuration cycles type 0 read/write cycles appropriately.	✓	
2	IUT does not respond to configuration cycles type 0 with <code>idsel</code> inactive.	✓	

Table 54. Test Scenario: 2.8 Target Receives Configuration Cycles with Address and Data Parity Errors

#	Requirement	Yes	No
1	IUT reports address parity error via <code>serrn</code> during configuration read/write cycles.	✓	
2	IUT reports data parity error via PERR during configuration write cycles.	✓	

Table 55. Test Scenario: 2.9 Target Receives Memory Cycles

#	Requirement	Yes	No
1	IUT completes single memory read and write cycles appropriately.	✓	

Table 56. Test Scenario: 2.10 Target Receives Memory Cycles with Address and Data Parity Errors

#	Requirement	Yes	No
1	IUT reports address parity error via <code>serrn</code> during all memory read and write cycles.	✓	
2	IUT reports data parity error via PERR during all memory write cycles.	✓	

References

Reference documents for the `pci_a` include:

- *PCI Local Bus Specification. Rev. 2.1.* Portland, Oregon: PCI Special Interest Group, June 1995
- *PCI Compliance Checklist. Rev 2.1.* Portland, Oregon: review copy.
- *Altera 1996 Data Book.* San Jose, California. June 1996.
- *IEEE Standard VHDL Language Reference Manual.* ANSI/IEEE Std 1076-1993. New York, New York. June 1994.



2610 Orchard Parkway
San Jose, CA 95134-2020
(408) 544-7000

<http://www.altera.com>

Applications Hotline:

(800) 800-EPLD

Customer Marketing:

(408) 894-7104

Literature Services:

(408) 894-7144

lit_req@altera.com

Altera, FLEX, FLEX 10K, EPF10K130V, AHDL, MegaCore, OpenCore, MAX, MAX+PLUS, and MAX+PLUS II are trademarks and/or service marks of Altera Corporation in the United States and other countries. Altera acknowledges the trademarks of other organizations for their respective products or services mentioned in this document. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

Copyright © 1997 Altera Corporation. All rights reserved.



I.S. EN ISO 9001