# PCI Master/Target MegaCore Function with DMA

November 1999, ver. 3.02 Data Sheet

## Features

- pci_a MegaCore™ function implementing a 32-bit peripheral component interconnect (PCI) master/target interface
- Optimized for the FLEX® 10K architecture
- Extensive hardware testing using:
  - HP E2925A PCI Bus Exerciser and Analyzer
  - FLEX 10K PCI prototype board
  - Validated against common PCI chipsets such as: Intel 430 and 440 chipsets, and DEC PCI-to-PCI bridges
- Dramatically shortens design cycles
- FLEX 10K PCI prototype board included
- Includes test vectors for user simulation
- OpenCore™ feature allows designers to instantiate and simulate designs in the MAX+PLUS® II software prior to licensing
- Uses approximately 1,000 FLEX logic elements (LEs), e.g., 35% the capacity of an EPF10K50 device
- PCI master features:
  - Memory read/write
  - Bus parking
  - Fully integrated DMA engine including address counter register, byte counter register, control and status register, and interrupt status register
  - Configurable interrupt source, including DMA terminal count, master abort, target abort, and 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 and write burst transactions
- PCI target features:
  - Type zero configuration space
  - Parity error detection
  - Memory read/write and configuration read/write
  - Target retry and disconnect
  - 1 Mbyte to 2 Gbytes of parameterized target memory space
- Configuration registers:
  - Parameterized: device ID, vendor ID, class code, revision ID, base address zero, subsystem ID, subsystem vendor ID
  - Non-parameterized: command, status, header type, latency timer, interrupt pin, interrupt line

# Introduction

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

# New in Version 2.0

The pci_a function version 2.0 includes the following enhancements:

- Additional device support
- Local-side initiated DMA
- Parameterized base address registers (BARs)
- Byte-wide selection during external target write transfers
- Use of l_holdn during external target transactions
- Larger DMA byte counter register

## More Device Support

The pci_a function supports a wide range of devices and packages including the following FLEX 10K devices:

- EPF10K30RC240
- EPF10K30RC208
- EPF10K30AQC240
- EPF10K30AQC208
- EPF10K40RC240
- EPF10K40RC208
- EPF10K50RC240
- EPF10K100ARC240
- EPF10K30BC356
- EPF10K50BC356
- EPF10K100ABC356

Additional device support will become available as new devices are released. Please check the Altera world-wide web site at **http://www.altera.com** for latest device support.

## Local-Side Initiated DMA

To perform a DMA burst transfer using the pci_a function, appropriate values must be written to the DMA registers to setup the transfer. In prior versions of the pci_a function, the host or a PCI master device was required to write to DMA registers. However, pci_a version 2.0 also allows DMA read and write transactions directly from the local side device. See "Initializing DMA Transfers from the Local Side" on page 50 for more information.

### Parameterized BARs

The BAR0 is parameterized to provide optimum efficiency for memory allocation.In pci_a version 1.3, the BAR0 address space is a constant 1 Mbyte of contiguous address space divided into two 512 Kbytes of memory space. However, in pci_a version 2.0 and later, users can vary the BAR0 address space from 1 Mbyte to 2 Gbytes of contiguous memory. See "Base Address Register Zero (Offset = 10 Hex)" on page 24 for more information.

### Byte-Wide Selection during Target Write Transfers

During target transfers, the PCI cben[3..0] bus signals are byte enable signals, indicating which byte carries meaningful data. Bit 3 of the cben[3..0] bus applies to byte 3, and bit 0 applies to byte 0. Likewise in pci_a version 2.0, the additional local-side l_ben[3..0] bus signals buffer the cben[3..0] bus signals and inform the local-side logic which byte carries meaningful data during external target write transactions.

### l_holdn for External Target Write Transactions

In pci_a version 1.3 the local application is required to supply or accept data within two clock cycles. In version 2.0, a slower application can assert l_holdn to extend the period necessary to transfer the data.

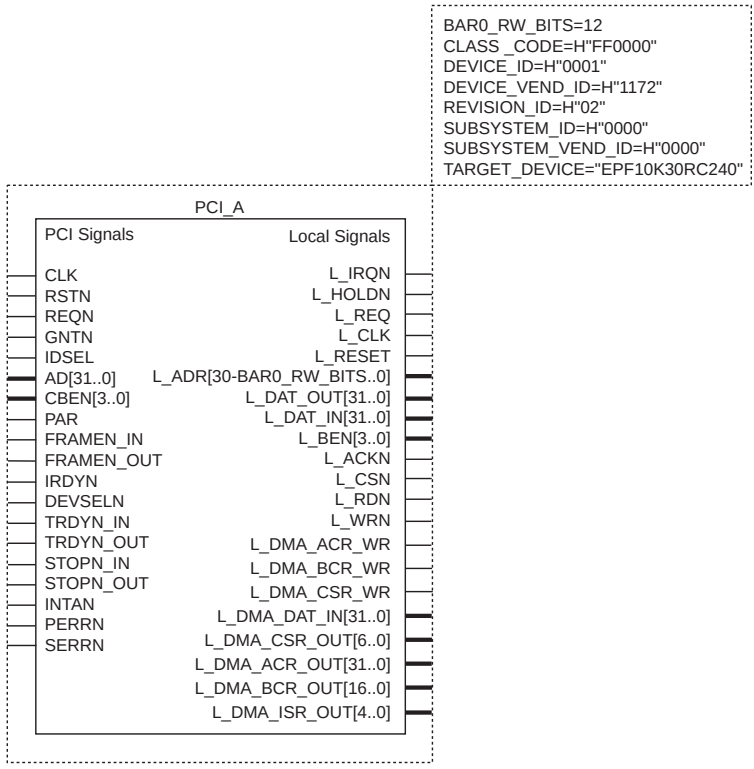
### Larger DMA Byte Counter Register

The DMA byte counter register was increased from 16 bits to 17 bits. As a result, the master DMA engine may initiate memory transfers up to 128 Kbytes for each DMA transaction.

## 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 function is optimized for the FLEX 10K device family, reducing the design task and enabling designers to focus efforts on the custom logic surrounding the PCI interface (ordering code: PLSM-PCI/A). Figure 1 shows the pci_a symbol.

***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, Revision 2.1,*** and ***Compliance Checklist, Revision 2.1.*** The pci_a function has successfully completed extensive hardware validation testing to ensure robustness and PCI bus compliance. The testing was performed using the following hardware and software:

- Altera FLEX 10K PCI prototype board
- BlueWater Systems WinDK (Windows NT-based) software driver
- HP E2925A PCI Bus Exerciser and Analyzer

The testing was performed in a fully-loaded PCI bus. In addition to the HP E2925A PCI Bus Exerciser and Analyzer and the Altera PCI prototype board, PCI bus agents such as the host bridge, Ethernet network adapter, and video card tested the function using data-intensive applications. The extensive testing ensures that the pci_a function operates flawlessly under the most stringent conditions.

The pci_a function performs master and target transactions to and from the Altera PCI prototype board. Along with typical burst and single-cycle transactions, the pci_a function runs various interrupt cycles and initiates different abnormal terminations. In addition to checking for data integrity, the HP E2925A PCI Bus Exerciser and Analyzer was used to ensure that the PCI bus is free of protocol violation. Each iteration of the test program transfers over 6.5 billion data bytes between the host memory and the pci_a-based EPF10K30 device. The test procedure was done overnight, thus accounting for hundreds of iterations. The tests were repeated across multiple PCI platforms to ensure compatibility with various chipsets. Table 1 shows a list of hardware platforms with which the pci_a function was tested at the time of this document printing.

*Table 1. pci_a Hardware Verified Platforms*

| Platform | Chipset | CPU Speed (MHz) | PCI Bus Speed (MHz) |
|---|---|---|---|
| Dell OptiPlex XM 5166 | Intel 430 NX | 166 | 33 |
| Dell OptiPlex GX Pro | Intel 440FX PCISet (Bus 0) | 200 | 33 |
| | DEC21052-AB PCI-PCI bridge (Bus 1) | 200 | 33 |
| Dell OptiPlex GXL 5166 | Intel 430 FX PCISet | 166 | 33 |
| U-tron (Pentium/MMX) | Intel 430 VX PCISet | 166 | 33 |

In addition to all the hardware testing, the pci_a function was verified using the applicable scenarios listed in Table 2. For a detailed listing of tests performed, see "PCI SIG Test Bench Summary" on page 66.

*Table 2. PCI Bus Tests Performed on the pci_a Function (Part 1 of 2)*

| PCI Test Scenario Number | Test Scenario Description | Simulation File Name *Note (1)* |
|---|---|---|
| 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 & trdyn 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 |

*Table 2. PCI Bus Tests Performed on the pci_a Function (Part 2 of 2)*

| PCI Test Scenario Number | Test Scenario Description | Simulation File Name, *Note (1)* |
|---|---|---|
| 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 (2)* | Programming the DMA registers and burst read transfers. | dma_rd |
| *Note (2)* | Programming the DMA registers and burst write transfers. | dma_wr |
| *Note (2)* | External target read/write transfers | trg_xrw |

*Note:*

(1) The file extension depends on the type of simulation file used, e.g., Simulator Channel File (**.scf**), Vector File (**.vec**), or VHDL file.

(2) This test is not required by the PCI SIG ***PCI Local Bus Specification, Revision 2.1***, and therefore does not have a test number.

## 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 3 summarizes the PCI bus signals interfacing the pci_a function to the PCI bus. See "Local Side Signals" on page 10 for information on local side signals.

*Table 3. PCI Signals Interfacing the pci_a to the PCI Bus (Part 1 of 2)*

| Name | Type | Polarity | Description |
|---|---|---|---|
| clk | Input | – | Clock. The clk input provides the reference signal for all other PCI interface signals, except rstn and intan. |
| rstn | Input | Low | Reset. The rstn input initializes the FLEX 10K PCI interface circuitry, and can be asserted asynchronously to the PCI bus clk edge. When active, the PCI output signals are tri-stated and the open-drain signals, such as serrn, float. |
| gntn | Input | Low | Grant. The gntn input indicates to the master device that it has control of the PCI bus. Every master device has a pair of arbitration lines (gntn and reqn) that connect directly to the arbiter. |
| reqn | Output | Low | Request. The reqn output indicates to the arbiter that the master wants to gain control of the PCI bus to perform a transaction. |
| ad[31..0] | Tri-State | – | Address/data bus. The ad[31..0] bus is a time-multiplexed address/data bus; each bus transaction consists of an address phase followed by one or more data phases. Each data phase completes when irdyn and trdyn are both asserted. |
| cben[3..0] | Tri-State<br>Master: Output<br>Target: Input | Low | Command/byte enable. The cben[3..0] bus is a time-multiplexed command/byte enable bus. During the address phase this bus indicates the command; during the data phase this bus indicates byte enables. |
| par | Tri-State | – | Parity. The par signal is a tri-stated output of even parity. The number of 1s on ad[31..0], cben[3..0], and par is an even number. |
| framen<br>*Note (1)* | Sustained<br>Tri-State<br>Master: Output<br>Target: Input | Low | Frame. The framen is an output from the current bus master that indicates the beginning and duration of a bus operation. When framen is initially asserted, the address and command signals are present on the ad[31..0] and cben[3..0] buses. The framen signal remains asserted during the data operation and is deasserted to identify the end of a transaction. |
| irdyn | Sustained<br>Tri-State<br>Master: Output<br>Target: Input | Low | Initiator ready. The irdyn 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, irdyn indicates that valid data is on the ad[31..0] bus. In a read transaction, irdyn indicates that the master is ready to accept the data on the ad[31..0] bus. |
| devseln | Sustained<br>Tri-State<br>Master: Input<br>Target: Output | Low | Device select. Target asserts devseln to indicate that the target has decoded its own address. |

*Table 3. PCI Signals Interfacing the pci_a to the PCI Bus (Part 2 of 2)*

| Name | Type | Polarity | Description |
|---|---|---|---|
| trdyn *Note (1)* | Sustained Tri-State Master: Input Target: Output | Low | Target ready. The trdyn signal indicates that the target can complete the current data transaction. In a read operation, trdyn indicates that the target is providing data on the ad[31..0] bus. In a write operation, trdyn indicates that the target is ready to accept data on the ad[31..0] bus. |
| stopn *Note (1)* | Sustained Tri-State Master: Input Target: Output | Low | Stop. The stopn signal is a target device request that indicates to the bus master to stop the current transaction. |
| idsel | Input | High | Initialization device select. The idsel input is a chip select for configuration read or write operations. |
| perrn | Sustained Tri-State | Low | Parity error. The perrn signal indicates a data parity error. |
| serrn | Open-Drain | Low | System error. The serrn signal indicates system and address parity errors. |
| intan | Open-Drain | Low | Interrupt A. The intan 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 function 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 IEEE Std. 1149.1 Joint Test Action Group (JTAG) boundary-scan testing (BST). To use IEEE Std. 1149.1 BST, designers should connect the PCI bus JTAG pins with the FLEX 10K device JTAG pins. See Table 4.

*Table 4. Optional IEEE Std. 1149.1 Signals*

| 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 5 summarizes the pci_a function signals that interface the pci_a function to the local side peripheral device(s).

*Table 5. pci_a Signals Interfacing the pci_a Function to the Local Side (Part 1 of 3)*

| Name | Type | Polarity | Description |
|---|---|---|---|
| l_irqn | Input | Low | Local side interrupt request. The local side peripheral device asserts l_irqn to signal a PCI bus interrupt. For example, when the local side peripheral device requires a DMA transfer, it could use the l_irqn input to request servicing from the host. |
| l_holdn | Input | Low | Local hold. During master transactions, l_holdn suspends the current DMA transfer. As long as l_holdn is active, data transfers cannot occur between the pci_a function and the local side peripheral device. During target transactions, the assertion of l_holdn extends the external target transfers. If l_holdn is not asserted, the pci_a function expects data to be supplied to or received from the local side on the second clock after l_csn is asserted. |
| l_req | Input | High | Local DMA request. After the DMA has been loaded with valid data, the local side peripheral device asserts l_req, which signals the pci_a function to start the PCI DMA operation. |
| l_dat_in[31..0] | Input | – | Local data bus input. The l_dat_in[31..0] input is driven active by the local side peripheral device during pci_a-initiated DMA write transactions (i.e., local side DMA read transactions) and PCI bus target read transactions. |
| l_dat_out[31..0] | Output | _ | Local data bus output. The pci_a function 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. |
| l_ben[3..0] | Output | Low | Local byte enable. The l_ben[3..0] outputs are driven by the pci_a function to indicate the byte select during target write transfers. |
| l_adr[30-BAR0_RW_BITS..0] | Output | – | Local target address. The l_adr[30-BAR0_RW_BITS..0] outputs represent address of the target transaction 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. The l_ackn and the l_csn outputs are never asserted at the same time. |

*Table 5. pci_a Signals Interfacing the pci_a Function to the Local Side (Part 2 of 3)*

| Name | Type | Polarity | Description |
|---|---|---|---|
| l_rdn | Output | Low | Read. The pci_a function asserts l_rdn to signal a read access to the local side peripheral device. The pci_a function uses the l_rdn for reading from peripheral device target registers and for PCI DMA write transactions. For target read operations, the pci_a function asserts the l_csn and l_rdn signals. For DMA write operations, the pci_a function asserts the l_ackn and l_rdn signals. |
| l_wrn | Output | Low | Write. The pci_a function asserts l_wrn to signal a write access to the local side peripheral device. The pci_a function uses the l_wrn output for writing to peripheral device target registers and for PCI DMA read transactions. For a write operation to the local side, pci_a asserts either l_csn and l_wrn for target accesses, or l_ackn and l_wrn for DMA read accesses. |
| 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 function. |
| 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 function. |
| l_reset | Output | High | Local reset. The pci_a function 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_dma_acr_wr | Input | High | Local DMA address counter register write. The local side asserts l_dma_acr_wr to signal a write access to the DMA address counter register. When l_dma_acr_wr is high, the data on l_dma_in[31..0] bus is written into the dma_acr register. |
| l_dma_bcr_wr | Input | High | Local DMA byte counter register write. The local side asserts l_dma_bcr_wr to signal a write access to the DMA byte counter register. When l_dma_bcr_wr is high, the data on l_dma_dat_in[31..0] bus is written into the dma_bcr register. |

*Table 5. pci_a Signals Interfacing the pci_a Function to the Local Side (Part 3 of 3)*

| Name | Type | Polarity | Description |
|---|---|---|---|
| l_dma_csr_wr | Input | High | Local DMA control status register write. The local side asserts l_dma_csr_wr to signal a write access to the DMA control/status registers. When l_dma_csr_wr is high, the data on l_dma_dat_in[31..0] bus is written into the dma_csr register. |
| l_dma_dat_in[31..0] | Input | – | Local DMA data in. While one of the DMA write signals (l_dma_acr_wr, l_dma_bcr_wr, or l_dma_csr_wr) is asserted, the l_dma_dat_in[31..0] supplies the data to be written to the corresponding DMA register. |
| l_dma_csr_out[6..0] | Output | – | Local DMA control status registers out. Direct output of the DMA control/status register. |
| l_dma_acr_out[31..0] | Output | _ | Local DMA address counter registers out. Direct output of DMA the address counter registers. |
| l_dma_bcr_out[16..0] | Output | _ | Local DMA byte counter registers out. Direct output of the DMA byte counter register. |
| l_dma_isr_out[4..0] | Output | _ | Local DMA interrupt status registers out. Direct output of the DMA interrupt status register. |

## Function Prototype

The Altera Hardware Description Language (AHDL) Function Prototype of the pci_a function 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, l_dma_acr_wr, l_dma_bcr_wr,
    l-dma_csr_wr, l_dma_dat_in[31..0])

    WITH (SUBSYSTEM_ID, SUBSYSTEM_VEND_ID, DEVICE_ID,
    DEVICE_VEND_ID, CLASS_CODE, REVISION_ID, BAR0_RW_BITS,
    TARGET_DEVICE)

    RETURNS (framen_out, l_ackn,
    l_adr[30-BAR0_RW_BITS..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; l_dma_csr_out[6..0],
    l_dma_acr_out[31..0], l_dma_bcr[16..0],
    l_dma_isr_out[4..0], l_ben[3..0]);
```

## Parameters

The pci_a parameters—except BAR0_RW_BITS and TARGET_DEVICE—set read-only PCI bus configuration registers in the pci_a function; these registers are called device identification registers. See "Configuration Registers" on page 18 for more information on device ID registers.

The BAR0_RW_BITS parameter controls the number of read/write bits instantiated for BAR0, and according to the PCI specification, the number of read/write bits instantiated for BAR0 controls the memory address range reserved by the BAR0. The value of the BAR0_RW_BITS parameter must be between 1 and 12. The TARGET_DEVICE parameter ensures that the most optimized design is used for a particular device and package, which ensures timing compliance of the target device. For the most updated list of support devices and packages, refer to the **readme.htm** file included with the pci_a function. Table 6 describes the parameters of the pci_a function.

*Table 6. Parameters*

| Name | Format | Default Value | Description |
|---|---|---|---|
| BAR0_RW_BITS | Decimal | 12 | BAR address space size |
| TARGET_DEVICE | String | "EPF10K30RC240" | Device selection |
| CLASS_CODE | 24-bit Hex | H"FF0000" | Class code register |
| DEVICE_ID | 16-bit Hex | H"0001" | Device ID register |
| DEVICE_VEND_ID | 16-bit Hex | H"1172" | Device vendor ID register |
| REVISION_ID | 8-bit Hex | H"02" | Revision ID register |
| SUBSYSTEM_ID | 16-bit Hex | H"0000" | Subsystem ID register |
| SUBSYSTEM_VEND_ID | 16-bit Hex | H"0000" | Subsystem vendor ID register |

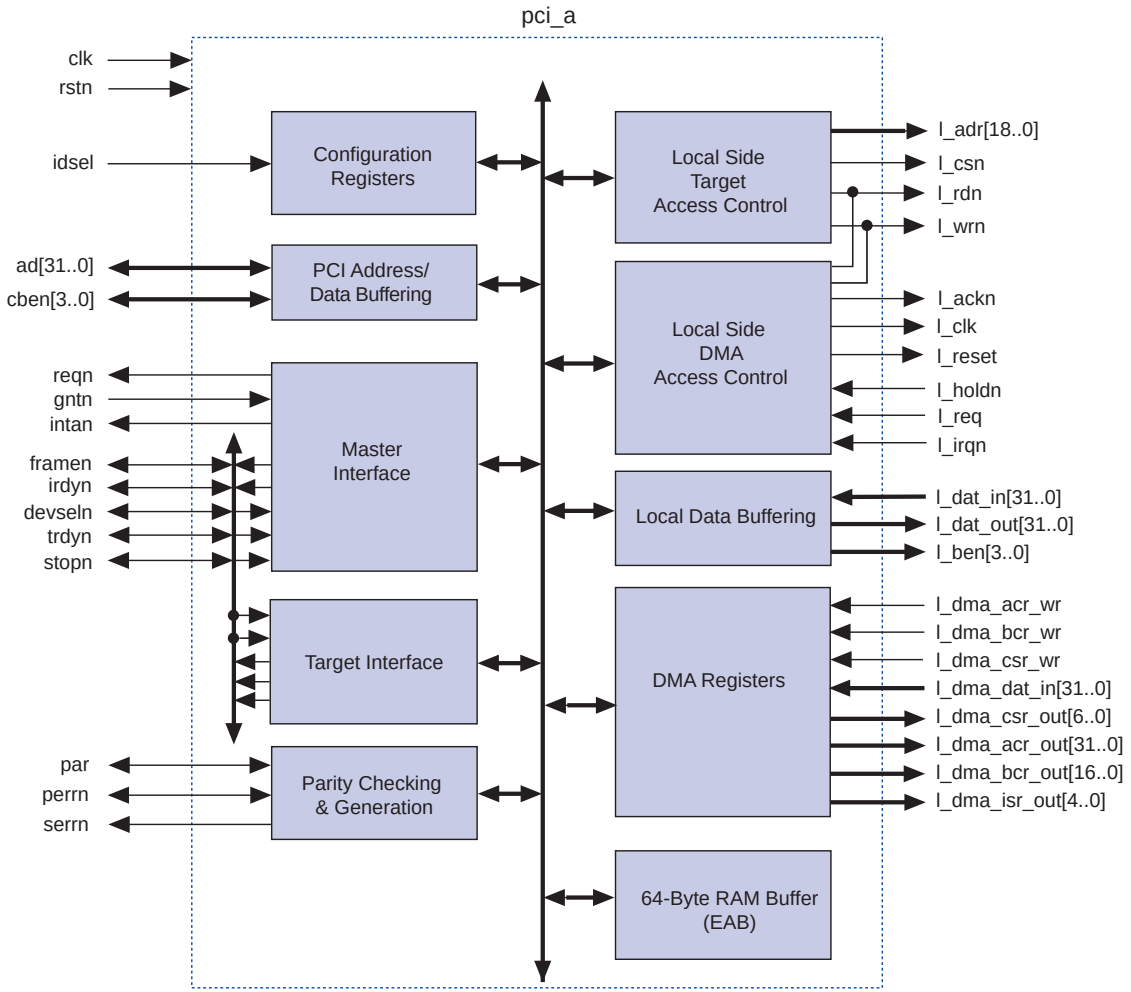
# Functional Description

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 function's block diagram.

***Figure 2. pci_a Function 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 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

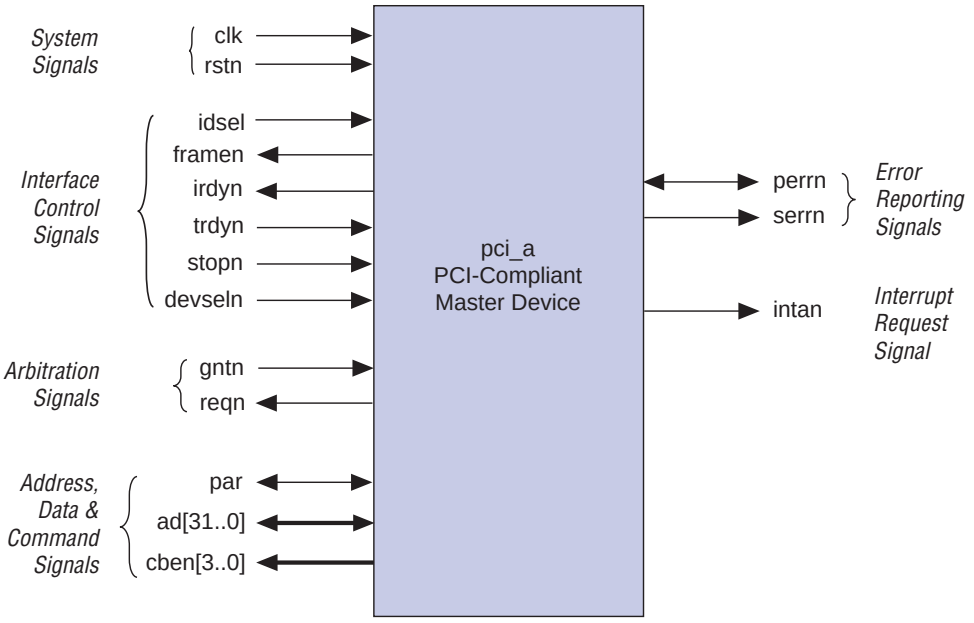
Figure 3 illustrates the PCI-compliant master device signals interfacing pci_a with the PCI bus. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the pci_a function operating as a master on the PCI bus.

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 function 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 master is ready to present data on the bus, it asserts irdyn. At this point, the pci_a function's 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 DWORD 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 function with the PCI bus. The signals are grouped by functionality, and signal directions are illustrated from the perspective of the pci_a function operating as a target on the PCI bus.

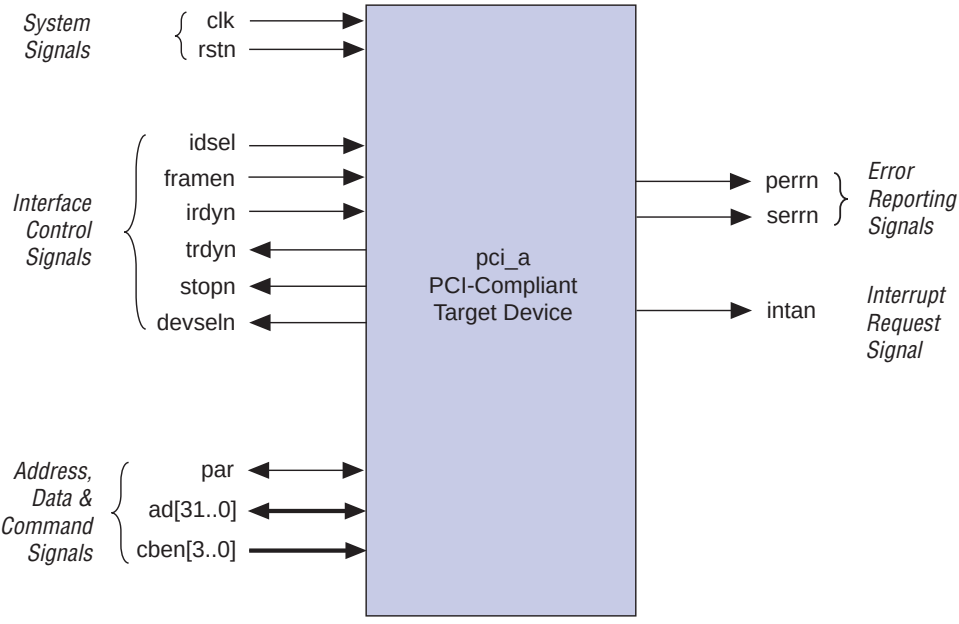
A pci_a target sequence begins when the master device asserts framen and drives the address of the target and the command on the PCI bus. When the target device decodes its address on the PCI bus, it asserts devseln to indicate to the master that it has accepted the transaction. The master will then 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 15.

As a target device, the pci_a function only supports single-cycle accesses; therefore, the pci_a function 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.

### PCI Bus Commands

Table 7 summarizes the PCI bus commands that are supported by the pci_a function.

*Table 7. PCI Bus Command Support Summary*

| cben[3..0] Value | Bus Command Cycle | Target Support | Master Support |
|---|---|---|---|
| 0110 | Memory read | ✓ | ✓ |
| 0111 | Memory write | ✓ | ✓ |
| 1010 | Configuration read | ✓ | |
| 1011 | Configuration write | ✓ | |

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

## 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 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 function.

Table 8 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 function.

*Table 8. PCI Bus Configuration Registers*

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

Table 9 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 function is instantiated in the MAX+PLUS II software. For example, the device ID register value can be modified from its default value by changing the DEVICE_ID parameter in the MAX+PLUS II software. The specified default state is defined as the state of the register when the PCI bus is reset.

*Table 9. pci_a-Supported Configuration Registers Address Map (Part 1 of 2)*

| 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 |

*Table 9. pci_a-Supported Configuration Registers Address Map (Part 2 of 2)*

| Address Offset (Hexadecimal) | Range Reserved (Hexadecimal) | Bytes Used/ Reserved | Read/Write | Mnemonic | Register Name |
|---|---|---|---|---|---|
| 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 function). 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 6), designers can change the value of the vendor ID register to their PCI SIG-assigned vendor ID value. See Table 10.

*Table 10. 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 function). 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 6 on page 13).

## 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 function to respond to and/or perform PCI bus accesses. See Table 11.

*Table 11. Command Register Format*

| Data Bit | Mnemonic | Read/Write | Definition |
|---|---|---|---|
| 0 | Unused | – | – |
| 1 | mem_ena | Read/Write | Memory access enable. When high, mem_ena enables the pci_a function to respond to the PCI bus memory accesses as a target. Because the DMA registers are set via memory target accesses, the mem_ena bit must be set as part of the initialization operation for the pci_a function to perform DMA transfers. |
| 2 | mstr_ena | Read/Write | Master enable. When high, mstr_ena enables the pci_a function to acquire mastership of the PCI bus. For the pci_a function to perform DMA transfers, the mstr_ena bit must be set as a part of the initialization operation. |
| 5..3 | Unused | – | – |
| 6 | perr_ena | Read/Write | Parity error enable. When high, perr_ena enables the pci_a function to report parity errors via the perrn output. |
| 7 | Unused | – | – |
| 8 | serr_ena | | System error enable. When high, serr_ena enables the pci_a function to report address parity errors via the serrn output. However, to signal a system error, the perr_ena 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 12.

*Table 12. Status Register Format*

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

*Table 13. Revision ID Register Format*

| Data Bit | Mnemonic | Read/Write | Definition |
|---|---|---|---|
| 7..0 | rev_id | 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, Revision 2.1*** for detailed bit information. See Table 14. 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 6).

*Table 14. Class Code Register Format*

| Data Bit | Mnemonic | Read/Write | Definition |
|---|---|---|---|
| 23..0 | class | 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 function can retain ownership of the PCI bus. After initiating a transaction, the pci_a function decrements its latency timer by one on the rising edge of each clock. The default value of the latency timer register is 00 hex. See Table 15.

*Table 15. Latency Timer Register Format*

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

## Header Type Register (Offset = 0E Hex)

Header type is an 8-bit read-only register that identifies the pci_a function as a single-function device. The default value of the header type register is 00 hex. See Table 16.

*Table 16. Header Type Register Format*

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

## Base Address Register Zero (Offset = 10 Hex)

Depending on the value of the BAR0_RW_BITS parameter, base address register zero (BAR0) consists of registers ranging from 12 to 1 bit. The BAR0_RW_BITS can be set when the pci_a function is instantiated, and determines the base memory address of the pci_a target space. This process is done in accordance with the ***PCI Local Bus Specification, Revision 2.1.,*** which states that the number of bits implemented as read/write registers defines the amount of memory address space reserved by the BAR. Power-up software can determine how much address space a device requires by writing a value of all 1s to the BAR and then reading the value back. To specify the required address space, the pci_a function will return 0s in all the lower bits. The amount of required address space is generally a function of the value of the BAR0_RW_BITS parameter, i.e., assuming BAR0_RW_BITS = *n*, the reserved address space is $2^{(32-n)}$ bytes. For example, when BAR0_RW_BITS = 4, the reserved address space is $2^{(32-4)}$ bytes, or 256 Mbytes. See Table 17.

*Table 17. Base Address Register Format (Part 1 of 2)*

| Data Bit | Mnemonic | Read/Write | Definition |
|---|---|---|---|
| 0 | mem_ind | Read | Memory indicator. The mem_ind bit indicates whether the register is I/O or a memory address decoder. In the pci_a function, the mem_ind bit is tied to GND, which indicates a memory address decoder. |
| 2..1 | mem_type | Read | Memory type. The mem_type bits indicate the type of memory that can be implemented in the pci_a function 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 pre_fetch bit indicates whether the block of memory defined by BAR0 is prefetchable by the host bridge. In the pci_a function, the address space is not prefetchable, i.e., it reads as low. |

*Table 17. Base Address Register Format (Continued) (Part 2 of 2)*

| Data Bit | Mnemonic | Read/Write | Definition |
|---|---|---|---|
| 31-BAR0_RW_BITS | Unused | – | – |
| 31..(32-BAR0_RW_BITS) | 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 18. 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 6).

*Table 18. Subsystem Vendor ID Register Format*

| 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 19. 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 6).

*Table 19. Subsystem ID Register Format*

| 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 20.

*Table 20. Interrupt Line Register Format*

| 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 function's PCI bus interrupt request line to be intan. The default value of the interrupt pin register is 01 hex. See Table 21.

*Table 21. Interrupt Pin Register Format*

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

## Minimum Grant Register (Offset = 3E Hex)

Minimum grant register consists of an 8-bit read-only register that defines the length of time the pci_a function would like to retain mastership of the PCI bus. The value set in this register indicates the required burst period length in 250-ns increments. The pci_a function requests a timeslice of 4 microseconds. The default state of the minimum grant register is 10 hex. See Table 22.

*Table 22. Minimum Grant Register Format*

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

## Maximum Latency Register (Offset = 3F Hex)

The maximum latency register is an 8-bit read-only register that defines the frequency in which the pci_a function 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 function has no major requirements for maximum latency. See Table 23.

*Table 23. Maximum Latency Register Format*

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

# PCI Bus Transactions

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

- All pci_a DMA accesses to the PCI bus are quad-byte, or 32-bit transfers; therefore, all byte enables are active for the duration of master data transfers. During pci_a external target write accesses, the transfers are byte selectable.
- Although Figures 5 through 16 show PCI bus signals as tri-stated when not driven by the pci_a function, 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 function accesses the PCI bus for three types of transactions:

- Target
- Configuration
- Master

## Target Transactions

The sequence of events for the beginning of all target transfers is exactly the same. A target read or write transaction begins after the master acquires mastership of the PCI bus. The master device then asserts framen and drives the address on the ad[31..0] bus and command on the cben[3..0] bus. The pci_a function latches the address and command signals on the first clock edge when framen is asserted and starts decoding the address.

### *Target Read Transactions*

The pci_a function 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. (See "External Target Read Transaction" on page 29 for more information.) A target read transaction from the local side target memory space requires more time because the pci_a function must wait for the local side to supply it with data.

### Internal Target Read Transaction

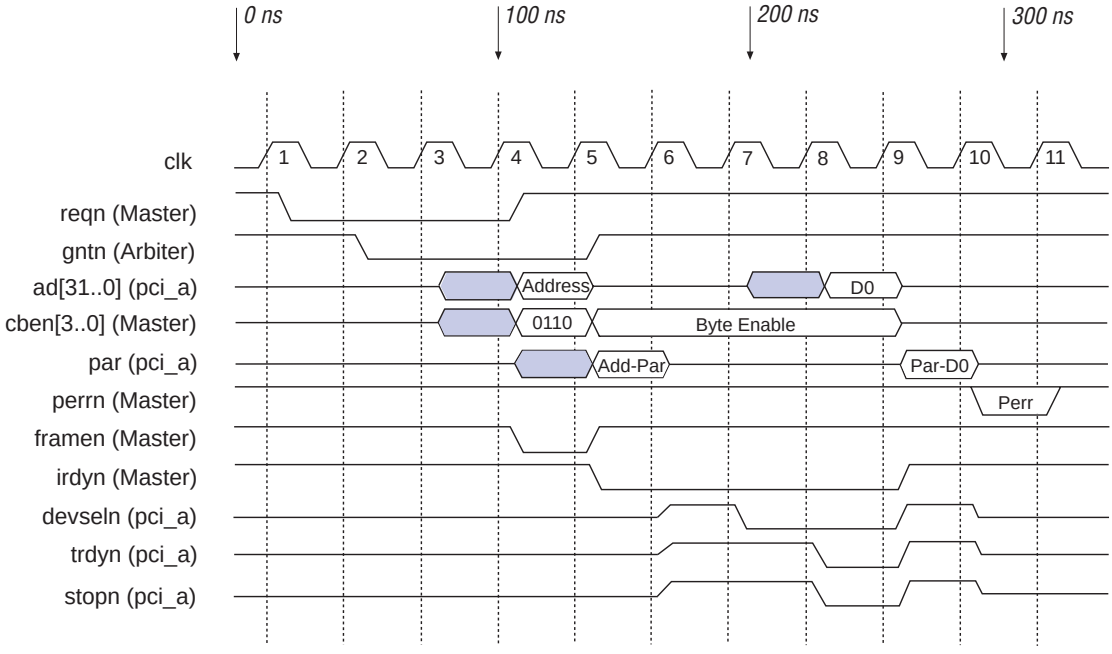
Immediately after the address phase (clock four), the master deasserts framen and asserts irdyn, indicating both of the following:

- The transaction contains a single data phase.
- The master device is ready to read the data that the pci_a function 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 function latches the address. The pci_a function 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 function 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 function asserts devseln in clock seven, which indicates to the master device that pci_a has claimed the transaction. The devseln is then sampled by the master device on the rising-edge of clock eight, which is slow decode, as defined by the PCI specification. 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 function 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 function 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 function 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

The sequence of events in an external target read transaction is identical to an internal target read transaction. However, 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 idle. If an external target read transaction is received by the pci_a function while the DMA is not idle, the pci_a function signals a retry.

Because the pci_a function must wait for the local side to supply it with data, a target read transaction from the local side target memory space (external target read) requires more time. If the local logic cannot supply the data within one clock after l_csn and l_rdn are asserted, l_holdn can be asserted low to halt the data transfers. The l_holdn signal may be driven low until the data is presented on the l_dat_in[31..0] bus.

PCI specification requires that the first data phase of a target transaction completes within 16 clock cycles. The local device must ensure that the PCI specification is not violated by an excessively long l_holdn assertion.

Figures 6 shows the timing of a pci_a external target read transaction.

***Figure 6. External Target Read Transaction***

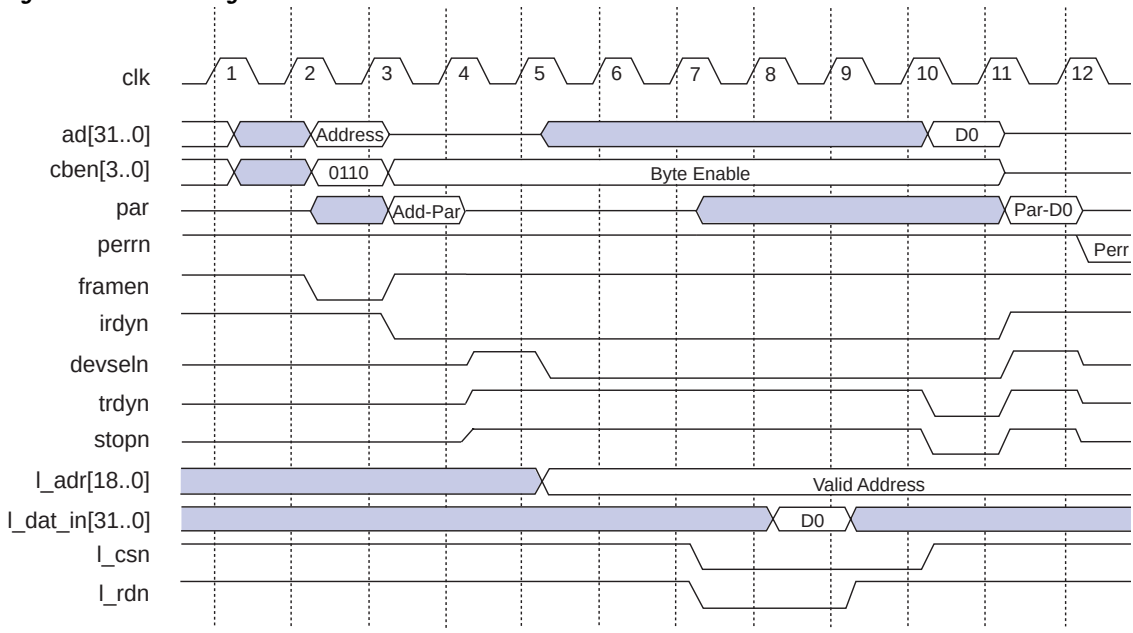
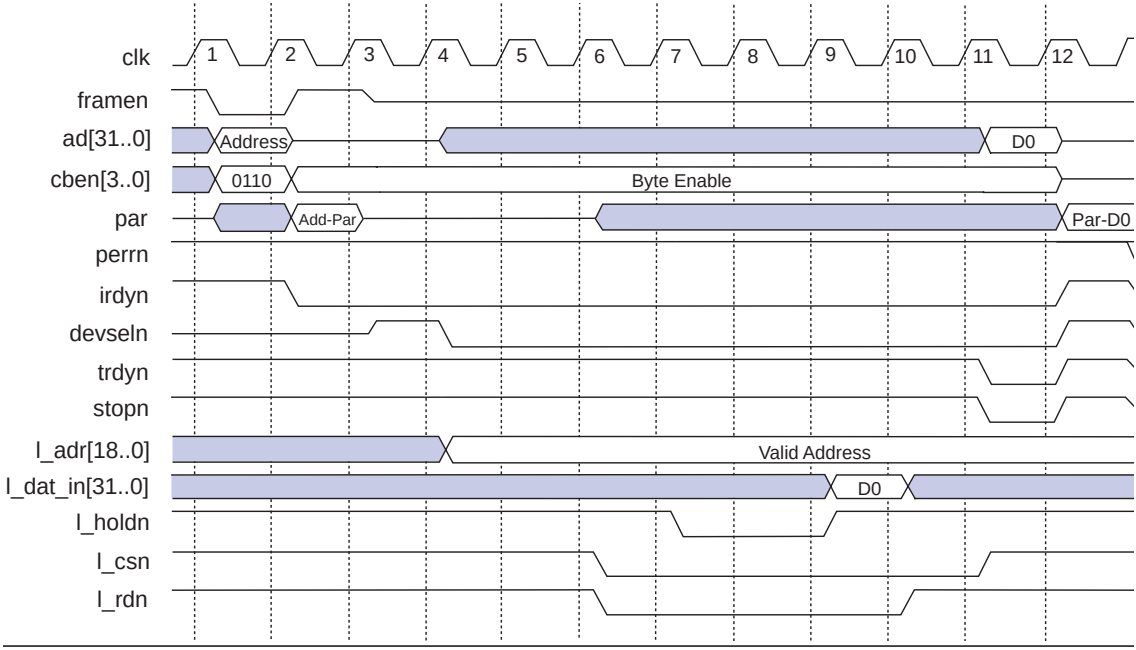



Figure 7 illustrates an external target read transfer where l_holdn is used to insert additional wait states on the local side. Unable to supply data immediately when l_csn and l_rdn are asserted, the local logic asserts l_holdn in clock eight for two clock cycles. The local side supplies the data on the l_dat_in[31..0] bus in clock 10 and deasserts l_holdn. The pci_a function latches the data internally on the rising edge of clock 11 and deasserts l_rdn. The l_csn is deasserted one clock later. The pci_a drives the data on the PCI bus one clock after it latches it from the local side (clock 13). Because l_holdn is registered, the local side must follow the $t_{SU}$ timing requirements (provided by the MAX+PLUS II Timing Analyzer) when it drives l_holdn.

To avoid excessive latency, the PCI specification requires that PCI target devices complete the initial data transaction within 16 clocks after framen is asserted. (The local logic must ensure that this PCI specification is met.) Therefore, l_holdn cannot be held active for more than 10 clock cycles.

***Figure 7. External Target Read Transaction with l_holdn Asserted***




### *Target Write Transactions*

The pci_a function 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.

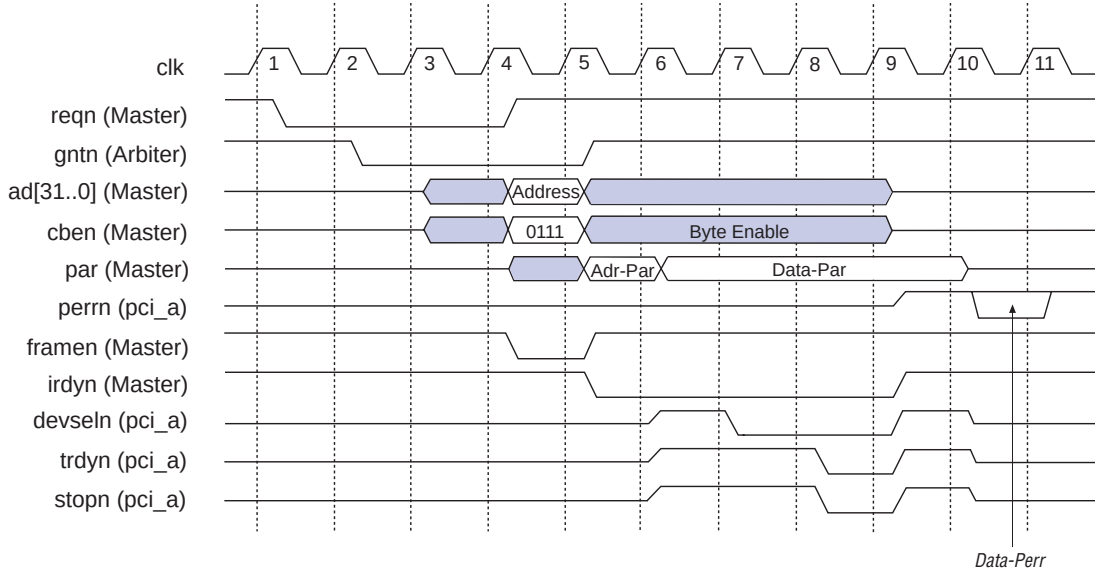
If the master device is not ready for the data phase to begin, irdyn is delayed and framen is not deasserted until the clock where irdyn goes active. If the master is attempting a burst access, it will keep both framen and irdyn signals asserted. However, because the pci_a function does not support target 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.

Figure 8 shows a typical waveform for an internal target write transaction. The address phase occurs during clock four, and the data phase begins in clock five. The pci_a function claims the transaction in clock eight by asserting devseln. On the rising edge of clock nine, data is transferred from the master device to the pci_a function because both irdyn and trdyn are asserted. At the same time when the pci_a function asserts trdyn, it also asserts stopn to indicate that it is unable to receive more data. The pci_a function always asserts stopn and trdyn at the same time to ensure that only one data phase occurs during each target transaction.

The master device drives par active in clock five for parity of the address bits, and clock six for parity of the data bits. If a parity error occurs, the pci_a function will drive perrn one clock cycle later.

In clock nine, because the data has been sampled, the pci_a function releases the ad[31..0] and cben[3..0] buses. One clock later par is released by the master device. The pci_a drives devseln, trdyn, and stopn high in clock nine and releases them one clock later.

***Figure 8. Internal Target Write Transaction***

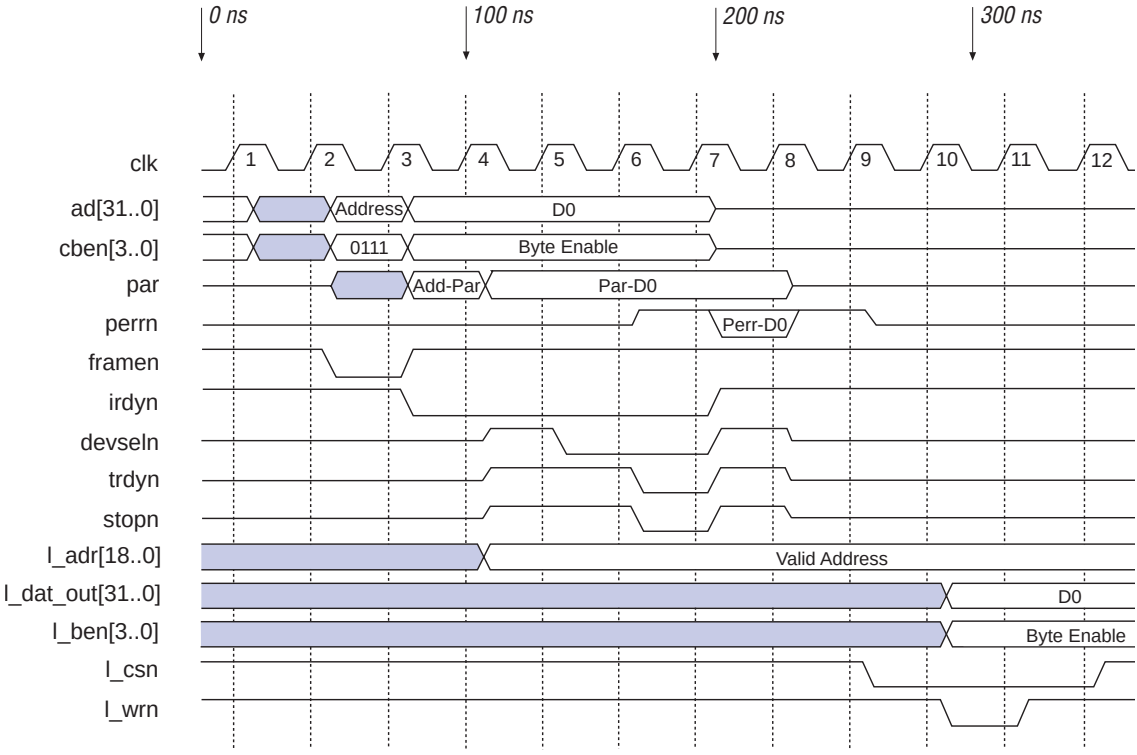



### External Target Write Transaction

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 function 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 function will subsequently assert its l_csn signal to indicate to the local side that there is a pending target access; one clock later (clock 10), the l_wrn is asserted and data is driven on l_dat_out[31..0] bus and the byte enables are driven on the l_ben[3..0] bus. Figure 9 shows the timing of an external target write transaction.

***Figure 9. External Target Write Transaction***




Similar to an external target read transaction, if the local logic is unable to receive the 32-bit data from the l_dat_out[31..0] bus, l_hold can be applied to delay the data transfer. Figure 10 on page 34 depicts an external target write transaction where l_holdn is asserted to extend the time required by the local side to transfer the data.

When the pci_a drives l_csn low, the l_wrn is driven low one clock cycle later. Because the local logic is unable to receive the write data, it drives l_holdn in clock 10.

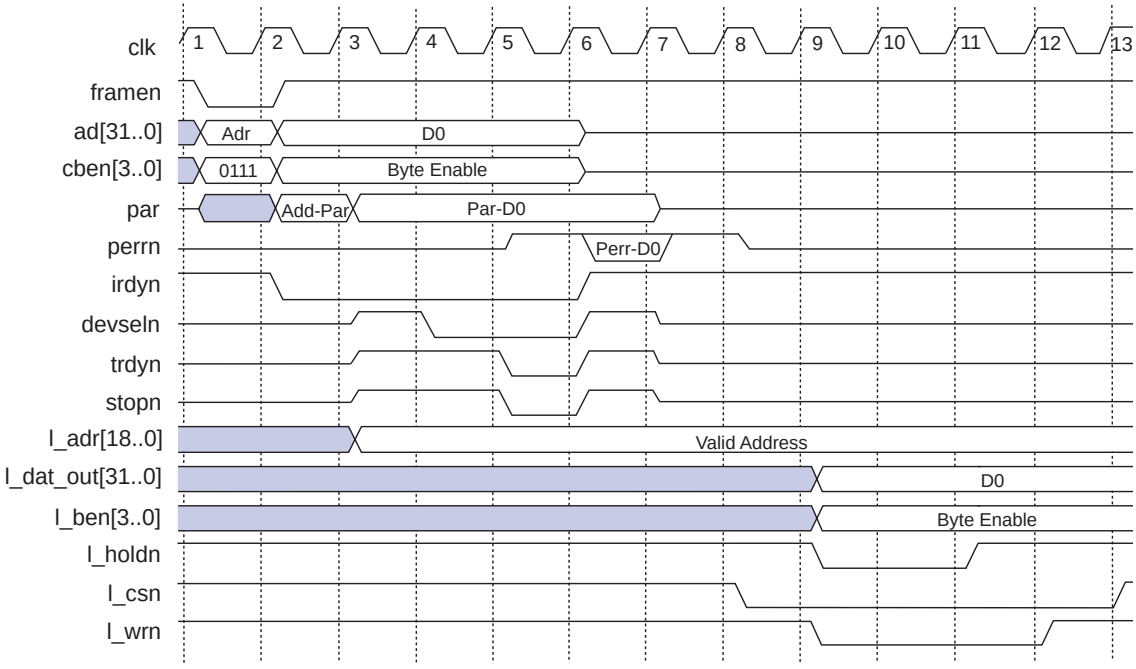
The local side can detect that the local target data transfer is a write cycle because in clock eight, when l_csn is asserted, l_rdn is not asserted.

Because pci_a detects the assertion of l_holdn, it continues to drive data0 (D0) on the l_dat_out[31..0] bus as well as l_csn and l_wrn until l_holdn is deasserted. The local application must assert l_holdn by clock 10 to extend the data cycle.

The local logic latches the data at clock 13. The l_wrn signal is asserted until one clock after l_holdn is deasserted; l_csn is then deasserted one clock after l_wrn is deasserted.

The pci_a function finishes the data transfers on the PCI bus before the data is presented to the local side. During an external target write transaction, l_holdn can be held active many clock cycles without affecting the PCI bus performance. However, it is generally a good practice to deassert l_holdn as soon as possible. Otherwise, if a PCI agent attempts to access the pci_a function again while the function has valid data, the pci_a function issues a retry.

***Figure 10. External Target Write Transaction with l_holdn Asserted***



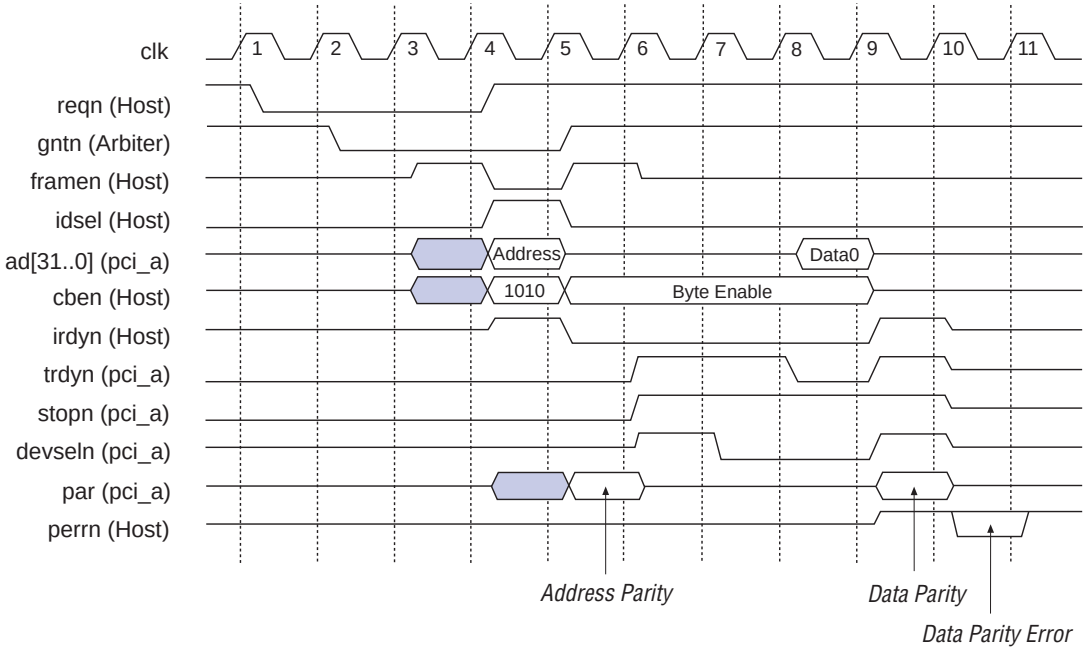
## Configuration Transactions

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 `idsel` signal of the PCI bus agent that it wants to access. If a PCI bus agent decodes the configuration command and detects its `idsel` to be high, the agent will claim the configuration access and assert `devseln`.

### *PCI Configuration Read Transaction*

Figure 11 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 27 except for the `idsel` signal, which is active during the address phase of a configuration transaction.

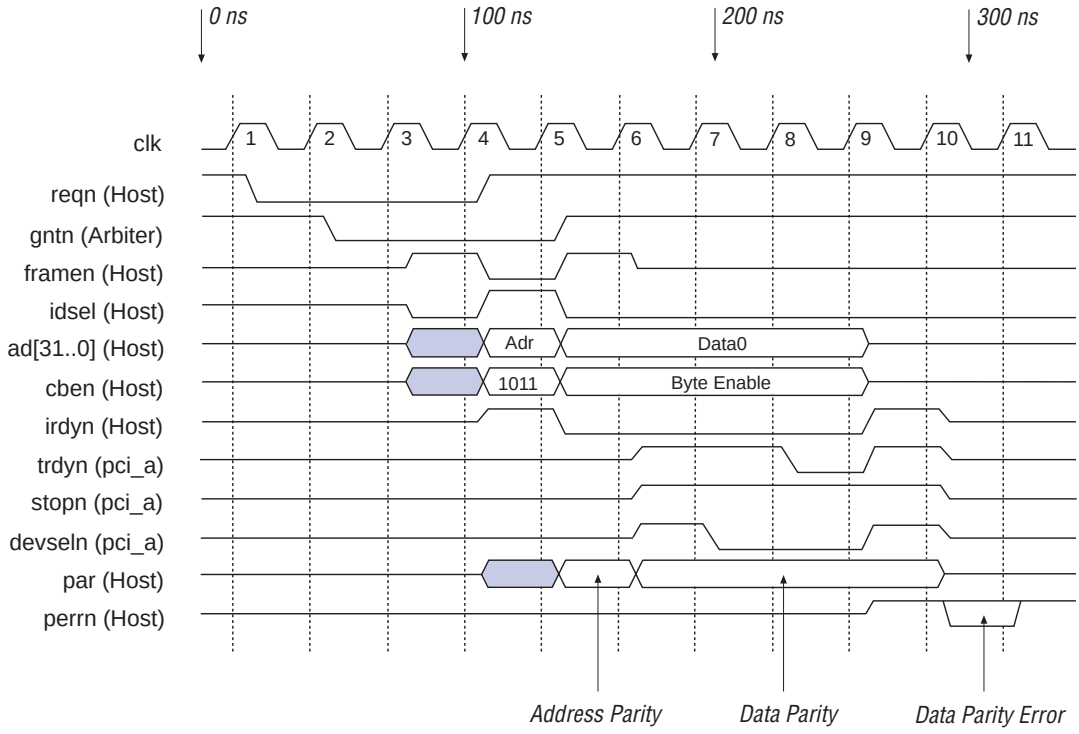
***Figure 11. Configuration Read Transaction***




### *PCI Configuration Write Transaction*

Figure 12 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 31 except for the `idsel` signal, which is active during the address phase of a configuration transactions.

*Figure 12. Configuration Write Transaction*




## Master Transactions

Master transactions in the pci_a function 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 Host Programming Guidelines" on page 54 for more detailed information on DMA register loading). The pci_a function waits for the local side to assert l_req, which indicates to the pci_a function that it can begin the DMA operation.

In a DMA read (PCI to local side) transaction, the pci_a function immediately asserts reqn to acquire mastership of the PCI bus. After the arbiter asserts gntn, the pci_a function 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 function 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 function begins the address phase.

### *Master Read Transactions*

The pci_a function 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 function has acquired mastership of the PCI bus, the start of a master read transaction is indicated when the pci_a function asserts framen.

After the master read transaction is initiated, the target devices latch the address and command on the clock edge when framen is active and start the address decode. The pci_a function is not ready to read data until clock five; therefore, framen 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 function on the rising-edge of clock four, which depicts a fast decode target device.

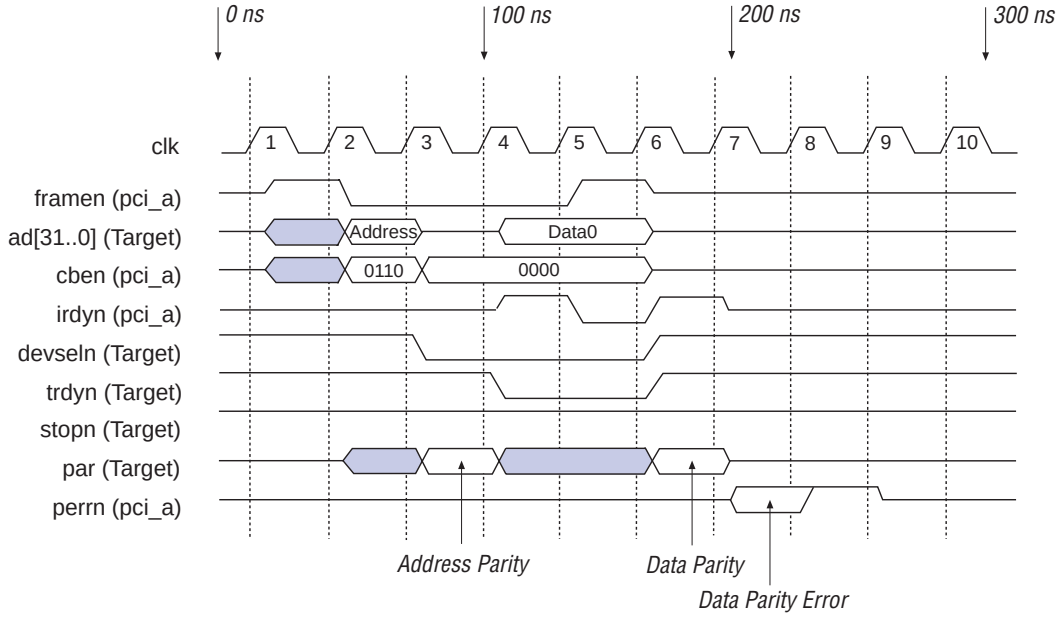
To indicate that it is ready to send data, the target device simultaneously asserts trdyn and drives data on the ad[31..0] bus beginning in clock four. The data phase begins in clock five when irdyn and trdyn are active and finishes on the rising edge of clock six with data latched by the pci_a function.

The pci_a function 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 function 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 13 shows the timing of a pci_a function master read transaction. The figure assumes the pci_a function has already acquired mastership of the PCI bus.

***Figure 13. 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 37. After the address phase, the protocol changes to reflect the additional read transactions.

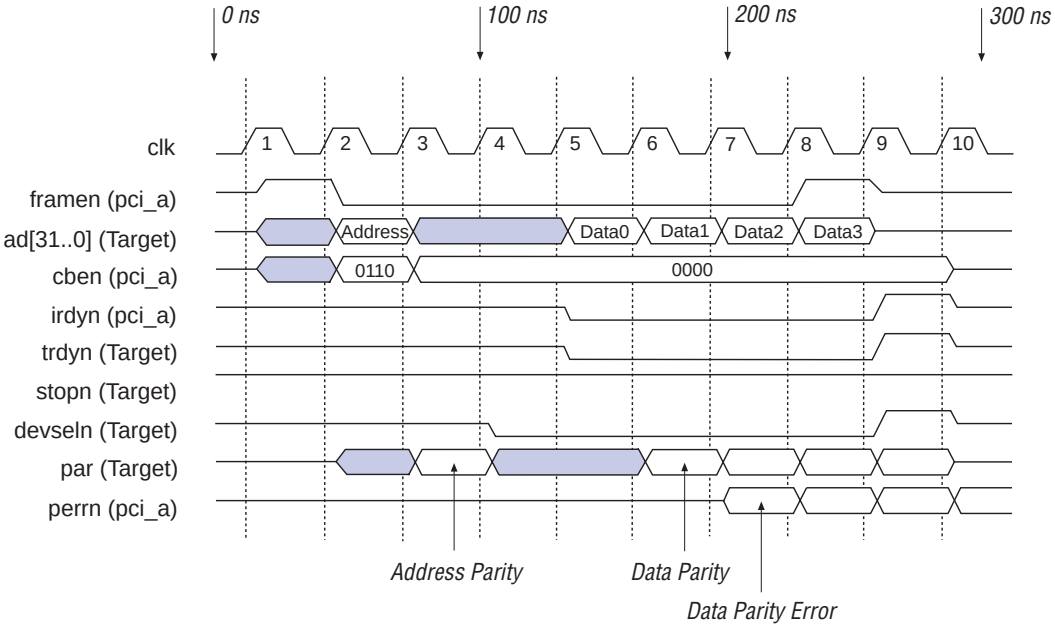
After the master burst read transaction is initiated, the selected target device asserts `devseln` in clock three, and the `pci_a` function samples `devseln` on the rising edge of clock five. This example displays a fast decode target. The target device then signals to the `pci_a` that it is ready to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

The `pci_a` function 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 seven, eight, and nine for parity of the second, third and fourth data phases.

Figure 14 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` function 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 function releases irdyn in clock nine.

***Figure 14. Master Burst Read Transaction***




## *Master Write Transactions*

The pci_a function 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 function has acquired mastership of the PCI bus, the start of a master device write transaction is indicated when the pci_a function 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 function in clock five, which depicts a medium decode target device.

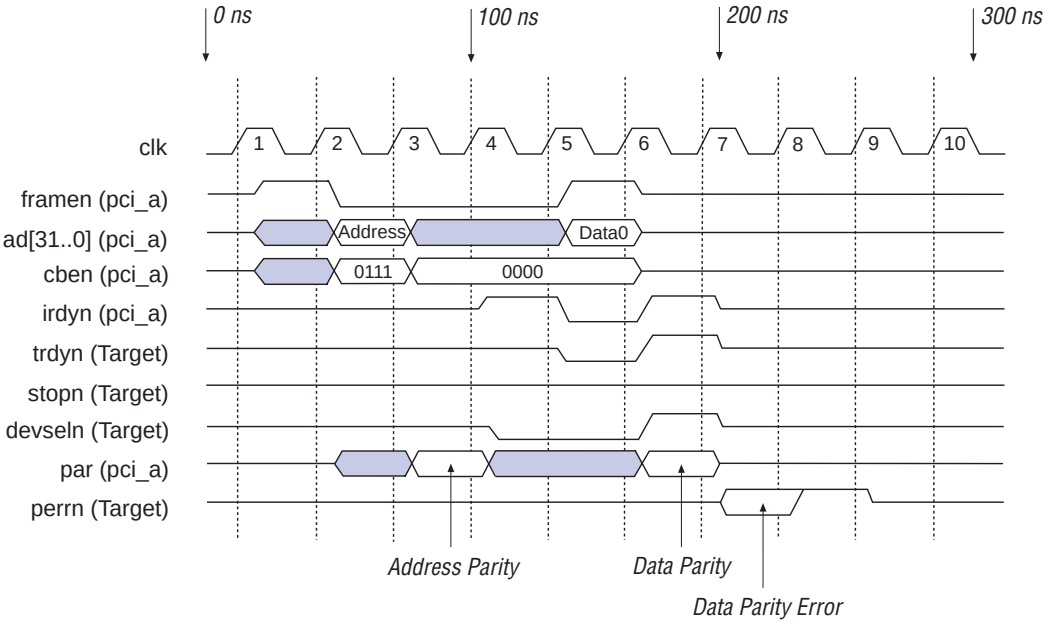
To indicate that it is ready to receive data, the target device drives trdyn active in clock five. Then, the pci_a function drives data on the ad[31..0] bus beginning in clock five and simultaneously with the assertion of irdyn. The data phase begins in clock five when irdyn and trdyn are active, and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

The pci_a function 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 function releases the ad[31..0]bus and cben[3..0] in clock six. One clock later, par is released by the pci_a function, 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 function drives irdyn high in clock six before releasing it in clock seven.

Figure 15 shows the timing of a pci_a master write transaction. The figure assumes the pci_a function has already acquired mastership of the PCI bus.

***Figure 15. 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 39. 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` function 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.

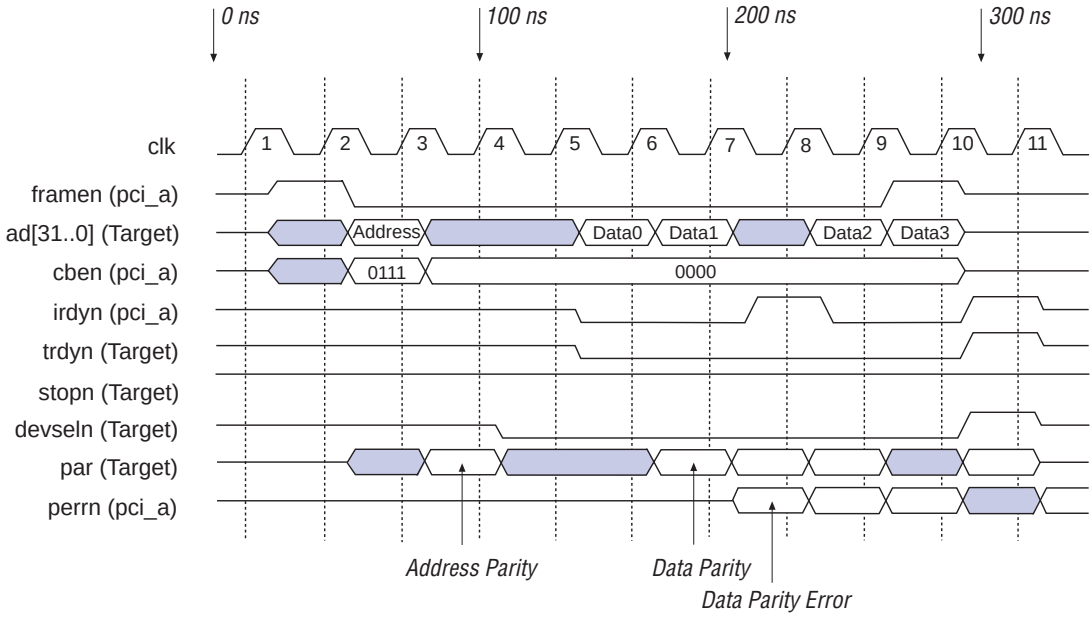
The master burst write transaction example in Figure 16 shows the data phases occurring in clocks five, six, seven, and nine when `irdyn` and `trdyn` are both active.

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

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

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

*Figure 16. 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
- Initializing DMA transfers from the local side
- General host programming guidelines

## Target Address Space

The pci_a function 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 24 lists the pci_a function's memory space address map. The pci_a function BAR0 address space ranges from 1 Mbyte to 2 Gbytes of contiguous address divided into two equal-sized regions (lower and upper). Each region reserves half of the total address space reserved by BAR0. The lower 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 24. Memory Space Address Map*

| Memory Space | Block Size (DWORDs) | Address Offset *Note (1)* | Words Used | Read/ Write | Description |
|---|---|---|---|---|---|
| BAR0 | 1/2 of reserved space | 00000h-7FFFFh | 4 bytes | Read/write | DMA registers |
| BAR0 | 1/2 of reserved space | 80000h-FFFFFh | All | Read/write | User-defined memory space, ranging in size from 512 Kbytes to 2 Gbytes |

*Note:*
(1) These values are based on the BAR0_RW_BITS parameter set to 12.

## 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 25 lists the pci_a function's DMA registers memory map.

*Table 25. Internal Target Registers Memory Map*

| Range Reserved *Note (1)* | Bytes Used/Reserved | Read/Write | Mnemonic | Default State (Hexadecimal) | Register Name |
|---|---|---|---|---|---|
| 00000h-00003h | 8/32 | Read/write | dma_csr | 00000000 | DMA control/status |
| 00004h-00007h | 32/32 | Read/write | dma_acr | 00000000 | DMA address counter |
| 00008h-0000Bh | 17/32 | Read/write | dma_bcr | 00000000 | DMA byte counter |
| 0000Ch-0000Fh | 8/32 | Read | dma_isr | 00000000 | DMA interrupt status |

*Note:*
(1) These values are based on the BAR0_RW_BITS parameter set to 12.

## 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 function 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 function's DMA operation, and provides status of the current memory transfer. See Table 26.

*Table 26. 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 10). The l_reset output of the pci_a function is active as long as the l_rst bit is high. (The l_reset output is also active for PCI bus resets.) |
| 3 | write | Read/write | Memory read/write. The write bit determines the direction of the pci_a function's DMA transfer. When write is high, the data flows from the local side to the PCI bus (PCI bus write); when write is low, the data flows from the PCI bus to the local device (PCI bus read). |
| 4 | dma_ena | Read/write | DMA enable. When high, dma_ena allows pci_a to respond to DMA requests from the local side (l_req) as long as the PCI bus activity is not stopped due to a pending interrupt, etc. |
| 5 | tci_dis | Read/write | Transfer complete interrupt disable. When high, tci_dis disables dma_tc (bit 3 of the DMA interrupt status register) from generating PCI bus interrupts. |

*Table 26. DMA Control & Status Register Format (Part 2 of 2)*

| Data Bit | Mnemonic | Read/Write | Definition |
|---|---|---|---|
| 6 | dma_on | Read | DMA on. When high, dma_on indicates that the pci_a function can request mastership of the PCI bus (reqn) if prompted by the local side (i.e., an active l_req). The dma_on bit is high when the address is loaded (ad_loaded), the DMA is enabled, and there are no pending errors. 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. The dma_on bit becomes set whether the write transaction occurs from the local side or via a target access. |
| 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 function must begin on DWORD boundaries. For monitoring progress, the dma_acr can be read via l_dma_acr_out[] ports. See Table 27.

*Table 27. DMA Address Counter Register Format*

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

### *Byte Counter Register (Offset = 00008 Hex)*

The DMA byte counter register (dma_bcr) is a 17-bit register consisting of a 15-bit counter (bits 16..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 function must be DWORD transfers. Reading the dma_bcr during a memory transfer can be achieved via the l_dma_bcr_out[] ports. See Table 28.

| Table 28. DMA Byte Counter Register Format | | | |
|---|---|---|---|
| **Data Bit** | **Name** | **Read/Write** | **Definition** |
| 1..0 | byte_cntr | Read | Bits are tied to GND. |
| 16..2 | byte_cntr | Read/write | 15-bit counter. |
| 31..17 | 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 29.

| Table 29. DMA Interrupt Status Register Format | | | |
|---|---|---|---|
| **Data Bit** | **Mnemonic** | **Read/Write** | **Definition** |
| 0 | int_pend | Read | The pci_a function automatically asserts int_pend to indicate that a pci_a interrupt is pending. The three possible interrupt signals from the pci_a are err_pend, dma_tc, and int_irq. |
| 1 | err_pend | Read | When high, err_pend indicates that an error occurred during a pci_a-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 err_pend: mstr_abrt, tar_abrt, and det_par_err. See "Control & Status Register (Offset = 00000 Hex)" on page 44. |
| 2 | int_irq | Read | When high, int_irq indicates that the local side is requesting an interrupt, i.e., the l_irqn input is asserted. |
| 3 | dma_tc | Read | When high, dma_tc indicates that the pci_a-initiated DMA transfer is complete. When the pci_a function sets the dma_tc bit, an interrupt will be generated on the intan output as long as interrupts are enabled by the int_ena bit (bit 0 of the dma_csr) and not disabled by the tci_dis bit (bit 5 of the dma_csr). The dma_tc bit is reset in one of three ways: a read transaction to the dma_isr; a write transaction to the dma_csr, which sets the flush bit (bit 1 of the dma_csr); or by writing to the dma_acr from the local side. |
| 4 | ad_loaded | Read | When high, ad_loaded indicates that the address has been loaded in the dma_acr. This bit is cleared in one of three ways: when the DMA operation is complete and the dma_tc bit is set; when the flush bit is set; or when the rstn input is asserted from the PCI bus. The ad_loaded bit triggers the beginning of a DMA operation because it sets the dma_on bit in the dma_acr register. It is automatically set by the pci_a when a write operation to the dma_acr is performed. Therefore, the dma_acr should be written to last when a DMA operation is being loaded into the DMA registers. |
| 31..5 | Unused | – | – |

## DMA Transactions

As a master device, the pci_a function performs DMA read and write transactions to system memory (typically 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 function to system memory consists of two separate transfers:

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

All DMA (PCI bus) write transactions from the pci_a function 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 transaction consists of:

- A pci_a master device read from a PCI agent to the pci_a RAM buffer.
- A write from the pci_a function's 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 function asserts reqn and waits for gntn to become active before assuming mastership of the PCI bus.

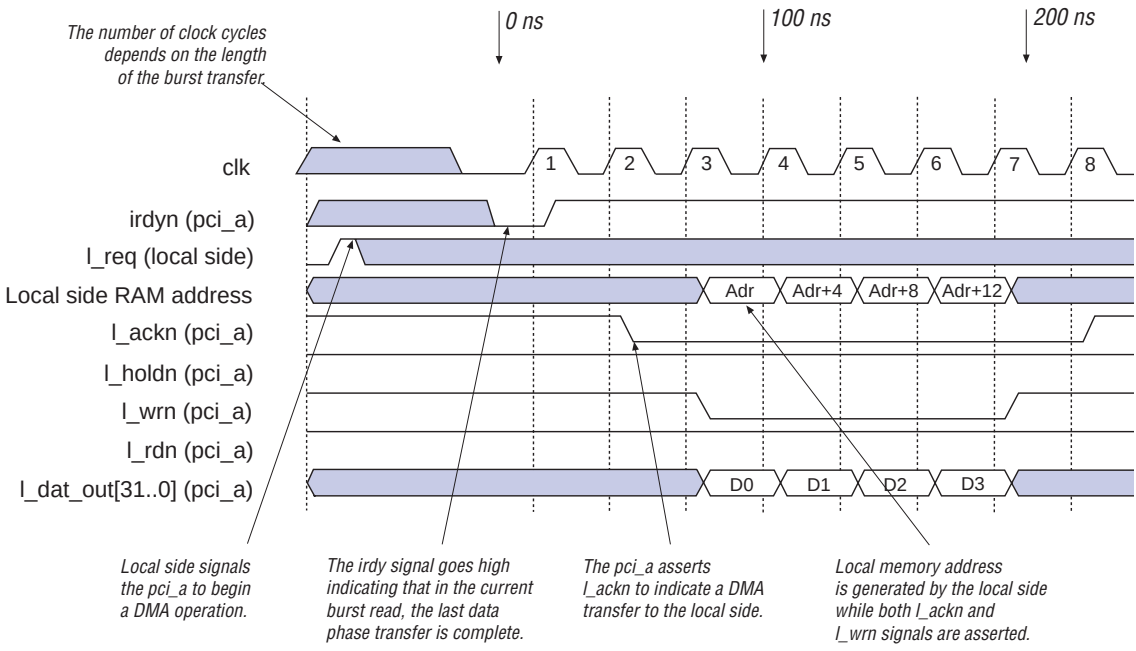
4. The pci_a function reads up to the 16 DWORDs from the PCI bus system memory and loads the data into the pci_a function's RAM buffer.

5. Once the PCI transfer is complete, the pci_a function 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 17 the address is not generated from the pci_a.

6. The pci_a function writes the data from the pci_a function's RAM buffer onto the l_dat_out[31..0] bus. When the last data word is written, the pci_a function disables l_ackn and l_wrn.

7. If the dma_bcr expires (i.e., the specified number of data bytes have been transferred), the pci_a function sets the dma_tc bit in the dma_isr register and asserts intan, provided that the interrupt is enabled and tci_dis = 0. Otherwise, steps 2 through 5 are repeated until dma_bcr expiration or until a DMA error occurs. See Figure 17.

***Figure 17. PCI Bus DMA Read 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 function's RAM buffer.
- A pci_a master write from the pci_a function's RAM buffer to a PCI bus agent.

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

1. The local side or 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.

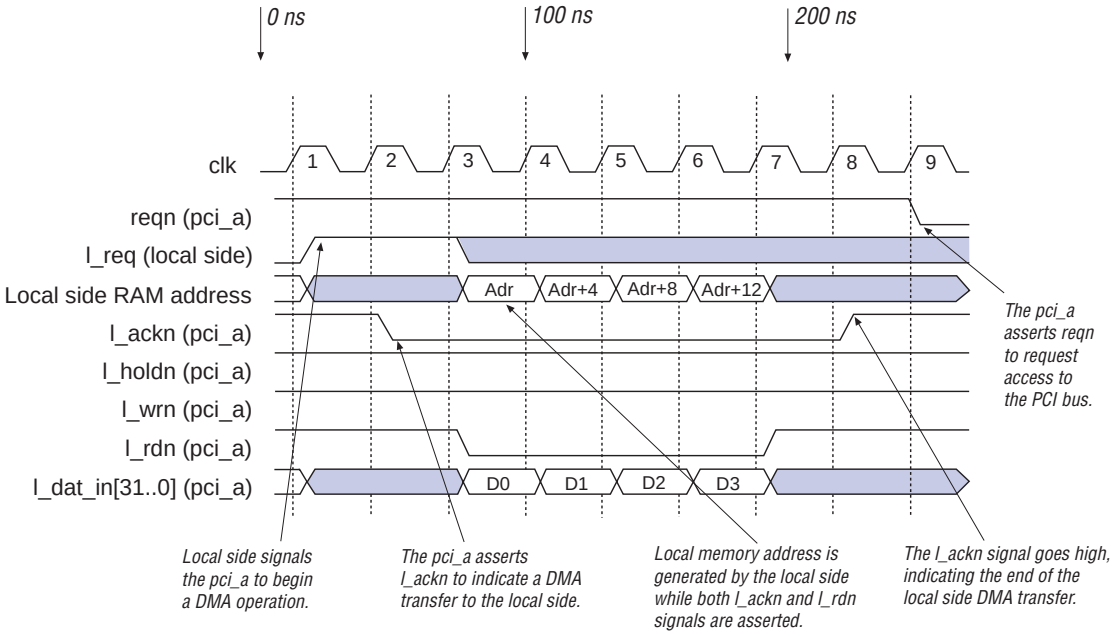
4. The pci_a function reads the data from the l_dat_in[31..0] bus into the pci_a RAM buffer. When the last DWORD in the DMA transfer is read, or when the RAM buffer is full, the pci_a function disables l_ackn and l_rdn.

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 expires (i.e., the specified number of data bytes have been transferred), the pci_a sets the dma_tc bit in dma_isr register and asserts intan provided that interrupt is enabled and tci_dis= 0. Otherwise, steps 2 through 5 are repeated until the dma_bcr expiration or until a DMA error occurs. See Figure 18.

*Figure 18. PCI Bus DMA Write Transaction*




## Initializing DMA Transfers from the Local Side

The pci_a function version 2.0 allows both the local side and the host to perform DMA read transactions. This section discusses how the local side may set up the DMA registers to initiate a master transfer. For more information on how the host may initiate DMA, see "General Host Programming Guidelines" on page 54.

The pci_a function's 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.

After the configuration space registers are properly set, either the host or the local logic can initiate burst DMA transfers by writing to the DMA registers in the pci_a function. This section is divided into two tasks:

- Initializing the pci_a function for a DMA read transaction
- Initializing the pci_a function for a DMA write transaction

### *Initializing the pci_a Function for a DMA Read Transaction*

To initialize a DMA read cycle, the local logic sequentially writes to the dma_csr, dma_bcr, and dma_acr registers. After the local logic writes to the dma_acr, the ad_loaded bit in the dma_isr register is set. The ad_loaded bit will set the dma_on bit in the dma_csr register if the DMA is enabled (dma_csr bit 4) and no errors are pending (dma_isr bit 1). When dma_on bit is set, the pci_a waits for the local device to assert l_req before it actually begins the DMA read transaction by requesting mastership of the PCI bus. It is important to check that the dma_acr is written to last, i.e., after proper values have been set in the dma_bcr and dma_csr registers. See Table 30.

***Table 30. Initialization the pci_a Function for a DMA Read Operation***

| Address (Hexadecimal) | Register Name | Data (Hexadecimal) | Definition |
|---|---|---|---|
| BAR0: 0.0000 | dma_csr | 0000.0031 | The value in the dma_csr enables the interrupts and the DMA engine, and disables DMA terminal count interrupt. |
| BAR0: 0.0008 | dma_bcr | 00084 | The value written in this register indicates the amount of data (in bytes) for a DMA transfer. The value must be in multiples of DWORDs. |
| BAR0: 0.0004 | dma_acr | 00400000 | The PCI bus address where the transfer should begin. This address is automatically updated after every data transfer. |

Figure 19 on page 52 shows the timing of a local side DMA read transaction. In this example, the local logic requests to read 33 DWORDs (132 bytes) from the system memory starting at the address 00400000 hex. Figure 19 illustrates the following signal sequence:

1. The local logic asserts l_req in clock one, indicating that it is ready for a transfer. The assertion of l_req can be delayed until the local side is ready for the DMA transfer to commence.

2. In clock two, the local logic asserts l_dma_csr_wr while supplying data value for l_dma_dat_in[31..0] bus. A hexadecimal value of 31 indicates that bit 0, 4, and 5 of the DMA control and status register are set, which enables the DMA and interrupts, and disables the DMA terminal count interrupt. In this case, bit 3 is not set, which indicates a DMA read transfer.

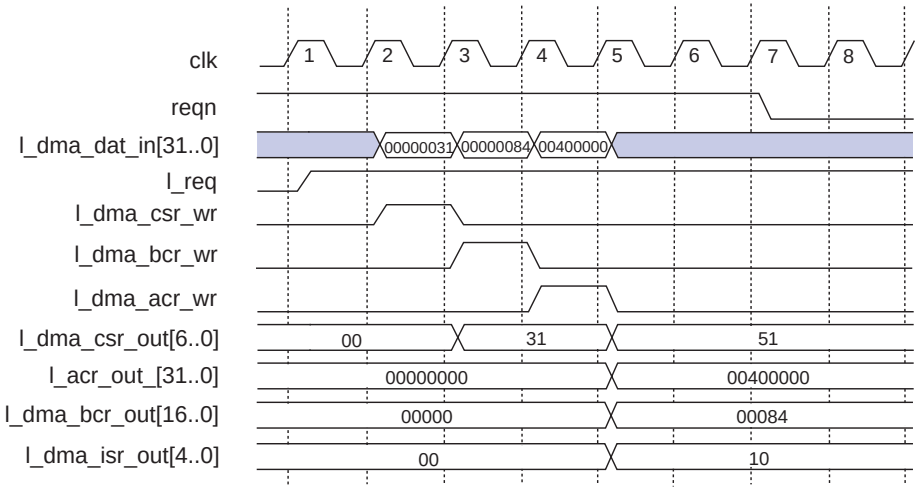
3. In clock three, the local logic asserts `l_dma_bcr_wr` while supplying the data value for the `dma_bcr` register on the `l_dma_dat_in[31..0]` bus. A hexadecimal value of `84` equals a decimal value of 132 bytes, indicating that the `pci_a` is going to read 33 DWORDs. Because the value of `l_dma_csr_out[6..0]` changes to the value written in clock 2, the write to the `dma_csr` register takes effect in clock 3.

4. The local logic asserts `l_dma_acr_wr` while supplying data value for the `dma_acr` register on the `l_dma_dat_in[31..0]` bus. This transaction writes the value of `00400000` hex into the `dma_acr` register. Thus, the `pci_a` function seeks to read from an address value of `00400000` hex.

5. In clock 5, the write transaction to the `dma_bcr` and `dma_acr` registers take effect. Figure 19 shows the changes in values on the `l_dma_bcr_out[16..0]` and `l_dma_acr_out[31..0]` buses. Figure 19 also shows changes in values on the `l_dma_isr_out[4..0]` and `l_dma_csr_out[6..0]` buses, which result from the `ad_loaded` and `dma_on` bits becoming set.

6. Because `l_req` is already asserted, the `pci_a` function seeks mastership of the PCI bus by asserting the `reqn` signal in clock seven. See Figure 19.

***Figure 19. Local SIde Initiated DMA Read Transaction***



### *Initializing the pci_a Function for a DMA Write Transaction*

Setting up the DMA registers for a burst write transaction from the local logic follows the same steps as setting up a DMA read transaction. The local logic sequentially writes the dma_csr, dma_bcr, and dma_acr registers. When the local logic writes to the dma_csr, dma_bcr, and dma_acr registers, the ad_loaded bit (bit 4 of the dma_isr) is set. The ad_loaded bit triggers the beginning of a DMA operation by setting the dma_on bit (bit 4 of the dma_csr), which prompts the pci_a to start the DMA write operation by asserting l_ackn and reading up to 16 DWORDs from the local side. Therefore, it is important to check that the dma_acr is written to last, i.e., after proper values have been set in the dma_bcr and dma_csr registers. See Table 31.

**Table 31. Initializing the pci_a Function for a DMA Write Operation**

| Address (Hexadecimal) | Register Name | Data (Hexadecimal) | Definition |
|---|---|---|---|
| BAR0: 0.0000 | dma_csr | 0000.0039 | The value in the dma_csr enables interrupts, indicates that the DMA operation is a write operation, enables the DMA engine and disables the DMA terminal count interrupt. |
| BAR0: 0.0008 | dma_bcr | 00084 | The value written in this register indicates the amount of data (in bytes) for a DMA transfer. The value must be in multiples of DWORDs (4 bytes). |
| BAR0: 0.0004 | dma_acr | 00400000 | The PCI bus address where the transfer should begin. This address is automatically updated after every data transfer. |

Figure 20 on page 54 shows the timing of a local side DMA register write transaction, and illustrates the following signal sequence:

1. The local logic asserts l_req in clock one, indicating that it is ready for a DMA transfer. The assertion of l_req can be delayed until the local side is ready for the DMA transfer to commence.

2. In clock two, the local logic asserts l_dma_csr_wr while suppling data value in the l_dma_dat_in[31..0] bus. A hexadecimal value of 39 is written to the dma_csr register, which enables interrupts, disables DMA terminal count interrupt, and enables the DMA engine and requests a write cycle.

3. In clock three, the local logic asserts l_dma_bcr_wr while supplying data value in the l_dma_dat_in[31..0] bus. This signal sequence writes the value of 84 hexadecimal (132 bytes) into the dma_bcr register. In clock three, the write to dma_csr takes place because the value of l_dma_csr_out[6..0] changed to the value written in clock two.

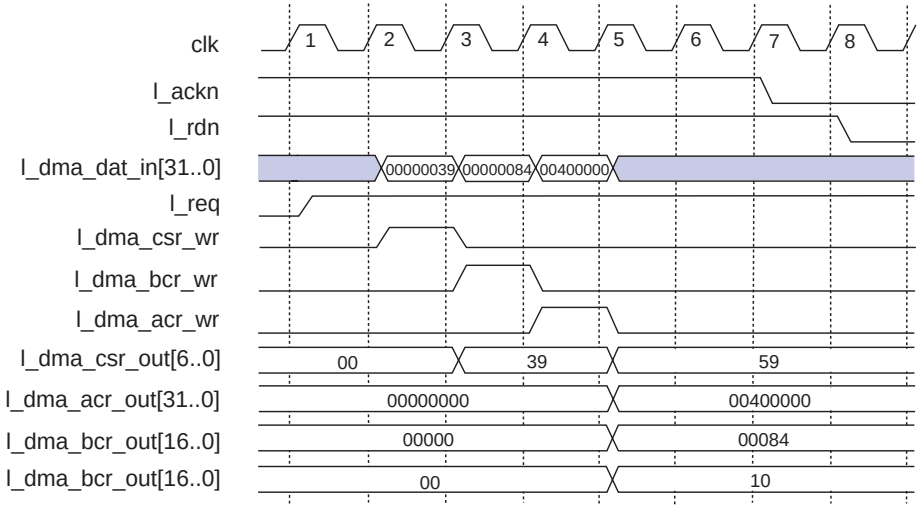
4. In clock four, local logic asserts `l_dma_acr_wr` while supplying data value in the `l_dma_dat_in[]` bus. This signal sequence writes a hexadecimal value of `00400000` into the `dma_acr` register. The `pci_a` function starts its PCI write operation at the the hexadecimal address of `00400000`.

5. In clock five, the write transaction to the `dma_bcr` and `dma_acr` take effect. Figure 20 shows the changes in the values on the `l_dma_bcr_out[16..0]` and `l_dma_acr_out[31..0]` buses. Figure 20 also shows the changes in values on the `l_dma_isr_out[4..0]` and `l_dma_csr_out[6..0]` buses, which set the `ad_loaded` and `dma_on` bits.

6. The `pci_a` function asserts `l_ackn`, indicating it is ready to accept data from the local side.

7. On the rising edge of clock nine, local logic begins to provide data on the `l_dat_in[31..0]` bus into the buffer.

***Figure 20. Local Side Initiated DMA Write Transaction***




## General Host Programming Guidelines

DMA transfers can be controlled by the host as well as the local logic. This section provides general programming guidelines—when the DMA is controlled by the host—and is divided into the following four tasks:

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

### *Initializing the pci_a Function*

To initialize the pci_a function:

1. Configure the pci_a-supported PCI bus configuration registers.
2. Configure the dma_csr register. See Table 32.

**Table 32. Initializing the pci_a Function**

| 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 perrn in the case of data parity errors, and the assertion of serrn in case of address parity errors. |
| 2 | BAR0: 0.0000 | | 0000.0011 | The value in the dma_csr enables both the interrupts and the DMA engine. |

### *DMA Operation*

To begin a DMA operation, perform the steps below:

1. 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.)
2. Load the dma_acr. (See "Internal Target Registers Memory Map" on page 43)
3. 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 (e.g., a microprocessor) is operating on the local side, this step may not be necessary. See Table 33.

4. At this point, the pci_a function generates a PCI interrupt (intan) to interrupt the controller due to byte counter expiration.

*Table 33. 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 step may involve several steps, e.g., setting-up the local address generator; or asserting l_req from the local side. |

### *Interrupt Service Operation*

To interrupt a service operation, perform the steps below:

1. 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 1 of "DMA Operation" on page 55.

   b. If the err_pend bit is high, indicating that the DMA operation was stopped due to an error, go to step 2 in "Clearing Error Bits" on page 57. Clear the error bit prior to continuing. See Table 34.

*Table 34. Interrupt Service Routine*

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

### *Clearing Error Bits*

To clear the error bits, perform the following steps:

1. Read the `dma_isr`. If the `err_pend` bit is active, go to step 2.
2. Configure the `dma_csr` by asserting the flush bit to clear the `ad_loaded` bit (bit 4 of the `dma_isr`).
3. Read the PCI bus configuration status register and determine which error is asserted (i.e., bit 15, 12, or 13).
4. Configure the `pci_a`-supported PCI status register and write a logic one to the appropriate error bit field. Writing a one to a bit in the status register clears the bit, allowing the designer to read the status register and write the same value to clear the error conditions.

## Applications

The pci_a function is ideal for add-in applications. Figure 21 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 21. Local Side Interface to an Intelligent Local-Side Host with a Shared Memory Bus*

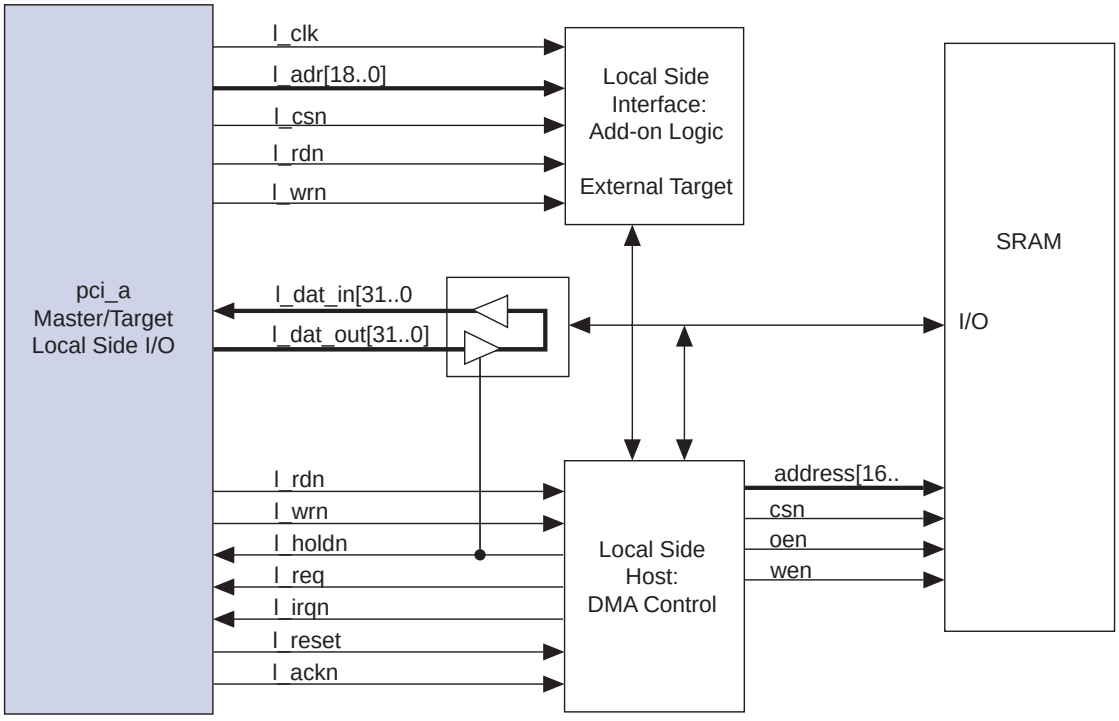
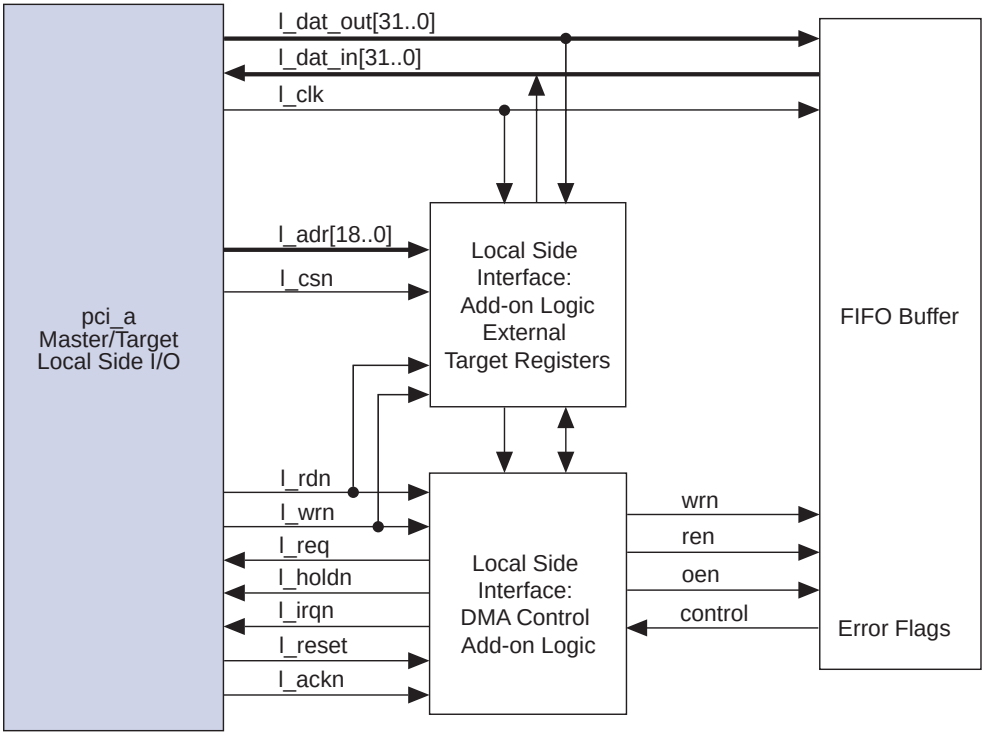



Figure 22 shows a typical pci_a connection to a dumb memory FIFO buffer. In this example, a target and a DMA control block are needed for access to the local side.

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 22 illustrates the process via the bidirectional signals going between the two control blocks.

*Figure 22. Local Side Interface to a Dumb FIFO Buffer*



# PCI SIG Protocol Checklists

Tables 35 through 42 list the applicable PCI SIG protocol requirements from the ***PCI Compliance Checklist, Revision 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, and table entries annotated with an em dash represent non-applicable PCI SIG requirements.

***Table 35. Component Configuration***

| 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? | | ✓ |
| 13 | If the device is multifunction, are configuration space accesses to unimplemented functions ignored? | | ✓ |

***Table 36. Component Configuration Space Summary (Part 1 of 2)***

| Location | Name | Required/Optional | N/A | Support |
|---|---|---|---|---|
| 00h-01h | Vendor ID | Required. | | ✓ |
| 02h-03h | Device ID | Required. | | ✓ |
| 04h-05h | Command | Required. | | ✓ |

*Table 36. Component Configuration Space Summary (Part 2 of 2)*

| Location | Name | Required/Optional | N/A | Support |
|---|---|---|---|---|
| 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 37. Device Control Summary*

| Location | Required/Optional | Yes | No |
|---|---|---|---|
| DC1 | When the command register is loaded with a 0000h, is the device/function logically disconnected from the PCI bus, with the exception of configuration accesses? (Devices in boot code path are exempt). | ✓ | |
| DC2 | Is the device/function disabled after the assertion of PCI `rstn`? (Devices in boot code are exempt.) | ✓ | |

*Table 38. 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 | `serrn` enable | Required if device/function has `serrn` 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 39. 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 `rstn` is asserted? | ✓ | |
| 4 | Are read/write bits reset to a 0 by writing a 1 to the bit? | ✓ | |

*Table 40. Status Register Summary (Part 1 of 2)*

| 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. | ✓ | | |

*Table 40. Status Register Summary (Part 2 of 2)*

| Bit | Name | Required/Optional | N/A | Target | Master |
|---|---|---|---|---|---|
| 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 `serrn`. | | | ✓ |
| 15 | Detected parity error | Required unless exempted per section 3.7.2. | | | ✓ |

*Table 41. Component Master Checklist (Part 1 of 2)*

| 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 irdyn when data is valid during a write transaction. (section 3.2.1) | ✓ | |
| 5 | IUT only transfers data when both `irdyn` and `trdyn` are asserted on the same rising clock edge. (section 3.2.1) | ✓ | |
| 6 | Once the IUT asserts `irdyn` it never changes `framen` until the current data phase completes. (section 3.2.1) | ✓ | |
| 7 | Once the IUT asserts `irdyn` it never changes `irdyn` until the current data phase completes. (section 3.2.1) | ✓ | |
| 8 | IUT never uses reserved burst ordering (`ad[1..0]` = “01”). (section 3.2.2) | ✓ | |
| 9 | IUT never uses reserved burst ordering (`ad[1..0]` = “11”). (section 3.2.2) | ✓ | |
| 10 | IUT always ignores configuration command unless `idsel` 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) | ✓ | |

*Table 41. Component Master Checklist (Part 2 of 2)*

| MP# | Requirement | Yes | No |
|---|---|---|---|
| 12 | The IUT's cben[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 cben[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 cben[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 cben[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 cben[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], cben[3..0], and par equals an even number. (section 3.8.1) | ✓ | |
| 30 | IUT always drives perrn (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 framen 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 42. 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 perrn 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 trdyn it never changes trdyn until the data phase completes. (section 3.2.1) | ✓ | |
| 6 | Once IUT has asserted trdyn it never changes devseln until the data phase completes. (section 3.2.1) | ✓ | |
| 7 | Once IUT has asserted trdyn it never changes stopn until the data phase completes. (section 3.2.1) | ✓ | |
| 8 | Once IUT has asserted stopn it never changes stopn until the data phase completes. (section 3.2.1) | ✓ | |
| 9 | Once IUT has asserted stopn it never changes trdyn until the data phase completes. (section 3.2.1) | ✓ | |
| 10 | Once IUT has asserted stopn it never changes devseln until the data phase completes. (section 3.2.1) | ✓ | |
| 11 | IUT only transfers data when both irdyn and trdyn are asserted on the same rising clock edge. (section 3.2.1) | ✓ | |
| 12 | IUT always asserts trdyn 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 idsel is asserted and ad[31..0] are “00”. (section 3.2.2) | ✓ | |
| 16 | IUT always disconnects after the first data phase when reserved burst mode is detected. (section 3.2.2) | — | |
| 17 | The IUT’s ad[31..0] lines are driven to stable values during every address and data phase. (section 3.2.4) | ✓ | |
| 18 | The IUT’s cben[3..0] 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 trdyn during turnaround cycle on a read. (section 3.3.1) | ✓ | |
| 20 | IUT always deasserts trdyn, stopn, and devseln 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 stopn the cycle immediately following framen being deasserted. (section 3.3.3.2.1) | ✓ | |

*Table 42. Component Target Checklist (Part 2 of 2)*

| MP# | Requirement | Yes | No |
|---|---|---|---|
| 23 | Once the IUT has asserted stopn it never deasserts stopn until framen is negated. (section 3.3.3.2.1) | ✓ | |
| 24 | IUT always deasserts trdyn before signaling target-abort. (section 3.3.3.2.1) | — | |
| 25 | IUT never deasserts stopn 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 devseln before any other response. (section 3.7.1) | ✓ | |
| 29 | Once IUT has asserted devseln it never deasserts devseln 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 par within one clock of cben[3..0] and ad[31..0] being driven. (section 3.8.1) | ✓ | |
| 32 | IUT always drives par such that the number of "1"s on ad[31..0], cben[3..0], and par equals an even number. (section 3.8.1) | ✓ | |

## PCI SIG Test Bench Summary

Tables 43 through 60 list the applicable PCI SIG test bench scenarios from the *PCI Compliance Checklist, Revision. 2.1.* A check mark in the yes column indicates that the pci_a function meets the requirement. Checklists not applicable to the Altera FLEX 10K pci_a function are not listed.

*Table 43. Test Scenario: 1.1 PCI Device Speed (as indicated by devsel) Tests (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. | ✓ | |
| 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: 1.1 PCI Device Speed (as indicated by devsel) Tests (Part 2 of 2)*

| # | Requirement | Yes | No |
|---|---|---|---|
| 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 44. Test Scenario: 1.2 PCI Bus Target Abort 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 45. Test Scenario: 1.3 PCI Bus Target 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. | ✓ | |

*Table 45. Test Scenario: 1.3 PCI Bus Target Retry Cycles (Part 2 of 2)*

| # | Requirement | Yes | No |
|---|---|---|---|
| 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: 1.4 PCI Bus Single 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 47. Test Scenario: 1.5 PCI Bus Single Data Phase Disconnect 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. | ✓ | |

*Table 47. Test Scenario: 1.5 PCI Bus Single Data Phase Disconnect Cycles (Part 2 of 2)*

| # | Requirement | Yes | No |
|---|---|---|---|
| 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 48. 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 49. Test Scenario: 1.7 PCI Bus Multi-Data Phase Disconnect 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. | ✓ | |

*Table 49. Test Scenario: 1.7 PCI Bus Multi-Data Phase Disconnect Cycles (Part 2 of 2)*

| # | Requirement | Yes | No |
|---|---|---|---|
| 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 50. Test Scenario: 1.8 PCI Bus Multi-Data Phase & trdyn Cycles*

| # | Requirement | Yes | No |
|---|---|---|---|
| 1 | Verify that data is written to primary target when `trdy`nis released after second rising clock edge and asserted on third rising clock edge after `framen`. | ✓ | |
| 2 | Verify that data is read from primary target when `trdyn` is released after second rising clock edge and asserted on third rising clock edge after `framen`. | ✓ | |
| 3 | Verify that data is written to primary target when `trdyn` is released after third rising clock edge and asserted on fourth rising clock edge after `framen`. | ✓ | |
| 4 | Verify that data is read from primary target when `trdyn` is released after third rising clock edge and asserted on fourth rising clock edge after `framen`. | ✓ | |
| 5 | Verify that data is written to primary target when `trdyn` is released after third rising clock edge and asserted on fifth rising clock edge after `framen`. | ✓ | |
| 6 | Verify that data is read from primary target when `trdyn` is released after third rising clock edge and asserted on fifth rising clock edge after `framen`. | ✓ | |
| 7 | Verify that data is written to primary target when `trdyn` is released after fourth rising clock edge and asserted on sixth rising clock edge after `framen`. | ✓ | |
| 8 | Verify that data is read from primary target when `trdyn` is released after fourth rising clock edge and asserted on sixth rising clock edge after `framen`. | ✓ | |
| 9 | Verify that data is written to primary target when `trdyn` alternately released for one clock cycle and asserted for one clock cycle after `framen`. | ✓ | |
| 10 | Verify that data is read from primary target when `trdyn` alternately released for one clock cycle and asserted for one clock cycle after `framen`. | ✓ | |
| 11 | Verify that data is written to primary target when `trdyn` alternately released for two clock cycles and asserted for two clock cycles after `framen`. | ✓ | |
| 12 | Verify that data is read from primary target when `trdyn` alternately released for two clock cycles and asserted for two clock cycles after `framen`. | ✓ | |

*Table 51. 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 perrn on IUT memory write. | ✓ | |
| 2 | Verify that perrn 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 52. 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 perrn on IUT multi-data phase memory write. | ✓ | |
| 2 | Verify that perrn 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 53. 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 54. Test Scenario: 1.13 PCI Bus Master Parking*

| # | Requirement | Yes | No |
|---|---|---|---|
| 1 | IUT drives ad[31..0] to stable values within eight PCI clocks of gntn. | ✓ | |
| 2 | IUT drives cben[3..0] to stable values within eight PCI clocks of gntn. | ✓ | |
| 3 | IUT drives par one clock cycle after IUT drives ad[31..0] | ✓ | |
| 4 | IUT tri-states ad[31..0] and cben[3..0] and par when gntn is released. | ✓ | |

*Table 55. Test Scenario: 1.14 PCI Bus Master Arbitration*

| # | Requirement | Yes | No |
|---|---|---|---|
| 1 | IUT completes transaction when deasserting gntn is coincident with asserting framen. | ✓ | |

*Table 56. 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 57. 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 `idsel` inactive. | ✓ | |

*Table 58. Test Scenario: 2.8 Target Receives Configuration Cycles with Address and Data Parity Errors*

| # | Requirement | Yes | No |
|---|---|---|---|
| 1 | IUT reports address parity error via `serrn` during configuration read/write cycles. | ✓ | |
| 2 | IUT reports data parity error via PERR during configuration write cycles. | ✓ | |

*Table 59. Test Scenario: 2.9 Target Receives Memory Cycles*

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

*Table 60. Test Scenario: 2.10 Target Receives Memory Cycles with Address and Data Parity Errors*

| # | Requirement | Yes | No |
|---|---|---|---|
| 1 | IUT reports address parity error via `serrn` 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 function include:


- PCI Special Interest Group. ***PCI Local Bus Specification. Revision 2.1***. Portland, Oregon: PCI Special Interest Group, June 1995.
- PCI Special Interest Group. ***PCI Compliance Checklist. Revision 2.1***. Portland, Oregon: PCI Special Interest Group, June 1995.
- Altera Corporation. ***1996 Data Book***. San Jose, California: Altera Corporation, June 1996.
- Institute of Electrical and Electronics Engineers, Inc.***IEEE Standard VHDL Language Reference Manual*** (ANSI/IEEE Std 1076-1993). New York: Institute of Electrical and Electronics Engineers, Inc., June 1994.


## Revision History

The information contained in the *PCI Master/Target MegaCore Function with DMA Data Sheet* version 3.02 supersedes information published in previous versions.

### Version 3.02

Figures 21 and 22 were updated in version 3.02 of the *PCI Master/Target MegaCore Function with DMA Data Sheet*.

### Version 3.01

Version 3.01 contains updated waveforms in Figures 11 and 12.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 544-7104
Literature Services:
(408) 544-7144
lit_req@altera.com