REAL

COMPLIANCE

FLEXIBILITY

PERFORMANCE

AVAILABILITY

The **Real-PCI**™

Design Guide V3.0

XILINX®

✖ XILINX®

The Xilinx logo shown above is a registered trademark of Xilinx, Inc.

The shadow X shown above is a trademark of Xilinx, Inc.

XILINX ®

LogiCORE™
PCI Design Guide
Version 3.0

LogiCORE/AllianceCORE Fax:	+1 408-626-6440
Xilinx Home Page URL:	www.xilinx.com
Technical Support URL:	support.xilinx.com
LogiCORE PCI Solutions URL:	www.xilinx.com/pci
Information & Feedback E-mail:	logicore@xilinx.com

September 25, 2002

Contents

Chapter 5   Target Design Tips

Chapter 6   Target Data Transfer and Control

Chapter 7   Target Data Phase Control

Chapter 8   Target Burst Transfers

Conventions

This manual uses the following conventions. An example illustrates each convention.

- `Courier font` denotes the following items:

    - Signals on PCI Bus side of the LogiCORE PCI Interface

      `FRAME_IO` (PCI Interface signal name)

      `FRAME#` (PCI Bus signal name)

    - Signals within the user application

      `BACK_UP, START`

    - Command line input and output

      `setenv XIL_MAP_LOC_CLOSED`

    - World Wide Web URLs

      `http://www.xilinx.com`

    - HDL pseudocode

      `assign question = to_be | !to_be;`

      `assign cannot = have_cake & eat_it;`

    - Design file names

      `pcim_top.v, pcim_top.vhd`

- **`Courier bold`** denotes the following items:

    - Signals on the user side of the LogiCORE PCI Interface

      `ADDR_VLD`

- Menu selections or button presses

  FILE -> OPEN

- *Italic font* denotes the following items:

  - Variables in statements which require user-supplied values

    ngdbuild *design_name*

  - References to other manuals

    See the *Libraries Guide* for more information.

  - Emphasis in text

    It is not a bug, it is a *feature*.

- Dark shading indicates items that are not supported or reserved:

SDONE_I	in/out	Snoop Done signal. Not Supported.

- Square brackets "[ ]" indicate an optional entry or a bus index:

  ngdbuild [*option_name*] *design_name*

  DATA[31:0]

- A vertical or horizontal ellipsis indicates repetitive material that has been omitted.

  A B C ... X Y Z

- The use of "fn(SIG1 . . . SIGn)" in an HDL pseudocode fragment should be interpreted as "combinational function of signals SIG1 through SIGn.

  SUM = fn(A, B, Cin);

- A vertical bar "|" separates items in a list of choices.

  OPTION = [enabled|disabled]

- The prefix "0x" or the suffix "h" indicate hexadecimal notation.

  A read of address 0x00110373 returned 45524943h.

- A "Q" on a signal means that it is registered; this is only used for PCI Bus signals that are delayed by one cycle. An "_N" means the signal is active low

  PERRQ_N is both registered and active low.

Chapter 1

Getting Started

Thank you for purchasing the LogiCORE PCI interface from Xilinx!

The Xilinx LogiCORE PCI interface is a fully verified, pre-implemented PCI Bus interface. This interface is available in 32-bit and 64-bit versions, with support for multiple Xilinx FPGA device families. It is designed to support both Verilog-HDL and VHDL. The design examples in this book are provided in Verilog.

This book is intended to serve as a reference for use during the design phase of a project using the Xilinx PCI interface. This book is comprehensive in nature; some portions may not apply to all designs.

For installation instructions and system requirements, refer to the *LogiCORE PCI Release Notes* which accompany the product.

Other Documentation

For more details on the LogiCORE PCI interface, refer to the following documents located on the Xilinx PCI Lounges, accessible from the www.xilinx.com/pci web site:

- *LogiCORE PCI Databook*

- *LogiCORE PCI Release Notes*

- *LogiCORE PCI Implementation Guide*

Further information is available in the Mindshare *PCI Systems Architecture* text, which is included in the Xilinx PCI Design Kit, and the *PCI Local Bus Specification*, which is available from the PCI Special Interest Group.

Technical Support

The fastest method for obtaining PCI specific technical support for the LogiCORE PCI interface is through the support.xilinx.com web site. Questions are routed to a team of engineers with specific expertise in using the LogiCORE PCI interface.

Xilinx will provide technical support for use of the LogiCORE product as described in the *LogiCORE PCI Design Guide* and the *LogiCORE PCI Implementation Guide*. Xilinx cannot guarantee timing, functionality, or support of the LogiCORE product for designs that do not follow these guidelines.

Chapter 2

Signal Description

Figure 2-1, "Top Level Block Diagram" shows how a typical user application is interfaced to the LogiCORE PCI interface. The interface signals are grouped into functional sections with the standard PCI Bus interface signals on the left-hand side of the interface symbol and all user interface signals on the right-hand side of the symbol.

Figure 2-1   Top Level Block Diagram

For HDL users, the file pcim_top.v or pcim_top.vhd represents this structure.

PCI Bus Interface Signals

Table 2-1, "PCI Bus Interface Signals," defines the interface signals
that comprise the PCI Local Bus. These signals appear on the left side
of Figure 2-2, "LogiCORE PCI Bus Interface Symbol".
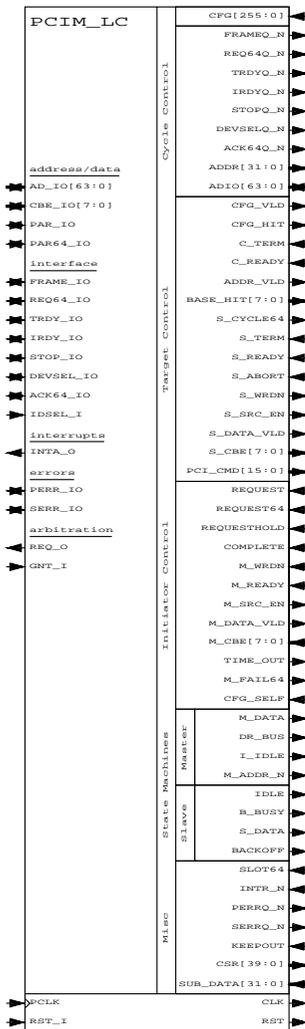
Figure 2-2   LogiCORE PCI Bus Interface Symbol

Note: Pin locations are device and package dependent. Refer to the
appropriate constraints file for specific device configurations.

Table 2-1   PCI Bus Interface Signals

Signal Name	Type	Functional Description
Address and Data Path		
AD_IO[31:0]	t/s	AD_IO[31:0] is a time-multiplexed address and data bus. Each bus transaction consists of an address phase followed by one or more data phases.
CBE_IO[3:0]	t/s	CBE_IO[3:0] is a time-multiplexed bus command and byte enable bus. Bus commands are asserted during an address phase on the bus. Byte enables are asserted during data phases.
PAR_IO	t/s	PAR_IO generates and checks even parity across AD_IO[31:0] and CBE_IO[3:0]. When the PCI interface is the source of an address or data, the interface generates even parity across AD_IO[31:0] and CBE_IO[3:0] and presents the result on PAR_IO one cycle after the values were presented on AD_IO[31:0] and CBE_IO[3:0]. When the interface receives an address or data, the interface checks for even parity across AD_IO[31:0] and CBE_IO[3:0] and compares it to PAR_IO one cycle later. Parity errors are reported via PERR_IO.

Table 2-1   PCI Bus Interface Signals (Continued)

Signal Name	Type	Functional Description
Transaction Control		
FRAME_IO	s/t/s active low	FRAME_IO is driven by an initiator to indicate a bus transaction. FRAME_IO is asserted for the duration of the operation and is deasserted during the last data phase to identify the end of the transaction. When operating as an initiator, the LogiCORE interface will only assert FRAME_IO when all of the following conditions are met: • GNT_I has been asserted for more than one cycle • IRDY_IO and FRAME_IO are deasserted, meaning the bus is idle • The bus master enable bit (**CSR2**) is set in the command register • The user application has asserted **REQUEST** or **REQUEST64** The LogiCORE interface will deassert FRAME_IO upon any of the following conditions: • The user application asserts **COMPLETE** • The interface receives a termination from the addressed target (retry, disconnect, or abort) • *Not* receiving a DEVSEL_IO assertion from the addressed target (master abort) • The internal latency timer has expired, if enabled, and the system arbiter is no longer asserting GNT_I • A 32-bit target responds to a 64-bit transfer request
DEVSEL_IO	s/t/s active low	DEVSEL_IO indicates that a target has decoded the address presented during the address phase and is claiming the transaction. This occurs when the address matches one of the Base Address Registers in the target.

Table 2-1   PCI Bus Interface Signals (Continued)

Signal Name	Type	Functional Description
TRDY_IO	s/t/s active low	TRDY_IO indicates that the target is ready to complete the current data phase. When TRDY_IO is asserted, the target is ready to transfer data. Data transfer occurs when both TRDY_IO and IRDY_IO are asserted on the bus.
IRDY_IO	s/t/s active low	IRDY_IO indicates that the initiator is ready to complete the current data phase. When IRDY_IO is asserted, the initiator is ready to transfer data. Data transfer occurs when both TRDY_IO and IRDY_IO are asserted on the bus.
STOP_IO	s/t/s active low	STOP_IO indicates that the target has requested to stop the current transaction. The target uses STOP_IO to signal a disconnect, retry, or target abort. The interface asserts STOP_IO under the control of the user application. However, the interface automatically asserts it during non-linear memory transactions, performing disconnect with data.
IDSEL_I	in	IDSEL_I indicates that the interface is the target of a configuration cycle.
Interrupts		
INTA_O	o/d active low	INTA_O indicates the LogiCORE PCI interface requests an interrupt. This may be disabled by setting the interrupt disable bit in the command register.

Table 2-1   PCI Bus Interface Signals (Continued)

Signal Name	Type	Functional Description
Error Signals		
PERR_IO	s/t/s active low	PERR_IO indicates that a parity error was detected while the LogiCORE PCI interface was the target of a write transfer or the initiator of a read transfer.
		Parity errors are reported two clock cycles after the data transaction appeared on the AD_IO and CBE_IO lines. Parity error reporting on PERR_IO is enabled by setting the report parity errors bit (**CSR6**) in the command register.
		Parity errors, except those during special cycles, are always reported in the status register (**CSR31**). Additionally, the initiator reports parity errors during a transaction when it was the bus master. The error is reported via the data parity error detected bit (**CSR24**) in the status register if the report parity errors bit (**CSR6**) is set in the command register.
SERR_IO	o/d active low	SERR_IO indicates that a parity error was detected during an address cycle, except during special cycles.
		SERR_IO is asserted on the third clock after FRAME_IO is first asserted. System errors are reported on the signaled system error bit (**CSR30**) in the status register if the SERR_IO enable bit (**CSR8**) and the report parity errors bit (**CSR6**) are set in the command register.
		SERR_IO is an open-drain output. Per the *PCI Local Bus Specification*, SERR_IO is not actively driven high after assertion.
Arbitration		
REQ_O	t/s active low	REQ_O indicates to the arbiter that the LogiCORE PCI initiator requests access to the bus. The initiator may only request the bus when it has been enabled by setting the bus master enable bit (**CSR2**) in the command register.

Table 2-1   PCI Bus Interface Signals (Continued)

Signal Name	Type	Functional Description
GNT_I	t/s active low	GNT_I indicates that the arbiter has granted the bus to the LogiCORE PCI initiator. If GNT_I is asserted and there is *not* a pending request, or the bus master enable bit is not set, then the interface performs bus parking.
System Signals		
RST_I	in active low	RST_I is the PCI Bus reset signal. This signal is used to bring PCI-specific registers, sequencers, and signals to a consistent state. Any time RST_I is asserted, all PCI output signals are three-stated.
PCLK	in	PCLK is the PCI Bus clock signal. This signal provides timing for all transactions on the PCI Bus and is an input to every PCI device. The frequency of PCLK may vary as allowed in the *PCI Local Bus Specification*.
64-bit Extension		
AD_IO[63:32]	t/s	AD_IO[63:32] is a time-multiplexed address and data bus. Each bus transaction consists of an address phase followed by one or more data phases. During address phases presented by 64-bit initiators, AD_IO[31:0] is driven with valid (reserved) values.
CBE_IO[7:4]	t/s	CBE_IO[7:4] is a time-multiplexed bus command and byte enable bus. During address phases presented by 64-bit initiators, CBE_IO[7:4] is driven with valid (reserved) values. Byte enables for the 64-bit extension are asserted during data phases.

Table 2-1   PCI Bus Interface Signals (Continued)

Signal Name	Type	Functional Description
PAR64_IO	t/s	PAR64_IO generates and checks even parity across AD_IO[63:32] and CBE_IO[7:4].
		When the PCI interface is the source of an address or data, the interface generates even parity across AD_IO[63:32] and CBE_IO[7:4] and presents the result on PAR64_IO one cycle after the values were presented on AD_IO[63:32] and CBE_IO[7:4].
		When the interface receives an address or data, the interface checks for even parity across AD_IO[63:32] and CBE_IO[7:4] and compares it to PAR64_IO one cycle later. Parity errors are reported via PERR_IO.
ACK64_IO	s/t/s active low	ACK64_IO indicates that a target has decoded the address presented during the address phase and is claiming the transaction as a 64-bit target. This occurs when the initiator makes a 64-bit transfer request using REQ64_IO, the address matches one of the Base Address Registers in the target, and the target is 64-bit enabled.

Table 2-1   PCI Bus Interface Signals (Continued)

Signal Name	Type	Functional Description
REQ64_IO	s/t/s active low	REQ64_IO is driven by the initiator to indicate a 64-bit bus transaction. REQ64_IO is asserted for the duration of the operation and is deasserted during the last data phase to identify the end of the transaction. Its behavior is similar to FRAME_IO. When operating as an initiator, the LogiCORE interface will only assert REQ64_IO when all of the following conditions are met: • GNT_I has been asserted for more than one cycle • IRDY_IO and FRAME_IO are deasserted, meaning the bus is idle • The bus master enable bit (**CSR2**) is set in the command register • The user application has asserted **REQUEST64** The LogiCORE interface will deassert REQ64_IO upon any of the following conditions: • The user application asserts **COMPLETE** • The interface receives a termination from the addressed target (retry, disconnect, or abort) • *Not* receiving a DEVSEL_IO assertion from the addressed target (master abort) • The internal latency timer has expired, if enabled, and the system arbiter is no longer asserting GNT_I • A 32-bit target responds to a 64-bit transfer request

in = input only signal

out = output only signal

t/s = bidirectional, three-state signal

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

o/d = open drain

User Interface Signals

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

Table 2-2, "User Interface Signals," describes the interface signals available to the user application. These signals appear on the right half of Figure 2-2, "LogiCORE PCI Bus Interface Symbol".

Table 2-2   User Interface Signals

Signal Name	Type	Functional Description
Configuration		
CFG[255:0]	in	**CFG[255:0]**, driven by a cfg module, configures the PCI interface.
Cycle Control		
FRAMEQ_N	out active low	**FRAMEQ_N** is a registered version of the PCI Bus FRAME_IO signal.
DEVSELQ_N	out active low	**DEVSELQ_N** s a registered version of the PCI Bus DEVSEL_IO signal.
IRDYQ_N	out active low	**IRDYQ_N** s a registered version of the PCI Bus IRDY_IO signal.
TRDYQ_N	out active low	**TRDYQ_N** s a registered version of the PCI Bus TRDY_IO signal.
STOPQ_N	out active low	**STOPQ_N** s a registered version of the PCI Bus STOP_IO signal.
Address and Data Path		
ADDR[31:0]	out	**ADDR[31:0]** holds PCI Bus addresses latched during address phases. It may be used for address decoding or for loading user address counters. The address is available in the cycle following the assertion of **ADDR_VLD** and remains stable until **ADDR_VLD** is asserted again.
ADIO[31:0]	t/s	**ADIO[31:0]** is a time multiplexed address and data bus. This bus must be driven using internal three-state buffers.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
Target Control		
`ADDR_VLD`	out	`ADDR_VLD` indicates the beginning of a *potential* address phase on the PCI Bus and that the address is available on the `ADIO[31:0]` internal bus. The latched address is presented on `ADDR[31:0]` one cycle later. Likewise, the PCI Bus command is presented on `S_CBE[3:0]` and latched, decoded, and presented on `PCI_CMD[15:0]`. `ADDR_VLD` is active only during potential target operations. It is not asserted during address phases that result from LogiCORE initiator activity.
`CFG_VLD`	out	`CFG_VLD` indicates the beginning of a *potential* configuration cycle. This signal is similar in nature to `ADDR_VLD` but is further qualified by IDSEL_I.
`S_DATA_VLD`	out	`S_DATA_VLD` indicates that a data transaction has occurred on the PCI Bus while the PCI interface is a target. `S_DATA_VLD` is asserted on the clock cycle after data transfer occurs on the PCI Bus and the target state machine is in the `S_DATA` state. When receiving data, `S_DATA_VLD` also indicates that the data is available on the `ADIO` bus. When sourcing data, `S_DATA_VLD` simply indicates successful data transfer.
`S_SRC_EN`	out	`S_SRC_EN` is an enable signal used to increment a data pointer when the interface is the source of data in a target burst read.
`S_WRDN`	out	`S_WRDN` indicates the data transfer direction during target transactions. During target writes, `S_WRDN` is asserted; during target reads, `S_WRDN` is deasserted.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
`PCI_CMD[15:0]`	out	`PCI_CMD[15:0]` indicates the current decoded and latched PCI Bus operation. This bus is a fully decoded (one hot) version of the current PCI Bus command. The command is captured during the address phase and remains stable until the next address phase. X8180
`S_CBE[3:0]`	out	`S_CBE[3:0]` indicates the current PCI Bus command or byte enables for a target access. Note that byte enables are active low.
`BASE_HIT[7:0]`	out	`BASE_HIT[7:0]` indicates that one of the Base Address Registers has decoded and matched an address. The bus is one-hot encoded as indicated below. The `BASE_HIT` signals are active for one clock cycle, the cycle preceding the `S_DATA` state. X8294
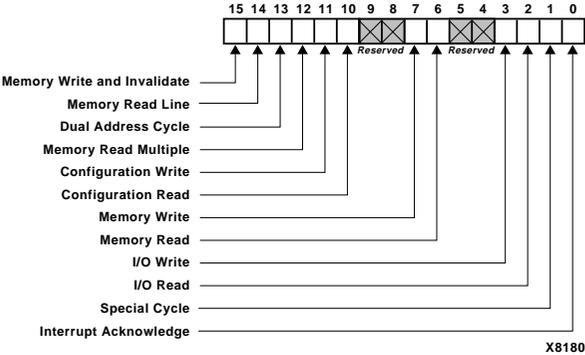
Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
CFG_HIT	out	CFG_HIT indicates the start of a valid configuration cycle. The CFG_HIT signal is active for one clock cycle, the cycle preceding the S_DATA state. This signal is similar in nature to the BASE_HIT signals.
C_READY	in	C_READY signals that the user application is ready to transfer configuration data. This is one of the signals which controls TRDY_IO. For most applications, C_READY should always be asserted. The exceptions are applications that require access to user configuration space.
C_TERM	in	C_TERM signals that the user application is terminating the transfer of configuration data. This is one of the signals which controls STOP_IO. For most applications C_TERM should always be asserted. The exceptions are applications that require wait states when accessing user configuration space.
S_READY	in	S_READY signals that the user application is ready to transfer data. This is one of the signals which controls TRDY_IO. If the user application is not ready to transfer data, S_READY should be delayed until the application is ready to support a sustained burst transfer. Do not deassert S_READY in the middle of a transfer.
S_TERM	in	S_TERM signals that the user application is terminating the transfer of data. This is one of the signals which controls STOP_IO.
S_ABORT	in	S_ABORT is used to signal a serious error condition which requires the current transaction to stop. S_ABORT should be used to signal an address overrun during a burst transfer or an unaligned 64-bit access.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
Initiator Control		
`REQUEST`	in	`REQUEST` is used to request a PCI initiator transaction. Assertion of `REQUEST` causes the PCI interface to assert REQ_O if the bus master enable bit (`CSR2`) is set in the command register. This bit is cleared at reset.
`REQUESTHOLD`	in	`REQUESTHOLD` is used to force an extended bus request. Assertion of `REQUESTHOLD` causes the PCI interface to assert REQ_O if the bus master enable bit (`CSR2`) is set in the command register. Unlike the `REQUEST` signal, `REQUESTHOLD` is not an input to the LogiCORE initiator state machine. `REQUESTHOLD` is intended to allow applications with very demanding bandwidth requirements to keep REQ_O asserted as long as possible. Do not assert `REQUESTHOLD` unless a transfer has been requested and is in progress. Otherwise, the arbiter may identify the LogiCORE interface as a broken master.
`M_CBE[3:0]`	in	`M_CBE[3:0]` is used by the user application to drive command and byte enables during initiator transactions. Bus commands should be presented during the assertion of `M_ADDR_N`, and byte enables should be presented during the `M_DATA` state. Note that byte enables are active low.
`M_WRDN`	in	`M_WRDN` indicates the data transfer direction during initiator transactions. During initiator writes, assert `M_WRDN`; during initiator reads, deassert `M_WRDN`.
`COMPLETE`	in	`COMPLETE` signals the initiator state machine to finish the current transaction. Once asserted, `COMPLETE` must remain asserted until the state machine leaves the `M_DATA` state. This is one of this signals which controls FRAME_IO and IRDY_IO.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
M_READY	in	M_READY signals that the user application is ready to transfer data. If deasserted, wait states are inserted. This is one of the signals which controls IRDY_IO. If the user application is not ready to transfer data, M_READY should be delayed until the application is ready to support a sustained burst transfer. Do not deassert M_READY in the middle of a transfer.
M_DATA_VLD	out	M_DATA_VLD indicates that a data transaction has occurred on the PCI Bus while the PCI interface is an initiator. M_DATA_VLD is asserted on the clock cycle after data transfer occurs on the PCI Bus and the initiator state machine is in the M_DATA state. When receiving data, M_DATA_VLD also indicates that the data is available on the ADIO bus. When sourcing data, M_DATA_VLD simply indicates successful data transfer.
M_SRC_EN	out	M_SRC_EN is an enable signal used to increment a data pointer when the interface is the source of data in an initiator burst write.
CFG_SELF	in	CFG_SELF indicates to the PCI interface that it is allowed to issue a configuration cycle to itself. The assertion of this signal overrides the bus master enable bit (CSR2) and modifies the internal data path. It is intended for use *only* in host bridge applications.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
`TIME_OUT`	out	`TIME_OUT` indicates that the internal latency timer has expired and that the user application has exceeded the maximum number of clock cycles allowed by the system configuration software.
		If the latency timer expires while the system arbiter is still asserting `GNT_I`, the operation will continue until either the operation completes, or the arbiter deasserts `GNT_I`. If the latency timer expires and the system arbiter has already deasserted `GNT_I`, then the operation terminates. The user application should handle this termination like any other target termination.
		Note: The default latency timer value is 0, indicating immediate time-out. Ensure that the system configuration software writes a sufficiently large value in the latency timer register to allow the desired transfer size.
State Machine - Initiator		
`M_DATA`	out	`M_DATA` indicates that the initiator is in the data transfer state. The `M_DATA` state will occur after the assertion of `M_ADDR_N` unless a single cycle assertion of `GNT_I` occurs.
`DR_BUS`	out	`DR_BUS` indicates that the bus is parked on the Logi-CORE interface. The LogiCORE initiator is then responsible for driving the `AD_IO[31:0]` bus, the `CBE_IO[3:0]` bus, and the `PAR_IO` signal to prevent these three-state bus signals from floating. The actual values driven on these lines are not important.
`M_ADDR_N`	out active low	`M_ADDR_N` indicates that the initiator is in the address state. During this time, the user application must drive a valid address on `ADIO` and a valid bus command on `M_CBE`.
		`M_ADDR_N` is asserted with a one clock cycle overlap with either the `I_IDLE` or `DR_BUS` states.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
`I_IDLE`	out	`I_IDLE` indicates that the initiator is in the idle state. The initiator is either not enabled, does not have an active request pending, or has not received `GNT_I` from the system arbiter.
		The state machine will always remain in either the `I_IDLE` or `DR_BUS` state when the bus master enable bit (`CSR2`) in the command register is reset.
State Machine - Target		
`IDLE`	out	`IDLE` indicates that the target is in the idle state.
`B_BUSY`	out	`B_BUSY` indicates that the PCI Bus is busy. An agent has started a transaction (`FRAME_IO` has been asserted) but the target state machine either has not yet finished decoding the address or has determined that it is not the target of the current operation.
`S_DATA`	out	`S_DATA` indicates that the target is in the data transfer state. The target has decoded the address and matched it to one of its Base Address Registers or a configuration operation is in progress. The target has accepted the request and will respond.
`BACKOFF`	out	`BACKOFF` indicates that the user application asserted `S_TERM` or `C_TERM` and the target state machine is waiting for the transaction to complete.
Miscellaneous Signals		
`PERRQ_N`	out active low	`PERRQ_N` is a registered version of the PCI Bus `PERR_IO` signal.
`SERRQ_N`	out active low	`SERRQ_N` is a registered version of the PCI Bus `SERR_IO` signal.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
INTR_N	in active low	INTR_N signals an interrupt request from the user application. The assertion of this signal generates an interrupt request on the PCI Bus unless the interrupt disable bit of the command register is set. Once the INTR_N signal is asserted, the user application must keep it asserted until the device driver clears the interrupt. This mechanism is implementation dependent.
KEEPOUT	in	KEEPOUT isolates the internal ADIO bus from the PCI LogiCORE interface. This allows the user application to perform data transfer over ADIO without interference. When using KEEPOUT, assert S_TERM and C_TERM, and deassert S_READY and C_READY to terminate all incoming transfer attempts with retry.
CSR[15:0]	out	CSR[15:0] provides access to the command register state bits. These bits are directly set or reset through the system configuration software. All values in the command register are either registered or read-only. Note: The bus master enable bit must be set in the command register before the initiator can access the PCI Bus. The I/O access enable bit and/or the memory access enable bit must be set in the command register before the target will respond.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
CSR[31:16]	out	CSR[31:16] provides access to the status register state bits. These are set automatically by the LogiCORE PCI interface. Individual status bits are reset by the system software by writing a '1' to the bit location to be reset. All values in the status register are either registered or read-only. Fast back to back transactions are not supported. 31 30 29 28 27 26 25 24 23 22 21 20 19 18 *Reserved* 16 Interrupt Status Capabilities List 66 MHz Capable Fast Back-to-Back Capable Data Parity Error Detected Device Select Timing [10] = slow, [01] = medium Signaled Target Abort Received Target Abort Received Master Abort Signaled System Error Detected Parity Error X8295

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
`CSR[39:32]`	out	`CSR[39:32]` provides access to the transaction status signals. These are an extension of the standard command and status register bits and reflect the status of a PCI transaction. With the exception of "master abort", these status bits reflect any bus activity, as they are derived from registered copies of PCI Bus signals. It is important to note that `CSR[38:32]` are combinational outputs generated by the equations shown below. X8182
`SUB_DATA[31:0]`	in	`SUB_DATA[31:0]` performs one of two functions depending on the PCI interface configuration. If the interface is not configured to use external Subsystem ID, this input provides the CardBus CIS Pointer data. If the interface is configured to use external Subsystem ID, this input provides the Subsystem ID data.
System Signals		
`CLK`	out	`CLK` is the PCI Bus clock driven by a global clock buffer. Use this clock for all flip-flops that are synchronized to the PCI Bus clock.

The diagram shows bits 39 38 37 36 35 34 33 32 with the following equations:

Data Transfer
!IRDYQ_N * !TRDYQ_N

Transaction End
FRAMEQ_N * (!TRDYQ_N + !STOPQ_N)

Normal Termination
STOPQ_N * FRAMEQ_N * !TRDYQ_N

Target Termination
FRAMEQ_N * !STOPQ_N

Disconnect Without Data
TRDYQ_N * !STOPQ_N * !DEVSELQ_N

Disconnect With Data
!TRDYQ_N * !STOPQ_N * !DEVSELQ_N

Target Abort
DEVSELQ_N * !STOPQ_N

Master Abort

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
`RST`	out	`RST` is an inverted copy of the PCI Bus reset signal. This signal should be used as an asynchronous reset signal for the user application.
64-bit Extension		
`REQ64Q_N`	out active low	`REQ64Q_N` is a registered version of the PCI Bus REQ64_IO signal.
`ACK64Q_N`	out active low	`ACK64Q_N` s a registered version of the PCI Bus ACK64_IO signal.
`ADIO[63:32]`	t/s	`ADIO[63:32]` is a time multiplexed address and data bus. This bus must be driven using internal three-state buffers.
`S_CYCLE64`	out	`S_CYCLE64` indicates that the PCI interface is engaged in a 64-bit target transaction. This signal is asserted at the same time as `BASE_HIT` and remains asserted until the transaction is complete.
`S_CBE[7:4]`	out	`S_CBE[7:4]` indicates the current PCI Bus command or byte enables for a target access. Note that byte enables are active low.
`REQUEST64`	in	`REQUEST64` is used to request a 64-bit PCI initiator transaction. Assertion of `REQUEST64` causes the PCI interface to assert REQ_O if the bus master enable bit (`CSR2`) is set in the command register. This bit is cleared at reset.
`M_FAIL64`	out	`M_FAIL64` indicates that a 64-bit initiator transfer attempt has encountered a 32-bit target. In such situations, the initiator will transfer at most two 32-bit words before terminating the transfer. The `M_FAIL64` signal should be used to adjust the increment value (step size) of initiator address pointers.

Table 2-2   User Interface Signals (Continued)

Signal Name	Type	Functional Description
`SLOT64`	in	`SLOT64` is used to enable the 64-bit extension. Refer to the appropriate implementation guide for details.
`M_CBE[7:4]`	in	`M_CBE[7:4]` is used by the initiator to drive byte enables during initiator transactions. Note that byte enables are active low.

Special Requirements

In general, the signals which interface the LogiCORE PCI interface to the user application may be connected as required by the application. However, certain signals have special connectivity requirements. To ensure a successful design, follow the design rules listed below.

- Any output signals from the LogiCORE PCI interface may remain unconnected if they are unused.

- The user application must connect to the full width of the internal `ADIO` address and data bus. Do not leave `ADIO` unconnected.

- All bits of the LogiCORE PCI interface configuration bus, `CFG`, must be driven by power or ground.

- Most input signals to the LogiCORE PCI interface that are unused or disabled may be connected to power or ground, as appropriate. However, the following signals must never be tied to power or ground: `SLOT64`, `S_TERM`, `S_READY`, `S_ABORT`, `COMPLETE`, `M_READY`, and `M_WRDN`.

Note: The fourth rule is of most importance. The first three design rules are trivial and do not require much attention during the design of a user application.

Signals are often explicitly tied to power or ground through assignment statements, but can "accidentally" be tied to power or ground through optimization of the driving logic.

In cases where the user application would normally assign one of these signals to power or ground, drive the signal from the output of a flip-flop.

For example, in place of:

```
assign WHATEVER = 1'b0;
```

Use the following:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized
   if (RST) WHATEVER = 1'b1;
   else WHATEVER = 1'b0;
end
```

The above requirements stem from the fact that certain pieces of logic inside the LogiCORE PCI interface, which are connected to these signals, must not be optimized or reduced. Following the rules presented above will prevent such undesired optimization.

Chapter 3

General Design Guidelines

This chapter describes the steps required to turn a LogiCORE PCI interface into a fully-functioning design integrated with user application logic. A target-only design does not require any of the initiator steps. However, an initiator always requires the target interface. The burst support steps may require four separate sub-steps: read and write operations for both target and initiator. Follow the logic design guidelines in this manual carefully.

Design Steps

- Configure the Base Address Register(s). Read the "Customizing the LogiCORE PCI Interface" chapter of this guide.

- Configure the contents of the Configuration Space Header ROM. This is covered in the "Customizing the LogiCORE PCI Interface" chapter of this guide.

- Configure the PCI interface options, also covered in the "Customizing the LogiCORE PCI Interface" chapter of this guide.

Target Designs

- Build an interface to read and write locations in the user application. Read the "Target Data Transfer and Control" chapter of this guide.

- Create logic to signal various target termination conditions if required by the user application. Read the "Target Data Phase Control" chapter of this guide.

- Read the "Target 64-bit Extension" chapter of this guide if the target uses the 64-bit extension. Read the "Target Only Designs" chapter of this guide if the design is target only.

Initiator Designs

- Build an initiator control state machine and the required support logic, and an interface to read and write locations in the user application. Read the "Initiator Data Transfer and Control" chapter of this guide.

- Create logic to control initiator data phases. See the "Initiator Data Phase Control" chapter of this guide.

- Read the "Initiator 64-bit Extension" chapter of this guide if the initiator uses the 64-bit extension.

Burst Designs

- Provide pipelined data sources which correctly respond to various target and initiator termination conditions, and build an address counter. Read the "Target Burst Transfers" chapter of this guide and the "Initiator Burst Transfers" chapter of this guide.

- Build FIFOs for the specific application, if required.

Advanced Designs

- Implement configuration space registers to support additional features and capabilities, if required. Read the "Other Bus Cycles" chapter of this guide.

- Modify the initiator or target logic to support special bus commands, if desired. Read the "Other Bus Cycles" chapter of this guide.

- Modify design to support operation as a host bridge, if required. Read the "Other Bus Cycles" chapter of this guide.

Know the Degree of Difficulty

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

Table 3-1, "Degree of Difficulty for Various PCI Implementations" indicates the degree of difficult in implementing various types of PCI designs. The degree of difficulty is sharply influenced by:

- Maximum system clock frequency

- Targeted device architecture

- Nature of the user application

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

Carefully review Table 3-1, "Degree of Difficulty for Various PCI Implementations" to determine the level of difficulty associated with different designs.

Table 3-1   Degree of Difficulty for Various PCI Implementations

Device Family	System Clock Frequency	Difficulty
Virtex Spartan-II	33 MHz	Easy
	66 MHz	Difficult
Virtex-E Spartan-IIE	33 MHz	Easy
	66 MHz	Moderate
Virtex-II Virtex-II Pro	33 MHz	Easy
	66 MHz	Moderate

Understand Signal Pipelining

In order to meet the stringent PCI performance requirements, the LogiCORE interface pipelines all of the bus control signals and the data path. Consequently, some signals must be presented up to two clock cycles before they appear on the PCI Bus. Likewise, arriving signals are captured and available to the user application one cycle after they appear on the PCI Bus. Figure 3-1, "Signal Pipeline Delay" provides some basic guidelines on how the LogiCORE interface is pipelined.
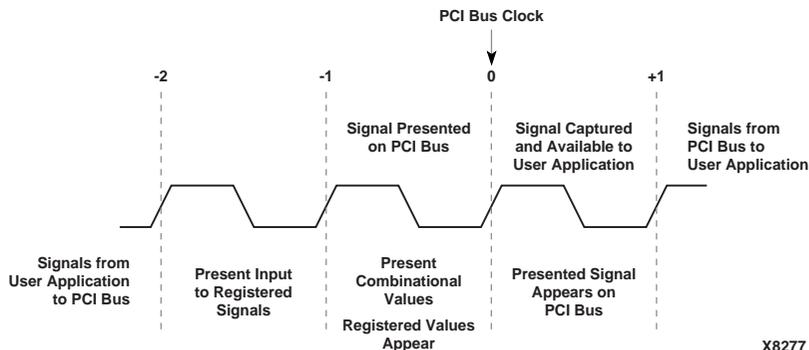
Figure 3-1   Signal Pipeline Delay

When the PCI interface receives a signal, it is captured in an input flip-flop to guarantee the setup time required by the *PCI Local Bus Specification*. The signal is available to the user application one clock cycle after it appears on the PCI Bus. For example, data is captured in input flip-flops and becomes available on the `ADIO` internal bus one cycle after it appeared on the PCI Bus. Signals like `M_DATA_VLD` and `S_DATA_VLD` signal the user application to grab the value from the `ADIO` bus.

When the user application is sending a combinational signal, the signal must be presented one cycle before it is to appear on the PCI Bus. Most of the outputs connected to the PCI Bus originate from an output flip-flop to meet the clock-to-output specification. If the signal first originates from a register in the user application, then the register inputs must be presented two cycles before the signal is to appear on the PCI Bus.

Keep it Registered

The best method to simplify timing and increase system performance in an FPGA design is to keep everything registered. This means that all inputs and outputs from the user application should come from, or connect to, a flip-flop. While registering signals may not be possible for all paths, it simplifies timing analysis.

Recognize Timing Critical Signals

Watch the timing and loading on the signals listed below. Some of these signals are part of the critical timing path. The following list of signals are timing critical and may require special attention when used in the user application.

- **M_SRC_EN** and **S_SRC_EN** – these signals are combinational outputs of the interface and are the two most time critical signals passed to the user application.

- **ADDR_VLD** – this is a heavily loaded signal.

- **M_DATA_VLD** and **S_DATA_VLD** – these become heavily loaded in most user applications.

- **C_READY**, **M_READY**, and **S_READY** – these connect to the target and initiator state machine control logic through multiple layers of logic. Drive **C_READY**, **M_READY**, and **S_READY** from a flip-flop, if possible.

- **C_TERM** and **S_TERM** – these signals connect to the target state machine control logic through multiple layers of logic. Drive **C_TERM** and **S_TERM** from a flip-flop, if possible.

- **COMPLETE** – this signal drives critical logic in the initiator state machine. Pay special attention to reducing delay for this signal.

- **M_ADDR_N** – this signal connects to the output enables driving the initiator start address onto the **ADIO** internal bus.

Use Supported Design Flows

The LogiCORE PCI interface uses a variety of advanced software features to obtain PCI system performance. These advanced features may include using a "guide file" to direct the placement and routing of timing-critical logic.

A guide file is a fragment of a full design. This fragment guarantees the performance of timing-critical PCI control signals in the target and initiator state machine logic. Guide files are carefully hand-crafted to achieve maximum performance.

The PAR (place and route) program matches logic and routing in the final PCI design to logic and routing in the hand-crafted guide file. Individual logic blocks and nets are matched by their instance name

and connectivity. For this reason, it is very important that the names used in the final design match those used in the guide file. If the names do not match, then PAR will not be able to guide the full design. Consequently, PAR may not be able to achieve the required PCI performance.

To guarantee that critical instance names and net names match those used in the guide file, the design must be processed using a supported design flow. Refer to the *LogiCORE PCI Implementation Guide* for a list of supported design flows.

Make Only Allowed Modifications

The LogiCORE PCI interface is not user-modifiable. Do not make modifications as they may have adverse effects on system timing and PCI protocol compliance. All modifications to the LogiCORE PCI interface must be done via the web-based Configuration and Download Tool or by hand editing the configuration file.

Chapter 4

Customizing the LogiCORE PCI Interface

This chapter describes the how to modify the LogiCORE PCI Interface. Modifications to the `pcim_lc` module are not allowed and may cause designs to fail timing specifications.

Using the Web-Based PCI Configuration Tool

There are two ways to modify the LogiCORE PCI interface. Select the method which is most appropriate.

The first method is to use the web-based PCI Configuration and Download Tool. This tool will automatically generate an appropriate HDL configuration file through a web-based application. It is available at:

`http://www.xilinx.com/products/logicore/pci`

Please see the on-line help for further details on how to use the PCI Configuration and Download Tool. This method is preferred when starting a new design.

The second method is to edit the configuration file by hand using a text editor. For small changes, this method is preferred. Users who do not have access to the web will also need to follow this method. The following sections describe how to modify the configuration file.

Editing the Configuration File

All recommended modifications can be made in the appropriate configuration file (`cfg.v` or `cfg.vhd`). The options selected in this module are communicated to the PCI interface through a 256-bit bus. Changes in the configuration file affect both the design and the functional simulation model.

Constant Declarations

A number of constant declarations are made at the beginning of the configuration file. Some of them are listed here; see the configuration file for a complete list. Use these in the subsequent sections to set the desired features.

```
`define MEMORY 1'b0
`define IO     1'b1


`define SPACE32 1'b0
`define SPACE64 1'b1

`define DISABLE  1'b0
`define ENABLE   1'b1

`define PREFETCH    1'b1
`define NOFETCH     1'b0
`define IO_PREFETCH 1'b1

`define TYPE00     2'b00
`define TYPE01     2'b01
`define TYPE10     2'b10
`define IO_TYPE    2'b11

`define SIZE2G     32'h8000_0000
.
. (see the configuration file for a complete list)
.
`define SIZE16     32'hffff_fff0
```

Device and Vendor ID

The Device ID is a unique identifier for the application. This field can be any value. Change this value for the application.

```
// Device ID and Vendor ID
assign CFG[151:120] = 32'h4062_10ee ;
```

The Vendor ID identifies the manufacturer of the device or application. Valid identifiers are assigned by the PCI Special Interest Group to guarantee that each identifier is unique. The value 10EEh, provided in the default configuration, is the Vendor ID for Xilinx. Enter a vendor identification number here. The value FFFFh is reserved.

Class Code and Revision ID

The Class Code identifies the general function of a device. The value, as provided in the default configuration, identifies the device as a generic co-processor function.

```
// Class Code and Revision ID
assign CFG[183:152] = 32'h0B40_0000 ;
```

The Class Code is divided into three byte-size fields as described in the *PCI Local Bus Specification*. The upper byte broadly identifies the type of function performed by the device.

The middle byte defines a sub-class that more specifically identifies the device's function. The sub-classes are defined in Appendix D of the *PCI Local Bus Specification*.

The lower byte defines a specific register-level programming interface (if any). This allows device-independent software to interact with the device.

The Revision ID indicates the revision of the device or application. It is an extension of the Device ID. Enter the values appropriate for the application.

Subsystem Vendor ID and Subsystem ID

The Subsystem Vendor ID further qualifies the manufacturer of the device or application. Enter a Subsystem Vendor ID here; typically, it is the same as the Vendor ID. Setting it to 0000h may cause issues with compliance testing.

The Subsystem ID can be set as desired to identify the device, revision, or other manufacturing data.

```
// Subsystem ID and SubVendor ID
assign CFG[215:184] = 32'h4062_10ee ;
```

Enabling the External Subsystem ID allows these fields to be supplied through the **SUB_DATA** bus. This may be used to load unique Subsystem Vendor IDs and Subsystem IDs. When this bit is set, the user applies the desired value to **SUB_DATA** and configuration reads will return this value.

```
// External Subsystem ID and Subvendor ID
assign CFG[114] = `DISABLE ;
```

CardBus CIS Pointer

The CardBus CIS Pointer is used in CardBus applications. Disabling the External Subsystem ID allows this field to be supplied through the `SUB_DATA` bus. When this bit is set, the user applies the desired value to `SUB_DATA` and configuration reads will return this value.

Base Address Registers

The LogiCORE PCI interface supports up to three BARs (Base Address Registers). The designer is free to use any BAR desired. Each BAR has several attributes. These attributes define:

- Whether the BAR is enabled. Disabling the BAR allows the optimization tools to delete the entire circuit.

- The size of the address space required. In the LogiCORE interface, the address space can be as small as 16 bytes, or as large as two gigabytes. For 80x86 systems, the maximum allowed I/O space is 256 bytes.

- The ability of memory space to be prefetched. The *PCI Local Bus Specification* defines memory as prefetchable if:

    - there are no side-effects on reads (i.e. data will not be destroyed by reading, as from a RAM).

    - byte write operations can be merged into a single double-word write, when applicable.

- Whether the address space is defined as memory or I/O. The BAR will only respond to commands that access the specified address space.

- The address space location "preference" of the device. The LogiCORE interface supports 32-bit address spaces for both memory and I/O.

- Whether the BAR address space is defined as 64-bit capable. This only applies to memory spaces.

Generally, memory spaces less than 4K in size should use a 4K block size, as recommended in the *PCI Local Bus Specification*. The maximum I/O space allowed is 256 bytes. Some machines may disable a card if it requests more than 256 bytes of contiguous I/O space.

```
// BAR0 -- 64-Bit Capable, 1 Mb Memory Space
assign CFG[0]       = `ENABLE ;
assign CFG[32:1]    = `SIZE1M ;
assign CFG[33]      = `PREFETCH ;
assign CFG[35:34]   = `TYPE00 ;
assign CFG[36]      = `MEMORY ;
assign CFG[241]     = `SPACE64 ;

// BAR1 -- 1 Mb Non-Prefetchable Memory Space
assign CFG[37]      = `ENABLE ;
assign CFG[69:38]   = `SIZE1M ;
assign CFG[70]      = `NOFETCH ;
assign CFG[72:71]   = `TYPE00 ;
assign CFG[73]      = `MEMORY ;
assign CFG[242]     = `SPACE32 ;

// BAR2 -- 16 Byte I/O Space
assign CFG[74]      = `ENABLE ;
assign CFG[106:75]  = `SIZE16 ;
assign CFG[107]     = `IO_PREFETCH ;
assign CFG[109:108] = `IO_TYPE ;
assign CFG[110]     = `IO ;
assign CFG[243]     = `SPACE32 ;
```

Max_Lat, Min_Gnt, and Latency Timer

These registers are used to specify the desired latency timer settings.
Max_Lat specifies how often the device needs to gain access to the
PCI Bus. Min_Gnt specifies how long the minimum burst period
should be, assuming a 33 MHz bus speed. The values for both regis-
ters are periods of time in units of a quarter microsecond.

```
// Max_Lat
assign CFG[231:224] = 8'h0f ;
// Min_Gnt
assign CFG[223:216] = 8'h0f ;
```

These registers are intended for use by the system software and do
not directly affect the operation of the LogiCORE PCI interface.

For applications that do not support bursting or that burst only two
words, the Latency Timer function can be disabled. This saves addi-
tional logic and routing resources.

```
// Latency Timer Enable
assign CFG[112] = `ENABLE ;
```

Interrupt Enable

This turns on the interrupt registers in the LogiCORE PCI interface. The interface only supports INTA_O.

```
// Interrupt Enable
assign CFG[113] = `ENABLE ;
```

Capabilities List

The LogiCORE PCI interface has the ability to implement a Capabilities List in configuration space. If the design requires a Capabilities List, enable this option and set the pointer to the desired address in configuration space. Otherwise set it to zero. Also enable the User Config Space option.

```
// Capabilities List Enable
assign CFG[116] = `DISABLE ;
// Capabilities List Pointer
assign CFG[239:232] = 8'h00 ;
// User Config Space Enable
assign CFG[118] = `DISABLE ;
```

Note: The User Config Space option may be enabled without using a Capabilities List to implement user-defined registers.

Interrupt Acknowledge

This bit enables the PCI interface to respond to interrupt acknowledge cycles as a target. Enabling this feature disables the last available Base Address Register.

```
// Interrupt Acknowledge
assign CFG[240] = `DISABLE ;
```

Reserved Settings

Several option settings are reserved for implementation specific features or for backwards compatibility. Do not modify these option settings. The following are reserved and should not be modified.

<div align="right">

Chapter 5

</div>

Target Design Tips

This chapter describes some of the design issues involved in building the target portion of an application. Before building the target portion of a user application, carefully consider what it must accomplish. Typically, there are multiple ways to achieve the same functionality.

Determine the Address Space Required

There is no central address decoding performed in a PCI system. Instead, the address decoding is distributed across the various agents present in the system.

A target design may request up to three separate address spaces of differing types and sizes. These selections are made when the base address registers in the PCI interface are configured for the user application. How should the user application allocate address space?

Where I/O space is required for PC legacy support, the use of an I/O base address register is unavoidable. In general, it is best to use base address registers located in memory space for several reasons. They support target burst, larger address spaces, prefetchable reads, and 64-bit data transfers.

User application design complexity increases with multiple base address registers, particularly if each address space responds to transactions in a different manner.

Determine the Bandwidth Required

The PCI interface supports both single and burst transfers. What type of bandwidth is required when other bus agents access the user application?

Currently, support for bursting to targets is poor in desktop PC chip sets. Additionally, the host bridge is involved in most, if not all, transactions (as either the initiator or the target). If the user application is intended for use in PC systems, higher bandwidth may be achieved by using single target transfers to set up an initiator burst transfer.

In embedded systems, it may be the case that the host bridge is used for configuration transactions only, and after configuration, agents on the bus burst data to each other in peer-to-peer transactions. In this case, it is critical to support target burst to achieve high bandwidth.

Where high bandwidth is desirable in a target design, posted writes and prefetched reads are two methods to increase bandwidth. These techniques are discussed in the *PCI Local Bus Specification*. Posted write buffers are generally easy to implement with FIFOs in a user application. Prefetched reads are more complicated in nature and require additional design considerations when the user application data source is not prefetchable.

Assemble the Design

The fundamentals of a target design are covered in the "Target Data Transfer and Control" chapter of this guide. The material covered in that chapter is sufficient for implementing non-burst transfers to registers and peripherals in the user application.

For more elaborate designs, the "Target Data Phase Control" chapter of this guide presents information on inserting target wait states and generating different types of target termination. This information is useful for both non-burst and burst designs.

For burst designs, the "Target Burst Transfers" chapter of this guide demonstrates the additional logic required to support target burst transfers. The "Target 64-bit Extension" chapter of this guide discusses the use of the 64-bit extension for increased performance.

Note that the LogiCORE PCI interface is both a target and an initiator. In designs where the initiator functions are unused, the initiator control signals must be "tied off" to benign values. The "Target Only Designs" chapter in this guide discusses how to do this.

Chapter 6

Target Data Transfer and Control

The purpose of this chapter is to demonstrate the logic required in the user application to generate the load and output enable signals for a typical target register.

In applications not using target burst transactions, data is usually transferred to and from registers in the user application. These registers are connected to control signals required for target data transfer and to any additional control and data path logic provided by the user. These registers may also connect to internal FIFOs or to I/O pins on the user application.
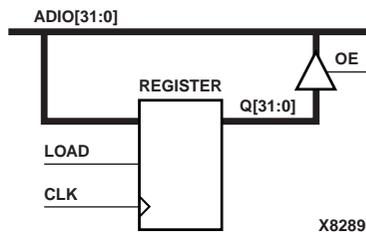
Figure 6-1   Example Target Register

A typical target register is interfaced as shown in Figure 6-1, "Example Target Register".

Target Interface Signals

The following signals control target data transfer to and from the PCI interface. For basic transfers, only a subset of these signals need be used. An example of a basic target design using this reduced subset is presented later in this chapter.

More elaborate designs involving non-deterministic target termination, target wait state insertion, or target burst are covered in later chapters. Note that references to inputs and outputs are made with respect to the user application.

- **BASE_HIT[7:0]** -- this input indicates that one of the base address registers recognizes that it is the target of a current PCI transaction. This is the first indicator to the user application that a target transaction is about to begin.

- **ADIO[31:0]** -- this bidirectional bus provides the means for data and address transfer to and from the PCI interface.

- **ADDR[31:0]** -- this input is a registered version of the PCI address provided by the PCI interface. It becomes valid in the cycle after **ADDR_VLD** is asserted, and remains valid through the entire transaction.

- **ADDR_VLD** -- this input indicates that a valid PCI address is available on the **ADIO** bus, and may be used as a clock enable by the user application to capture a copy of this address. This is particularly useful in target burst applications where a loadable counter must track the target address. In non-burst applications, the latched address present on the **ADDR** bus may suffice. Note, however, that the assertion of **ADDR_VLD** does not mean that the user application will be the selected target. The **ADDR_VLD** signal is asserted for a single cycle.

- **S_WRDN** -- this input indicates the direction of data transfer for the current target transaction. Logic high indicates that the user application is sinking data (i.e. target write). It is valid during the cycle **BASE_HIT** is asserted and is held through the entire transaction.

- **S_CBE[3:0]** -- this input is a registered version of the CBE_IO lines, and is delayed by one cycle. It indicates the PCI command and byte enables during a target transaction. This signal is used primarily for byte enable information, as the command is decoded and latched in **PCI_CMD** during the address phase of the transaction.

- **PCI_CMD[15:0]** -- this input is a decoded and latched version of the PCI command for the current bus transaction.

- **S_DATA_VLD** -- this input has two interpretations depending on the direction of data transfer. When the user application is

sinking data (target writes), **S_DATA_VLD** indicates that the user application should capture valid data from the **ADIO** bus. When the user application is sourcing data (target reads), **S_DATA_VLD** indicates that a data phase has completed on the PCI Bus.

- **S_SRC_EN** -- this input is only used during target burst reads. It indicates to the user application that the data source which drives output data onto the **ADIO** bus must provide the next piece of data. In most applications, this signals the user application to advance the data pointer for the data source that is providing the data.

- **CSR[39:0]** -- this input provides general status information about the PCI interface. The high eight bits provide status information about the current transfer. This status information is used primarily in target burst applications with non-prefetchable sources to determine if any associated address pointers must be "backed up".

- **S_READY** -- this output from the user application indicates that it is ready to transfer data, and can be used to insert wait states during the first data phase of a transaction. Together with **S_TERM**, it is also used to signal different types of target termination. It is important to note that the user application is prohibited from using **S_READY** to insert wait states after the first data phase.

- **S_TERM** -- this output from the user application indicates that data transfer should cease. It is also used with **S_READY** to signal different types of target termination.

- **S_ABORT** -- this output from the user application indicates that a serious (fatal) error condition has occurred and that the current transaction must stop.

The following signals are output by the target state machine in the PCI interface. These states are defined in Appendix B of the *PCI Local Bus Specification*.

- **IDLE** -- this input indicates that the target state machine is in the idle state and that there is no activity on the PCI Bus.

- **B_BUSY** -- this input indicates that the target state machine has recognized the beginning of a PCI Bus transaction. The target state machine will change to the **S_DATA** state if it determines that it is the target of the transaction.

- **S_DATA** -- this input indicates that the target state machine is in the data transfer state.

- **BACKOFF** -- this input indicates that the target state machine is waiting for a transaction to complete because the user application has asserted **S_TERM**.

Decoding Target Transactions

The user application is responsible for monitoring outputs from the PCI interface to respond to target transactions. The signals used in target transactions are active and available at different times. The most important signal is **BASE_HIT[x]**, which indicates that the PCI interface has claimed the current PCI transaction for base address register "**x**". It is asserted for a single cycle. The following logic decodes target reads and writes directed at base address register "**x**".

```verilog
always @(posedge CLK or posedge RST)
begin : decode
    if (RST)
    begin
        BAR_x_RD = 1'b0;
        BAR_x_WR = 1'b0;
    end
    else
    begin
        if (BASE_HIT[x])
        begin
            BAR_x_RD = !S_WRDN & OPTIONAL;
            BAR_x_WR = S_WRDN & OPTIONAL;
        end
        else if (!S_DATA)
        begin
            BAR_x_RD = 1'b0;
            BAR_x_WR = 1'b0;
        end
    end
end
```

```
assign OPTIONAL = fn(PCI_CMD[15:0]) & fn(ADDR[31:0]);
```

The optional term is for sub-decode and allows the address space allocated by a base address register to be mapped into multiple regions. If the user application performs sub-decoding of the address space, the above logic needs to be replicated to cover each region. Do not generate "holes" in the address space. If desired, the user application can also distinguish between different types of PCI read and write commands.

The last clause in the decoding block holds the decode asserted throughout the entire data transfer state. Again, **BASE_HIT[x]** is only active for a single clock cycle at the beginning of the transaction. Effectively, the above code describes a synchronous set/reset flip-flop with set dominant.

Target Writes

During a target write operation, data is captured from the **ADIO** bus to a data register in the user application by asserting the load input. The first step is to generate the BAR_x_WR signal as described in the previous section on decoding target transactions.

The critical gating signal is **S_DATA_VLD**. It is the final signal required to qualify the write operation. Consequently, the other signals can be decoded earlier and gated with **S_DATA_VLD**. The assignment for the load input of the register would then be:

```
assign LOAD = BAR_x_WR & S_DATA_VLD;
```

Decoding BAR_x_WR and registering it before the assertion of **S_DATA_VLD** allows more time for routing and reduces the number of logic levels from the critical input, **S_DATA_VLD**. If the user application supports byte-addressable registers, separate load signals should be generated for each byte in the register by further gating the expression shown above. The **S_CBE** signals are available for this purpose.

The time relationship is shown in the waveform of Figure 6-2, "Target Write Transaction". This waveform includes both PCI Bus signals and internal user application signals.

Figure 6-2   Target Write Transaction

Target Reads

During a target read operation, data from the user application is driven onto the **ADIO** bus. To do this, the user application must assert the output enable for the desired register. The first step is to generate the BAR_x_RD signal as described in the previous section on decoding target transactions. The assignment for the output enable would then be:

```
assign OE = BAR_x_RD & S_DATA;
```

Note: Ensure that all output enables are deasserted at system reset to avoid possible bus contention.

Do not qualify the output enable signal with the **S_CBE** signals even if the user application supports byte-addressable registers; drive the entire **ADIO** bus with valid data.

Although **S_DATA_VLD** is not used in the output enable logic, the user application may monitor this signal during target reads. The assertion of **S_DATA_VLD** during a target read indicates that the initi-

ator has acknowledged the data transfer. This is useful in designs where a data source must change state after it is read.

The time relationship is shown in the waveform of Figure 6-3, "Target Read Transaction". This waveform includes both PCI Bus signals and internal user application signals.

Figure 6-3   Target Read Transaction

Terminating Target Transactions

In general, PCI transactions may be terminated by the initiator or the target. In the specific case of non-burst target transactions, the PCI interface must terminate the transaction after the first data phase even if the initiator wishes to continue. Target terminations are controlled using the `S_READY`, `S_TERM`, and `S_ABORT` signals.

The `S_READY`, `S_TERM`, and `S_ABORT` signals must not be assigned static values. For timing reasons, these signals should be driven from the output of a flip-flop, although it is permitted to drive these signals from combinational logic.

Note: Never tie these signals to logic one or to logic zero.

The best way to achieve termination after the first data phase is to instruct the PCI interface to always disconnect with data:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
    if (RST) S_ABORT = 1'b1;
    else S_ABORT = 1'b0;
end


always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
    if (RST) S_READY = 1'b0;
    else S_READY = 1'b1;
end


always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_c
    if (RST) S_TERM = 1'b0;
    else S_TERM = 1'b1;
end
```

This will cause all target transactions to terminate after a single data phase. This termination behavior is exhibited in the target write and read transactions shown in Figure 6-2, "Target Write Transaction" and Figure 6-3, "Target Read Transaction".

This type of deterministic termination is sufficient for simple user application designs that do not support non-deterministic target termination, target wait state insertion, or target burst. More information about various target termination options is presented in later chapters.

Chapter 7

Target Data Phase Control

This chapter discusses the mechanism by which the user application can control various aspects of target transactions to accommodate its own ability to source or sink data.

Target initiated wait states allow a user application additional time before the first data transfer, and target initiated terminations allow the user application to limit the number of data phases in a transaction. These features are useful in both burst and non-burst target designs.

Control Modes

Data phase control is achieved using the **S_TERM** and **S_READY** signals. Combinations of the two control signals yield the following four modes:

- Wait -- the wait mode inserts wait states at the beginning of a PCI Bus transaction (holds off the first data phase) by delaying the assertion of TRDY_IO by the PCI interface.

- Normal -- the normal mode allows PCI Bus data phase(s) to complete without the insertion of extra wait states or termination by the target.

- Disconnect without data -- this mode terminates the current PCI Bus transaction without data transfer on the final data phase. A disconnect without data on the first data phase is equivalent to a retry.

- Disconnect with data -- this mode terminates the current PCI Bus transaction with data transfer on the final data phase.

The exact disconnect sequence is affected by whether or not the initiator also terminates the transaction. The PCI interface will automati-

cally generate the correct behavior. Note that the PCI interface will immediately disconnect with data if it receives a transaction using the non-linear addressing mode.

Table 7-1, "Data Phase Control Signals for Targets", shows the four modes of operation and the corresponding `S_TERM` and `S_READY` values.

Table 7-1   Data Phase Control Signals for Targets

Condition	Bus Signals	From User Application	
		S_TERM	S_READY
Wait	TRDY_IO = 1 DEVSEL_IO = 0 STOP_IO = 1	Low	Low
Normal	TRDY_IO = 0 DEVSEL_IO = 0 STOP_IO = 1	Low	High
Disconnect Without Data (Retry)	TRDY_IO = 1 DEVSEL_IO = 0 STOP_IO = 0	High	Low
Disconnect With Data	TRDY_IO = 0 DEVSEL_IO = 0 STOP_IO = 0	High	High

Changing from one mode to another must not be done in arbitrary sequence. In addition, the timing of mode transitions is critical to ensure precise control over the number of data phases that occur in a transaction, particularly when changing to a disconnect mode.

The permitted data phase control sequences for target designs using the PCI interface are shown in Figure 7-1, "Permitted Data Phase Control Sequences". The exact timing details are covered in subsequent sections of this chapter.
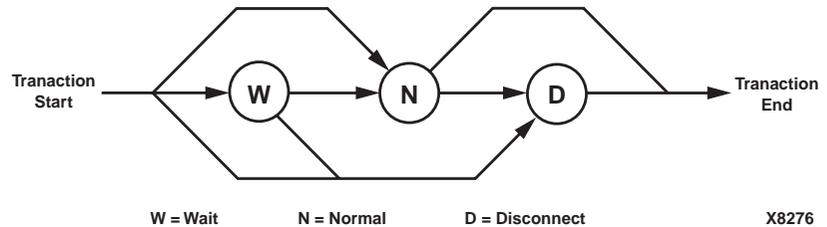
Figure 7-1   Permitted Data Phase Control Sequences

Note that the Wait mode cannot be used to insert wait states during arbitrary data phases in a transaction. It may only be used to delay the completion of the first data phase of a transaction. This is called initial latency. The target is required to complete the first data phase of a transaction within 16 clocks from the assertion of FRAME_IO. The user application is responsible for observing this requirement.

Note: During target state machine activity, do not violate the mode sequencing shown above. When the target state machine is inactive, **S_TERM** and **S_READY** are ignored by the PCI interface.

Control Pipeline

In order to meet the stringent PCI Bus performance requirements, the PCI interface pipelines all of the bus control signals and the data path. Consequently, the **S_TERM** and **S_READY** signals must be presented one cycle in advance of the desired effect. See Figure 7-2, "Control Signal Pipeline Delay" for a graphical interpretation.

PCI Bus Clock
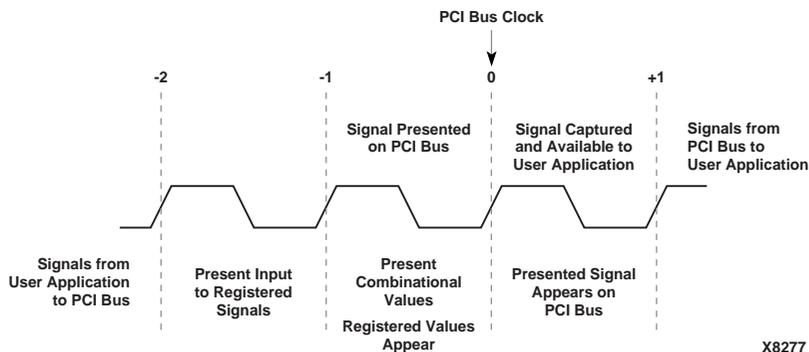
Figure 7-2   Control Signal Pipeline Delay

The signals **S_TERM** and **S_READY** connect to the target state machine through multiple levels of logic. For this reason, it is highly recommended that these signals are driven directly from the output of a flip-flop. This adds an extra cycle of latency.

Deterministic Control

Deterministic control refers to cases where the initial response of a user application to a target transaction does not depend on the parameters of the transaction. That is, the logic that drives **S_TERM** and **S_READY** does not use any information about the incoming target transaction to control the initial data phase. Such information includes, but is not limited to:

- Base address register number

- Target transaction address

- Target transaction command

In these cases, the user application knows how it will respond to a target transaction before the transaction occurs or selects control modes based on internal state information.

Deterministic control is easy to implement and simplifies timing considerations. While it is possible to feed transaction specific information back to the **S_TERM** and **S_READY** outputs in time for the first data phase, doing so results in driving **S_TERM** and **S_READY** with several levels of combinational logic. This is not recommended.

For deterministic control, the **S_TERM** and **S_READY** signals must be set properly before any transaction begins (at the time **BASE_HIT** is asserted). The following examples demonstrate **S_TERM** and **S_READY** generation for several common cases.

Example 1: Always Ready, Single Transfers

A simple user application that does not perform target burst transactions and is always ready to transfer data, as discussed in earlier chapters, can instruct the PCI interface to automatically disconnect with data on the first data phase.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
    if (RST) S_READY = 1'b0;
    else S_READY = 1'b1;
end

always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
    if (RST) S_TERM = 1'b0;
    else S_TERM = 1'b1;
end
```

An example of such an application is a bank of registers mapped in memory or I/O space. The correct control is achieved by assigning **S_READY** and **S_TERM** to values that result in disconnect with data, as shown in Table 7-1, "Data Phase Control Signals for Targets".

Example 2: Always Ready, Burst Transfers

Another very simple method of control may be used where the user application does perform target burst transactions and is always ready. For this case, the user application can instruct the PCI interface to proceed normally.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
    if (RST) S_READY = 1'b0;
    else S_READY = 1'b1;
```

```
end

always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
   if (RST) S_TERM = 1'b1;
   else S_TERM = 1'b0;
end
```

This allows the initiator of a transaction to burst data at full speed until it is done or its latency timer expires. The user application must be capable of sourcing or sinking data at full speed. The target may disconnect by adding logic to assert **S_TERM**. This behavior is commonly found in target designs that perform posted writes.

Example 3: Initial Latency

This is similar to Example 1, with the addition of initial latency. The initial latency may be fixed by a counter or may be variable and determined by other internal state. In cases where the initial latency is variable, the user application must not cause the PCI interface to violate the *PCI Local Bus Specification* limit on initial latency. The target is required to complete the first data phase of a transaction within 16 clocks from the assertion of FRAME_IO.

The following Verilog pseudocode demonstrates how to insert initial wait states:

```
always @(posedge CLK or posedge RST)
begin : bl_timer
   if (RST) TIMER = 4'h0;
   else if (ADDR_VLD) TIMER = BL_WAIT[3:0];
   else if (TIMER != 4'h0) TIMER = TIMER - 4'h1;
end

assign S_READY = (TIMER <= 4'h3);
assign S_TERM = (TIMER <= 4'h3);
```

The start of any transaction on the PCI Bus is marked by the falling edge of FRAME_IO. This information is needed to reload the initial wait timer. A registered version of FRAME_IO is available to the user application as **FRAMEQ_N**. However, this signal is heavily loaded

within the PCI interface. Fortunately, **ADDR_VLD** is asserted with the falling edge of **FRAMEQ_N** during target transactions, so **ADDR_VLD** may be used instead.

At the beginning of all transactions, the logic sets **S_TERM** and **S_READY** to the proper values in advance of target state machine signalling a pending target transaction. In non-target transactions, **S_TERM** and **S_READY** are simply ignored.

The latency is determined by the value present on BL_WAIT, and is expressed in cycles after FRAME_IO is asserted. Note that the timeout does not occur when the timer reaches zero. This is due to the latency in detecting the start of a transaction and the latency involved in propagating the **S_TERM** and **S_READY** signals through the PCI interface to the PCI Bus.

This example can easily be extended to allow multiple transfers, as shown in Example 2, by removing the assignment for **S_TERM** and adding the following:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_b
    if (RST) S_TERM = 1'b1;
    else S_TERM = 1'b0;
end
```

Bounded initial latency is useful in user application designs that access off chip registers or peripherals.

Example 4: Retries

In some user application designs, it may be necessary to retry target transactions if the data source or sink cannot be ready within the initial latency bounds given in the *PCI Local Bus Specification*. This situation may arise with very slow peripherals or if the user application implements delayed reads. In such cases, the initiator is bound by PCI protocol to retry the original transaction at a later time.

When the user application is ready to transfer data, **S_TERM** and **S_READY** may be generated as required. The following Verilog pseudocode provides an example, using the BLAT_RDY signal generated from Example 3:

```
always @(posedge CLK or posedge RST)
```

```
begin : keep_it_registered
   if (RST)
   begin
      S_READY = 1'b1;
      S_TERM = 1'b0;
   end
   else
   begin
      S_READY = RETRY ? 1'b0 : BLAT_RDY;
      S_TERM  = RETRY ? 1'b1 : BLAT_RDY;
   end
end


assign RETRY = fn(PERIPHERAL_STATE);
```

This code generates retries if the peripheral is not ready. If the peripheral is ready, the user application disconnects with data after several cycles of initial latency.

Care must be taken to ensure that RETRY is valid before the initial data phase of a target transaction and that it does not change during the transaction. One method to produce this result is to sample the peripheral state at the beginning of each PCI Bus transaction, in the same way that the initial latency timer is loaded in Example 3.

Other Possibilities

These examples are only a subset of the possible data phase control schemes. The ideas presented above may be combined in many ways to control the flow of data across the target interface so that the data rate is acceptable to the user application.

Note: Do not forget to account for the latency in **S_TERM** and **S_READY**. This latency is two cycles if **S_TERM** and **S_READY** are registered inside the user application, as is recommended. Otherwise, the latency is one cycle.

Non-Deterministic Control

Non-deterministic control refers to cases where the initial response of a user application to a target transaction depends on the parameters of the transaction. Such information includes, but is not limited to:

- Base address register number

- Target transaction address

- Target transaction command

In these cases, the user application does not know how to respond to a target transaction until the transaction has already started. The user application must feed transaction specific information back to the `S_TERM` and `S_READY` outputs in time for the first data phase. The difficulty in this is created by the latency in `S_TERM` and `S_READY` as discussed earlier in this chapter.

In order feed this information back in time to control the first data phase, `S_TERM` and `S_READY` must be driven with combinational logic and not from flip-flops. This is because the `S_TERM` and `S_READY` must be presented in the same cycle that a target transaction is detected (when a `BASE_HIT` signal is asserted). Although this is not recommended, it is possible if the logic is very fast.

For non-deterministic control, the `S_TERM` and `S_READY` signals must be presented as soon as one of the `BASE_HIT` signals is asserted. That is, `S_TERM` and `S_READY` will be combinational functions of `BASE_HIT` and other signals from the user application. The restrictions shown in Figure 7-1, "Permitted Data Phase Control Sequences" still apply.

Driving `S_READY` and `S_TERM` with combinational logic is highly discouraged for timing reasons. In cases where transaction specific information must be fed back to the PCI interface to control the initial data phase, an alternative method exists at the expense of performance.

The solution is to insert a single wait state at the beginning of all target transactions. The extra cycle in latency allows `S_TERM` and `S_READY` to be registered. Although performance is reduced, potential timing problems are avoided.

If, for performance reasons, the suggested method is not feasible, the following examples illustrate how to drive **S_TERM** and **S_READY** with combinational logic.

Example 5: Subdividing an Address Space

Consider a user application design where the address space assigned to a base address register is partitioned into two regions. The first region maps to a peripheral device which supports bursts but may require retries, and the second region maps to registers that require immediate disconnect with data. The data phase control signals for the peripheral are P_READY and P_TERM, while the data phase control signals for the registers are R_READY and R_TERM.

```
assign PERIPH_ADDR = fn(ADDR[31:0]);
assign RETRY = fn(PERIPHERAL_STATE);


assign P_READY = RETRY ? 1'b0 : 1'b1;
assign P_TERM = RETRY ? 1'b1 : 1'b0;
assign R_READY = 1'b1;
assign R_TERM = 1'b1;


assign S_READY = PERIPH_ADDR ? P_READY : R_READY;
assign S_TERM = PERIPH_ADDR ? P_TERM : R_TERM;
```

This logic can be greatly reduced. It is presented this form for clarity. As in Example 4, care must be taken to ensure that RETRY is valid during the cycle a **BASE_HIT** signal is asserted, and that it does not change during the transaction.

As an alternative approach, use an additional base address register (if available) instead of subdividing the address space of a single base address register.

Example 6: Multiple Base Address Registers

Consider a user application design with two base address registers, "x" and "y". The first maps to a peripheral device which supports bursts but may require retries, and the second maps to registers that require immediate disconnect with data. In concept, this is similar to Example 5 but is complicated by the fact that the **BASE_HIT** signals

are only valid for a single cycle, unlike the **ADDR** bus which is valid throughout a transaction.

```
always @(posedge CLK or posedge RST)
begin : similar_to_decode
    if (RST) BAR_x_IN_USE = 1'b0;
    else if (BASE_HIT[x]) BAR_x_IN_USE = 1'b1;
    else if (!S_DATA) BAR_x_IN_USE = 1'b0;
end


assign PERIPH_ADDR = BASE_HIT[x] | BAR_x_IN_USE;
```

The final multiplexing logic, as shown in Example 5, would use this version of PERIPH_ADDR as the select signal.

Target Abort

There is a special type of target termination called target abort. It informs the initiator that the target cannot perform the requested transaction.

A target abort is a serious error and signals that data may have been lost or corrupted. A target must respond to an initiator burst beyond its allocated address space with a target abort. Other fatal conditions detected by the user application should also result in target abort.

When the **S_ABORT** signal is asserted from the user application, the PCI interface automatically signals the target abort condition on the PCI Bus and sets the signaled target abort bit (CSR27) in the status register. The user application must continue to assert **S_ABORT** until the transaction is complete.

The **S_ABORT** signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic. When target aborts are not used, drive the **S_ABORT** signal as shown below:

```
always @(posedge CLK or posedge RST)
begin : not_using_abort
    if (RST) S_ABORT = 1'b1;
    else S_ABORT = 1'b0;
end
```

Chapter 8

Target Burst Transfers

Performing a single data transfer across the PCI Bus is the simplest type of transaction. However, because of the overhead of distributed address decoding, this wastes valuable bus bandwidth. The performance advantage in PCI is derived from burst transactions, where two or more data words are transferred during the transaction.

Building a user application that supports single target transfers is the easiest to design. Building a user application that supports target burst transfers is significantly more complex, but worth the effort, if maximum bandwidth is the goal.

Keeping Track of the Address Pointer

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

If the user application performs target burst transfers, then it must keep a local copy of the current address pointer and increment it after every successful data transfer cycle. Luckily, this counter can be small, depending on the address block size of the target (set in the base address register). The counter must be able to support bursts throughout the entire address range of the base address register.

For example, if the target decodes a 4 Kb block of memory space, then the address counter needs to keep track of addresses within the 4 Kb block. A 10-bit loadable binary counter will suffice. Although 4 Kb requires 12-bits to cover the address space, bits 0 and 1 will always equal zero for 32-bit transfers over the PCI Bus. If an initiator attempts to burst beyond the 4 Kb block boundary, then the target should issue a target abort, indicating that it is not able to perform the requested operation.

The upper 20 bits of the address pointer are a simple register, loaded during the address cycle of the transaction when **ADDR_VLD** is asserted. Likewise, the starting address within the address block is loaded into the 10-bit binary counter during the address cycle. The upper 20 bits, which are seldom required in the user application, can be eliminated. See Table 8-1, "Example Target Address Pointer".

Table 8-1   Example Target Address Pointer

31 12	11 2	1	0
20-bit address register (optional)	10-bit loadable binary counter	0	0

During target writes, the counter portion of the target address pointer should be incremented when **S_DATA_VLD** is asserted. During target reads, the address pointer should be incremented when **S_SRC_EN** is asserted. A full example of the count enable logic is presented later in this chapter.

If the user application supports multiple base address registers, a single address pointer should suffice, but the counter must support the largest block of address space. If one base address register supports a 4 Kb block while the other supports a 16 Mb block, then the counter must support the 16 Mb block, which requires a 22-bit loadable binary counter.

Sinking Data in Burst Transfers

During target writes, the PCI interface transfers burst data using a pipelined data path. The data valid signal, **S_DATA_VLD**, is used to advance the target address pointer (and any other data pointers in the user application logic). At the same time the target data pointer is advanced, the user application also captures valid data from the internal **ADIO** bus.

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

A target burst write is shown in the waveform of Figure 8-1, "Target Burst Write Transaction". This waveform includes both PCI Bus signals and internal user application signals.

Figure 8-1   Target Burst Write Transaction

Sourcing Data in Burst Transfers

During target reads, the PCI interface transfers burst data using a pipelined data path. The data source enable signal, **S_SRC_EN**, is used to advance the target address pointer (and any other data pointers in the user application logic). The result is that the user application drives new data onto the internal **ADIO** bus.

Internally, the PCI interface captures the data value provided by the user application on the **ADIO** bus and holds this value in the output flip-flops driving the AD_IO pins. The user application then presents the next data word on **ADIO**, instead of holding the previous data word until the current data phase completes. This approach results in higher data throughput.

A target burst read is shown in the waveform of Figure 8-2, "Target Burst Read Transaction". This waveform includes both PCI Bus signals and internal user application signals.
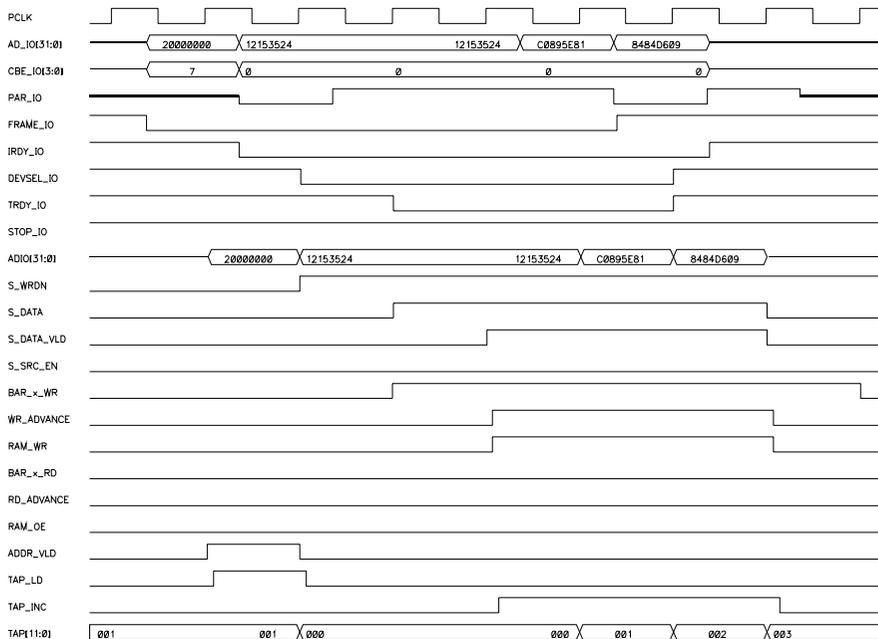
Figure 8-2   Target Burst Read Transaction

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

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

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

For non-prefetchable data sources, as is the case when a FIFO itself is the data source, pulling data out of the FIFO may be destructive. The

unused data must be restored so it is available for future use should it not be transferred. This may require decrementing internal counters or keeping a shadow copy of the previous data values.

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

One way to determine the number of times the target address pointer has been over-advanced during a burst read is to monitor the difference in the number of cycles **S_SRC_EN** and **S_DATA_VLD** have been asserted during a transaction. During target reads, the signal **S_DATA_VLD** represents the number of data phases that actually complete with data transfer.

Design Examples

The following design examples demonstrate the use of prefetchable and non-prefetchable data sources.

Example 1: Prefetchable Data Source

Prefetchable data sources, such as RAM and general purpose register files, do not exhibit "side effects" from reads (that is, the state is not altered). Figure 8-3, "Prefetchable Data Source" shows a target address pointer with a simple RAM as they might appear in a user application.
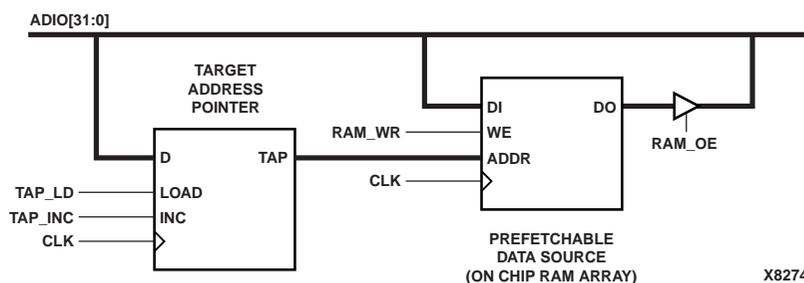
Figure 8-3   Prefetchable Data Source

Assume that the target address space, as specified in the base address register, is 4 Kb. After implementing the target transaction decode logic that generates `BAR_x_RD` and `BAR_x_WR`, the next step is to implement the target address pointer.

In the specific case of a 4 Kb address space, the two lowest bits of the address counter are always logic zero and do not need to be explicitly represented. The middle ten bits must be implemented as a loadable counter, and the high twenty bits are optional.

```
assign RD_ADVANCE = BAR_x_RD & S_SRC_EN;

assign WR_ADVANCE = BAR_x_WR & S_DATA_VLD;

assign TAP_INC = RD_ADVANCE | WR_ADVANCE;

assign TAP_LD = ADDR_VLD;


always @(posedge CLK or posedge RST)

begin : target_address_pointer

    if (RST) TAP = 10'h0;

    else if (TAP_LD) TAP = ADIO[11:2];

    else if (TAP_INC) TAP = TAP + 10'h1;

end
```

The remainder of the design is trivial. The target address pointer, `TAP`, is routed to the address input of the RAM array. The remaining control signals may be generated as shown below.

```
assign RAM_OE = BAR_x_RD & S_DATA;

assign RAM_WR = BAR_x_WR & S_DATA_VLD;


always @(posedge CLK or posedge RST)

begin : cannot_be_optimized_a

    if (RST) S_READY = 1'b0;

    else S_READY = 1'b1;

end


always @(posedge CLK or posedge RST)

begin : cannot_be_optimized_b

    if (RST) S_TERM = 1'b1;
```

```
            else S_TERM = 1'b0;
    end
```

The waveforms shown in Figure 8-1, "Target Burst Write Transaction",
and Figure 8-2, "Target Burst Read Transaction" are the result of this
example.

In this design, an initiator can perform burst data transfers to and
from the RAM throughout the entire address range of the target. If
the initiator attempts to continue a burst beyond the 4 Kb address
boundary, the target address pointer will wrap around and transfers
will continue from address zero. This behavior is undesirable; the
user application should signal a target abort and disable the RAM_WR
signal when a wrap around is detected.

Example 2: Non-Prefetchable Data Source

In Xilinx FPGAs, FIFOs to support PCI burst transfers are efficiently
implemented using the distributed on-chip RAM. In user applica-
tions employing FIFOs for target burst, the burst size (and aggregate
bandwidth) is limited by the depth of the FIFOs. Most FIFO designs
of 64 entries deep or less are feasible. Larger FIFOs are possible
depending on the FIFO configuration and the device speed grade.

Non-prefetchable data sources, such as FIFOs, exhibit "side effects"
from reads (that is, the state is altered or lost). Special care must be
taken during target burst reads so that state information is not lost.
The use of **S_SRC_EN** results in reading the data source ahead of the
actual transfer. Unless special precautions are taken, the data will be
lost.

Figure 8-4, "Non-Prefetchable Data Source" shows a FIFO suitable for
target burst reads in a user application. To present a concise example,
this example uses a single FIFO with both ports accessible through
the PCI interface. In practice, the best structure for most applications
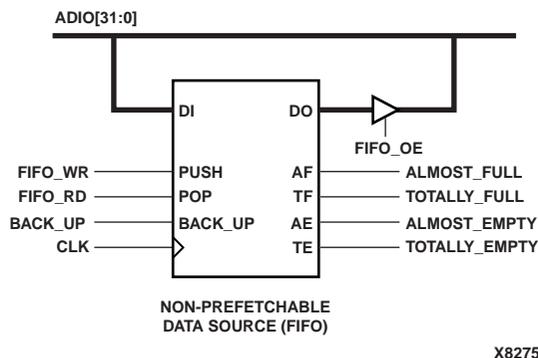is a dual-FIFO design with separate read and write FIFOs.

Figure 8-4   Non-Prefetchable Data Source

As in the previous example, the user application must decode the target access and generate appropriate read and write signals.

```
assign FIFO_OE = BAR_x_RD & S_DATA;
assign FIFO_WR = BAR_x_WR & S_DATA_VLD;
assign FIFO_RD = BAR_x_RD & S_SRC_EN;
```

The most crucial element is the FIFO itself. The FIFO must have an additional control signal to back up, or "undo" up to two reads. A typical FIFO implementation consists of a circular buffer implemented with RAM, a read pointer, and a write pointer.

The back up feature can be incorporated in a FIFO by making the actual depth two less than the maximum possible, and by using a bidirectional read pointer. This prevents new data entering the FIFO from overwriting data in the FIFO that may need to be restored.

The FIFO must also have a set of flags that provide FIFO status information. These flags, and their uses, are listed below:

- TE -- this flag indicates a totally empty condition. If the FIFO is totally empty at the beginning of a read transaction, the user application should respond with a retry.

- TF -- this flag indicates a totally full condition. If the FIFO is totally full at the beginning of a write transaction, the user application should respond with a retry.

- AE -- this flag indicates an almost empty condition. When the FIFO becomes almost empty during a read transaction, the user

application should signal a disconnect. The threshold for almost empty depends on the type of disconnect that is signalled (with or without data).

•   AF -- this flag indicates an almost full condition. When the FIFO becomes almost full during a write transaction, the user application should signal a disconnect. The threshold for almost full depends on the type of disconnect that is signalled (with or without data).

The logic for **S_TERM** and **S_READY** is then a function of the four FIFO flags and the signal **S_WRDN**. Again, the exact implementation depends on the version of the PCI interface and the desired type of disconnect, but all of the techniques demonstrated in earlier chapters apply.

To determine the number of times the FIFO must be backed up after a target read, the user application must include a small state machine to monitor the difference between anticipated transfers and actual transfers. When a target read has completed, the state machine should back up the FIFO if necessary.

```
assign ANTICIPATED = BAR_x_RD & S_SRC_EN & !TE;
assign ACTUAL = BAR_x_RD & S_DATA_VLD;


always @(posedge CLK or posedge RST)
begin : oops_counter
    if (RST) OOPS = 2'b00;
    else
    case({ANTICIPATED, ACTUAL, BACK_UP, OOPS})
        5'b00000: OOPS = 2'b00;
        5'b00001: OOPS = 2'b01;
        5'b00010: OOPS = 2'b10;
        5'b00011: OOPS = 2'b11;
        5'b00100: OOPS = 2'b00;
        5'b00101: OOPS = 2'b00;
        5'b00110: OOPS = 2'b01;
        5'b00111: OOPS = 2'b10;
        5'b01000: OOPS = 2'b00;
        5'b01001: OOPS = 2'b00;
```

```
          5'b01010: OOPS = 2'b01;
          5'b01011: OOPS = 2'b10;
          5'b01100: OOPS = 2'b00;
          5'b01101: OOPS = 2'b00;
          5'b01110: OOPS = 2'b00;
          5'b01111: OOPS = 2'b01;
          5'b10000: OOPS = 2'b01;
          5'b10001: OOPS = 2'b10;
          5'b10010: OOPS = 2'b11;
          5'b10011: OOPS = 2'b11;
          5'b10100: OOPS = 2'b00;
          5'b10101: OOPS = 2'b01;
          5'b10110: OOPS = 2'b10;
          5'b10111: OOPS = 2'b11;
          5'b11000: OOPS = 2'b00;
          5'b11001: OOPS = 2'b01;
          5'b11010: OOPS = 2'b10;
          5'b11011: OOPS = 2'b11;
          5'b11100: OOPS = 2'b00;
          5'b11101: OOPS = 2'b00;
          5'b11110: OOPS = 2'b01;
          5'b11111: OOPS = 2'b10;
          default : OOPS = 2'b00;
      endcase
   end


   assign BACK_UP = (| OOPS) & !S_DATA;
```

This state machine describes a two bit saturating up/down counter with one increment input and two decrement inputs. As required, it tracks the number of actual transfers versus the number of anticipated transfers. The BACK_UP signal is asserted when the transfer is over (**S_DATA** is deasserted) and the OOPS counter is non-zero. This backs up the FIFO and decrements the OOPS counter.

A three data phase target burst read using a FIFO like the one shown in Figure 8-4, "Non-Prefetchable Data Source" and an OOPS counter is shown in the waveform of Figure 8-5, "Burst Read of Target FIFO". This waveform includes both PCI Bus signals and internal user application signals.
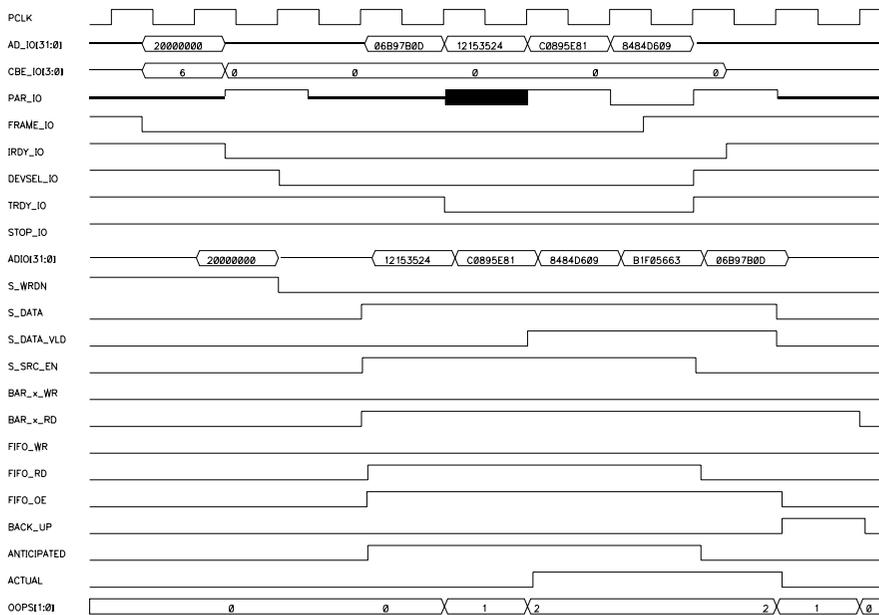

Figure 8-5   Burst Read of Target FIFO

Chapter 9

Target 64-bit Extension

This chapter provides additional details about the target 64-bit extension. Implementation of a user application which is a 64-bit target is almost identical to the implementation of a 32-bit version. The notable exception is that the data path is twice as wide. The target control signals behave the same regardless of the data path width.

Note that 64-bit transfers only apply to memory spaces. Other spaces do not support this capability. In order for a 64-bit implementation of the LogiCORE PCI interface to support 64-bit transfers as a target, at least one of the base address registers must be configured for memory space and be 64-bit enabled. This is discussed in the "Customizing the LogiCORE PCI Interface" chapter in this guide. Memory spaces which are 64-bit enabled support both 32-bit and 64-bit transfers.

Target Extension Signals

In addition to the target signals presented in the "Target Data Transfer and Control" chapter in this guide, there are a few additional signals present in 64-bit implementations of the LogiCORE PCI interface.

- **ADIO[63:32]** -- this bidirectional bus provides the means for 64-bit data and address transfer to and from the PCI interface.

- **S_CBE[7:4]** -- this input is a registered version of the CBE_IO lines, and is delayed by one cycle. It indicates the PCI command and byte enables during a target transaction. This signal is used primarily for byte enable information, as the command is presented on the lower half of the **S_CBE** bus.

- **S_CYCLE64** -- this input indicates to the user application that the PCI interface has claimed a 64-bit target transaction. It is asserted

at the same time as **BASE_HIT** and remains asserted until the transaction is complete.

- **SLOT64** -- this signal enables the 64-bit extension signals (AD_IO[63:32], CBE_IO[7:4], and PAR64_IO). This signal must remain static at all times except at power-on and immediately after a PCI Bus reset. The method for determining the appropriate value for this signal is design and device family dependent. Refer to the *LogiCORE PCI Implementation Guide* for more details about **SLOT64**.

Handling 64-bit Transfers

The concepts presented in earlier chapters apply to 64-bit transfers. The methods for transaction decoding and data phase control are the same.

For burst transfers, **S_CYCLE64** should be used to determine the correct "increment value" for target address pointers. Note that 64-bit enabled memory spaces will accept the following types of transactions:

- 64-bit transfers aligned on a QWORD boundary

- 32-bit transfers aligned on a DWORD boundary

The *PCI Local Bus Specification* forbids initiators from requesting unaligned 64-bit transfers. The user application should respond to such transfer attempts with target abort. The following logic will produce this result.

```
assign S_ABORT = S_CYCLE64 & ADDR[2];
```

Figure 9-1   Unaligned 64-bit Target Read Transaction

Using this logic, Figure 9-1, "Unaligned 64-bit Target Read Transaction" shows the LogiCORE PCI interface responding to an unaligned 64-bit read attempt with a target abort.
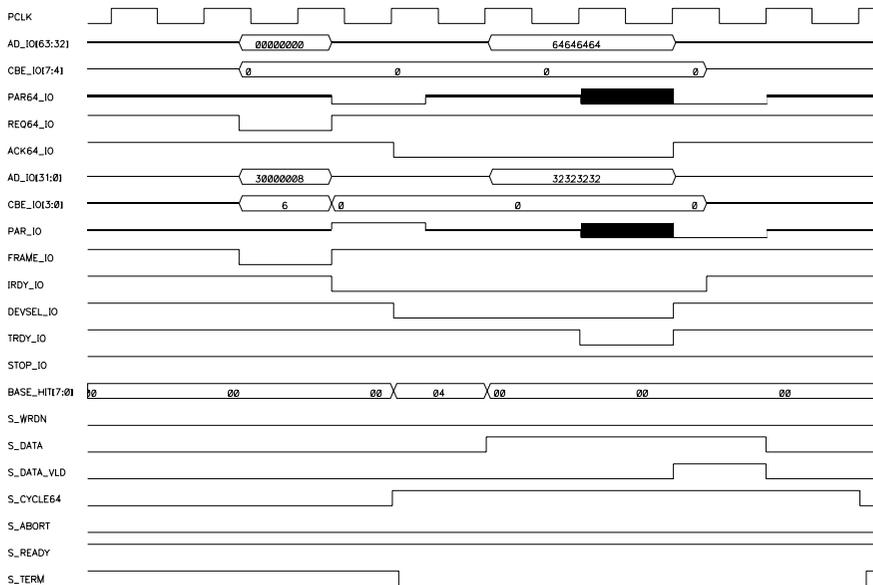
PCLK

AD_IO[63:32]   00000000   64646464

CBE_IO[7:4]   Ø   Ø   Ø   Ø

PAR64_IO

REQ64_IO

ACK64_IO

AD_IO[31:0]   30000008   32323232

CBE_IO[3:0]   6   Ø   Ø   Ø

PAR_IO

FRAME_IO

IRDY_IO

DEVSEL_IO

TRDY_IO

STOP_IO

BASE_HIT[7:0]   00   00   00 04   00   00   00

S_WRDN

S_DATA

S_DATA_VLD

S_CYCLE64

S_ABORT

S_READY

S_TERM

Figure 9-2   Aligned 64-bit Target Read Transaction

Figure 9-2, "Aligned 64-bit Target Read Transaction" demonstrates an aligned 64-bit read attempt. Note that the behavior of the 64-bit extension signals mirrors that of the standard 32-bit signals. In this transfer attempt, as in Figure 9-1, "Unaligned 64-bit Target Read Transaction", the `S_CYCLE64` signal is asserted.

For adequate support of 32-bit transfers, the user application must monitor `S_CYCLE64` to behave appropriately. Typically, this will involve changes to the target address pointer, changes to any "back up" counters, and the addition of multiplexers on the data path.

To avoid the added complexity involved with elaborate address pointers and back up counters, an alternate solution is to only support burst transfers when the transaction request is 64-bit (that is, when `S_CYCLE64` is asserted). When 32-bit transfers are requested, the user application can instruct the PCI interface to disconnect with data after a single data phase. Note that this does not eliminate the need for multiplexers on the data path.
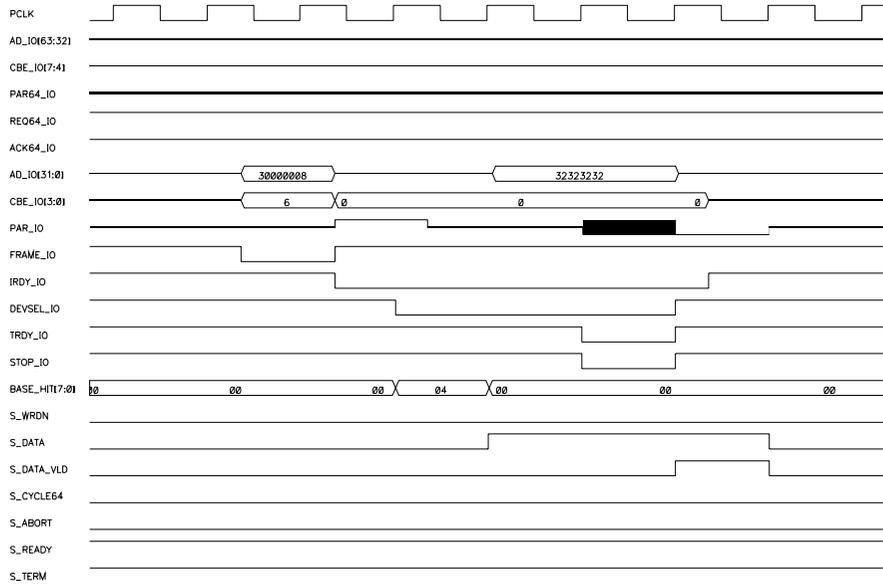
PCLK	
AD_IO[63:32]	
CBE_IO[7:4]	
PAR64_IO	
REQ64_IO	
ACK64_IO	
AD_IO[31:0]	30000008 32323232
CBE_IO[3:0]	6 0 0 0
PAR_IO	
FRAME_IO	
IRDY_IO	
DEVSEL_IO	
TRDY_IO	
STOP_IO	
BASE_HIT[7:0]	00 00 00 04 00 00 00
S_WRDN	
S_DATA	
S_DATA_VLD	
S_CYCLE64	
S_ABORT	
S_READY	
S_TERM	

Figure 9-3   Aligned 32-bit Target Read Transaction

Figure 9-3, "Aligned 32-bit Target Read Transaction" shows a 32-bit transfer attempt to a 64-bit enabled memory space. This may occur if 32-bit initiators reside in a 64-bit system, or if the 64-bit PCI interface is used in a 32-bit slot. In this case, `S_CYCLE64` is not asserted and the PCI interface does not drive the 64-bit extension signals.

Chapter 10

Target Only Designs

This chapter discusses how to correctly disable the initiator functions present in the LogiCORE PCI interface. Previous versions of the Logi-CORE PCI interface were available in "target/initiator" and "target-only" versions. Current versions of the interface are available only in the "target/initiator" configuration.

Disabling the initiator functions is trivial, but it must be done as directed to ensure that the design can be implemented correctly.

Logic Design Considerations

Conceptually, creating a target-only design is as simple as driving all initiator control signals to a benign (deasserted) state. Additionally, the user application should be designed without making use of any initiator status or state outputs from the PCI interface.

However, simply connecting all unused control signals to logic high or logic low may have unwanted side effects. By doing so, the map program may optimize these signals away during the implementation step. In itself, this is not a problem.

Unfortunately, the guide files, which are required to guarantee timing in some designs, will fail to work correctly if certain of these signals have been optimized away.

The solution to this problem is to drive the initiator control signals from the output of flip-flops. An example of this is presented below:

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized
   if (RST) FAKE_LOGIC_0 = 1'b1;
   else FAKE_LOGIC_0 = 1'b0;
end
```

Table 10-1, "Initiator Control Signals", lists the initiator control signals that must be tied off and the appropriate "benign" values. The table also identifies which signals must be driven from a flip-flop as specified above. Some signals are present only in 64-bit implementations of the interface.

Table 10-1   Initiator Control Signals

Signal Name	Benign Value	Flip-Flop
REQUEST	0	No
REQUESTHOLD	0	No
COMPLETE	1	Yes
M_WRDN	0	Yes
M_READY	1	Yes
M_CBE[3:0]	0110	Optional
CFG_SELF	0	No
REQUEST64	0	No
M_CBE[7:4]	0000	Optional

System Level Considerations

One additional system level issue should be considered during the design phase. A target only design has no need to drive REQ_O or to monitor GNT_I on the PCI Bus.

The REQ_O and GNT_I pins may be connected to the PCI Bus (as they would be for initiator designs) or left unconnected at the discretion of the designer. If GNT_I is not connected to the PCI Bus, it must be passively pulled up by an external resistor.

Chapter 11

Initiator Design Tips

This chapter describes some of the design issues involved in building the initiator portion of an application. A good way to begin building the initiator portion of a user application is to write a "mission" statement for what it must accomplish.

Determine the Required Transfers

Consider what data must be moved around by the initiator:

- How big is each transfer and what is its alignment? These affect the transfer counter size and the address counter size.

- How fast can the intended target accept or send data?

- How fast can the user application provide or receive data?

Determine Termination Behavior

Invariably, a target will signal some form of termination condition. How will the user application respond? What should it do?

- An initiator is obliged to retry an operation over again if the target signals a target retry condition. Restart the operation from the beginning.

- An initiator should not retry an operation if it detects a target abort or a master abort condition. This indicates that the operation is either illegal for the selected target or that no target exists.

- Handling a disconnect condition is at the discretion of the user application. Initiators are not required to retry an operation terminated with a disconnect. An initiator can also be terminated if the internal latency timer expires while GNT_I is deasserted.

Determine Transaction Ordering Rules

When the user application requests the bus for an initiator operation, it may not be granted the bus for quite some time. In the meantime, another agent may initiate a target access to the user application. How should the user application respond?

Does the user application accept the target access? It may contain important information relevant to the pending initiator transaction.

Denying the other agent access by forcing a target retry will be disastrous in a system with priority-based arbitration. The initiating agent may keep retrying the transaction because it has higher priority, and the user application will never access the bus. This results in deadlock. However, in a round-robin system, forcing a target retry is a good way for the user application to perform its pending transaction first. It can then respond to the target access when it is later retried by the other agent.

Assemble the Design

The fundamentals of an initiator design, including a simple state machine, are covered in the "Initiator Data Transfer and Control" chapter of this guide. The material covered in that chapter is sufficient for implementing non-burst transfers to other PCI agents.

For more elaborate designs, the "Initiator Data Phase Control" chapter of this guide presents information on inserting initiator wait states and controlling the length of initiator transfers. This information is useful for both non-burst and burst designs.

For burst designs, the "Initiator Burst Transfers" chapter of this guide demonstrates how to support initiator burst transfers through modifications to the simple state machine and the inclusion of additional logic. The "Initiator 64-bit Extension" chapter of this guide discusses the use of the 64-bit extension for increased performance.

Chapter 12

Initiator Data Transfer and Control

Transferring data as an initiator is a multi-step process. Some of the timing depends on other PCI agents, such as the arbiter and the selected target. The basics of controlling initiator transactions are presented in this chapter. Techniques for more elaborate transactions are covered in later chapters.

Typical Initiator Data Interface

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

A typical initiator data register is interfaced as shown in Figure 12-1, "Example Initiator Register".
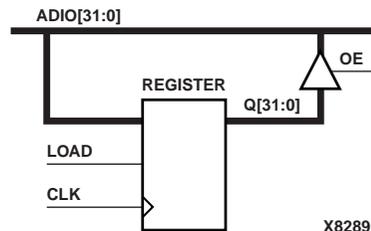

Figure 12-1   Example Initiator Register

The purpose of this chapter is to demonstrate the logic required in the user application to complete simple initiator transfers and thereby generate the load and output enable signals for the data register.

Initiator Interface Signals

The following signals control initiator data transfer to and from the PCI interface. For basic transfers, only a subset of these signals need be used. More elaborate designs involving initiator wait state insertion or initiator burst are covered in later chapters. Note that references to inputs and outputs are made with respect to the user application.

- **ADIO[31:0]** -- this bidirectional bus provides the means for data and address transfer to and from the PCI interface.

- **M_DATA_VLD** -- this input has two interpretations depending on the direction of data transfer. When the user application is sinking data (initiator reads), **M_DATA_VLD** indicates that the user application should capture valid data from the **ADIO** bus. When the user application is sourcing data (initiator writes), **M_DATA_VLD** indicates that a data phase has completed on the PCI Bus.

- **M_SRC_EN** -- this input is only used during initiator burst writes. It indicates to the user application that the data source which drives output data onto the **ADIO** bus must provide the next piece of data. In most applications, this signals the user application to advance the data pointer for the source that is providing data.

- **TIME_OUT** -- this input indicates to the user application that the latency timer has expired. This means that the user application has exceeded the maximum number of clock cycles allowed by the system configuration software. It is only of importance to designs that perform initiator bursts.

- **CSR[39:0]** -- this input provides status information about the current transfer. This is used primarily in initiator burst applications with non-prefetchable sources to determine if any associated address pointers must be "backed up".

- **REQUEST** -- this output from the user application instructs the PCI interface to request the PCI Bus and to begin an initiator transaction once GNT_I is asserted.

- **REQUESTHOLD** -- this output from the user application indicates that it desires to continue requesting the PCI Bus for an extended period of time.

- **M_WRDN** -- this output indicates the direction of data transfer for the current initiator transaction. Logic high indicates that the user application is sourcing data (i.e. initiator write). The user application should not change this output during a transaction.

- **M_CBE[3:0]** -- this output indicates the PCI command and byte enables during an initiator transaction. The command should be presented during the assertion of **M_ADDR_N** by the PCI interface, and the byte enables should be presented during each data phase.

- **M_READY** -- this output from the user application indicates that it is ready to transfer data, and can be used to insert wait states during the first data phase of a transaction.

- **COMPLETE** -- this output from the user application informs the initiator state machine that it should complete the current transaction.

The following signals are output by the initiator state machine in the PCI interface. These states are defined in Appendix B of the *PCI Local Bus Specification*.

- **I_IDLE** -- this input indicates that the initiator state machine is in the idle state and that it is not actively driving the PCI Bus.

- **DR_BUS** -- this input indicates that the initiator state machine is driving the PCI Bus because the arbiter has parked the bus on the PCI interface (GNT_I asserted with no pending or active request for bus grant from the user application).

- **M_ADDR_N** -- this input indicates that the initiator state machine expects the user application to drive **ADIO** with the PCI Bus address for a requested transaction.

- **M_DATA** -- this input indicates that the initiator state machine is in the data transfer state.

Initiator Control

Performing a complete initiator transaction consists of several steps. The proper sequencing of the PCI interface control signals is best performed by a state machine in the user application. In addition to a state machine and a data register, additional logic is required to drive outputs to the PCI interface and to generate inputs used by the state

machine. This is represented in the block diagram of Figure 12-2,
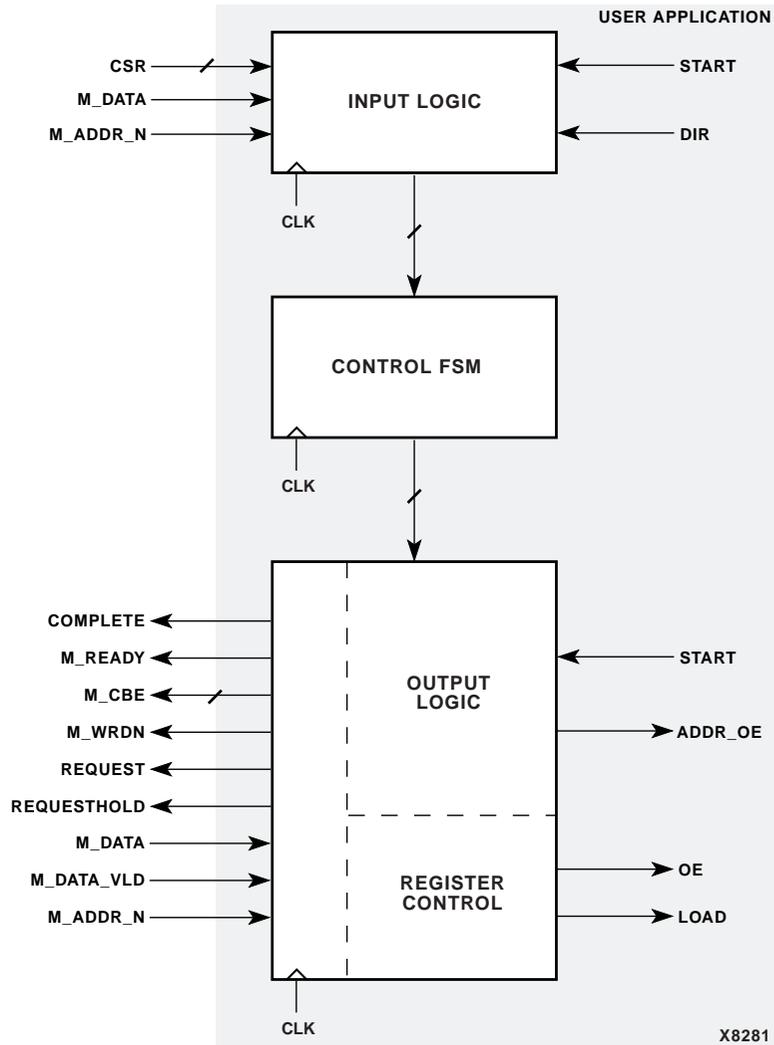"Initiator Control Block Diagram".

Figure 12-2   Initiator Control Block Diagram

The following discussion presents a sample design that is suitable for
performing single data phase initiator transfers in the form of

memory reads and memory writes. The extension of this example to add support for burst transfers is discussed in later chapters. The entire design breaks down into several distinct sections:

- Inputs to the state machine
- The state machine itself
- Outputs to the PCI interface
- Data register control signals

Inputs to the State Machine

The sample state machine requires one input to tell it to start, another input to indicate the direction of the desired transfer, and five additional inputs to effectively monitor the progression of the transaction. These signals are discussed below:

The START signal indicates to the state machine that it should begin a transaction. This signal is generated by the user application. The source of this signal will vary depending on the specific user application function.

The DIR signal indicates the direction of the desired transfer, and is generated by the user application. The source of this signal will vary depending on the specific user application function. The DIR signal should not change from the time START is asserted until the end of the initiator transfer. In this particular example, initiator writes are selected by driving DIR to logic high.

The signals **M_ADDR_N** and **M_DATA** are outputs of the PCI user interface that are used by the state machine. The signal M_DATA_FELL indicates that a falling edge has been detected on **M_DATA**. This is generated by the following code.

```
always @(posedge CLK or posedge RST)
begin : edge_detect
    if (RST) M_DATAQ = 1'b0;
    else M_DATAQ = M_DATA;
end


assign M_DATA_FELL = !M_DATA & M_DATAQ;
```

The signals FATAL and RETRY are derived from PCI interface signals. These signals indicate how a transaction attempt has terminated. The FATAL signal indicates that a master abort or target abort has occurred. The RETRY signal indicates that the target issued a discon-nect without data. As the initiator state machine only performs single transfers, this implies a retry by the target.

This information is obtained from the extended status signals, **CSR[39:32]**. The extended status information is valid one clock cycle after a particular event has occurred on the PCI Bus. Unless the status is used during that cycle, it must be registered to preserve it for later use. The following code performs the task of preserving the status information from the final data phase.

```
always @(posedge CLK or posedge RST)
begin : watch_status
   if (RST)
   begin
      FATAL = 1'b0;
      RETRY = 1'b0;
   end
   else if (!M_ADDR_N)     // clear at beginning
   begin
      FATAL = 1'b0;
      RETRY = 1'b0;
   end
   else if (M_DATA)        // latch until end
   begin
      FATAL = CSR[39] | CSR[38];
      RETRY = CSR[36];
   end
end
```

The State Machine

The sample state machine is shown in Figure 12-3, "Sample User Application State Machine". As stated previously, this state machine is suitable for performing single data phase initiator transfers.
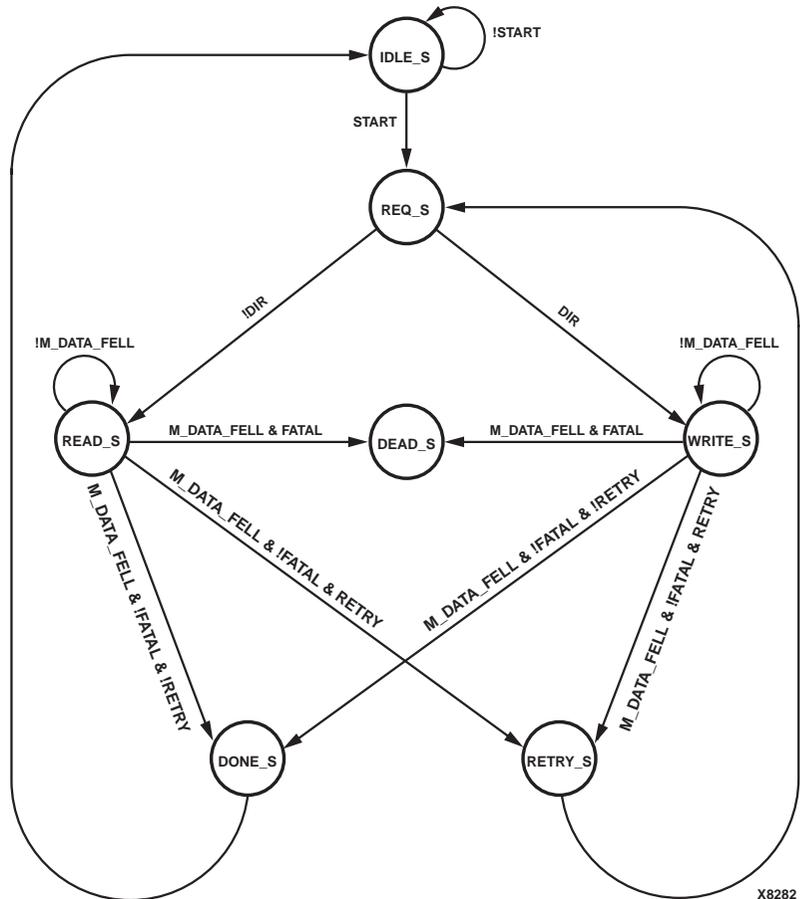
Figure 12-3   Sample User Application State Machine

Each state performs a specific step in the sequence. The individual states and their purposes are listed below.

IDLE_S is the idle state. During this state, no initiator activity takes place, and the state machine waits for the user application to assert START. Once START has been asserted, the state machine proceeds to the REQ_S state.

REQ_S is the request state. During this state, a request is made to begin an initiator transaction. The state machine then branches based

on the value of the DIR signal. If DIR indicates an initiator read, the next state is READ_S. Otherwise, the next state is WRITE_S.

READ_S and WRITE_S are data transfer states. The state machine stays in READ_S or WRITE_S until it detects that the transaction is over. The transaction is over when M_DATA_FELL is asserted.

These states are identical except that the state machine outputs differ. The output logic is discussed in the next section. Since DIR is available in the user application, these two states may be reduced into a single XFER_S state if DIR is used in the data register control equations. The states are separate in this example for clarity. In more elaborate designs, it is often desirable to completely split the state machine into separate read and write state machines. For very simple designs, it is possible to further reduce the example state machine presented here.

Whether in READ_S or WRITE_S, the state machine makes a three way branch after the transfer state, based on the FATAL, and RETRY signals. If a fatal error has occurred, the state machine moves to the DEAD_S state. If no fatal error occurred and no data transfer occurred, the state machine moves to the RETRY_S state. Otherwise, the state machine moves to the DONE_S state.

DEAD_S is the terminal state for handling fatal errors. In practice, the user application should provide some method for resetting the state machine to handle such errors.

RETRY_S is the retry state. In this simple example, no actions are performed in the RETRY_S state, and the state machine immediately transitions to the REQ_S state and attempts the transaction again. More elaborate designs may implement a retry counter to discard transactions after a certain number of retries, as permitted in the *PCI Local Bus Specification*.

DONE_S is the done state. As in the RETRY_S state, no specific actions are performed in this state, and the state machine immediately transitions back to the IDLE_S state.

The state machine shown in Figure 12-3, "Sample User Application State Machine" could be implemented as shown:

```
always @(posedge CLK or posedge RST)
begin : initiator_fsm
    if (RST) STATE = IDLE_S;
```

```
else case (STATE)
    IDLE_S :  begin
                if (START) STATE = REQ_S;
                else STATE = IDLE_S;
                end
    REQ_S :   begin
                if (DIR) STATE = WRITE_S;
                else STATE = READ_S;
                end
    WRITE_S : begin
                if (M_DATA_FELL)
                begin
                    if (FATAL) STATE = DEAD_S;
                    else if (RETRY) STATE = RETRY_S;
                    else STATE = DONE_S;
                end
                else STATE = WRITE_S;
                end
    READ_S :  begin
                if (M_DATA_FELL)
                begin
                    if (FATAL) STATE = DEAD_S;
                    else if (RETRY) STATE = RETRY_S;
                    else STATE = DONE_S;
                end
                else STATE = READ_S;
                end
    RETRY_S : STATE = REQ_S;
    DONE_S :  STATE = IDLE_S;
    DEAD_S :  STATE = DEAD_S;
    default : STATE = IDLE_S;
endcase
end
```

Outputs to the PCI Interface

The state machine in the user application is responsible for driving
several signals which are output to the PCI interface. These signals
are discussed below along with the logic required to generate them.
This discussion assumes the existence of a 32-bit ADDRESS signal
provided by other logic in the user application. This ADDRESS repre-
sents the PCI Bus address of the desired target. The source of this
signal will vary depending on the specific user application function.

Before requesting a transaction, the user application should indicate
that it is ready to transfer data. Although it is possible to initiate a
transaction and then insert wait states using the **M_READY** signal, bus
bandwidth is conserved by requesting a transaction only after the
user application is ready to transfer data. In this example, the user
application is always ready.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
   if (RST) M_READY = 1'b0;
   else M_READY = 1'b1;
end
```

The **M_READY** signal must not be assigned a static value. For timing
reasons, this signal should be driven from the output of a flip-flop,
although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

The state machine initially requests access to the PCI Bus by asserting
REQUEST. This is done in the REQ_S state and results in the assertion
of REQ_O. The **REQUEST** signal must only be asserted for a single
cycle to initialize a request. Note that the bus master enable bit in the
command register must be set before the PCI interface is able to
request the bus. This is the responsibility of the host bridge and
system configuration software.

```
assign REQUEST = (STATE == REQ_S);
```

The **REQUESTHOLD** signal is not used in this example, and is simply
assigned to logic zero.

```
assign REQUESTHOLD = 1'b0;
```

When the PCI interface has received a bus grant, the user application must drive the target address on the **ADIO** bus. This address is presented on the PCI Bus during the address phase.

```
assign ADDR_OE = M_ADDR_N;
assign ADIO = ADDR_OE ? 32'bz : ADDRESS;
```

Simultaneously, the user application must also drive **M_CBE** to indicate the desired PCI Bus command. At other times, the **M_CBE** signal is used to indicate byte enables. In this example, the initiator performs memory read and memory write transactions (commands 0x6 and 0x7, respectively), depending on the value of the DIR signal. All bytes are enabled. The bus command and byte enables are easily modified to accommodate the requirements of the user application.

```
assign COMMAND = {3'b011, DIR};
assign BYTE_ENABLE = 4'b0000;
assign M_CBE = M_ADDR_N ? BYTE_ENABLE : COMMAND;
```

Throughout the entire transaction, the user application must drive **M_WRDN** to indicate the desired transfer direction. This signal must remain constant during a transaction. By assigning it to track the DIR signal, which was stipulated to be constant during a transaction, this requirement is met.

```
assign M_WRDN = DIR;
```

The **M_WRDN** signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

The final signal, **COMPLETE**, indicates to the PCI interface that it should finish the current transaction. In the case of single transfers, **COMPLETE** must be asserted no later than one cycle after **REQUEST**, and deasserted when the transfer is complete.

```
always @(posedge CLK or posedge RST)
begin : driving_complete
    if (RST) COMPLETE = 1'b0;
    else case (STATE)
        REQ_S :   COMPLETE = 1'b1;
        READ_S :  COMPLETE = 1'b1;
```

```
            WRITE_S : COMPLETE = 1'b1;
            default : COMPLETE = 1'b0;
         endcase
      end
```

The **COMPLETE** signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, if possible. In practice, it is usually driven by combinational logic.

Note: Never tie this signal to logic one or to logic zero.

Data Register Control Signals

As a final consideration, the logic for the load and output enables of the typical initiator data register shown in Figure 12-1, "Example Initiator Register" are as follows:

```
assign LOAD = (STATE == READ_S) & M_DATA_VLD;
assign OE = (STATE == WRITE_S) & M_DATA;
```

As is the case in target read transactions, the user application should always drive the entire width of the ADIO bus during initiator writes, even if some byte enable signals are not asserted. For initiator reads, the LOAD signal may be further qualified by the byte enable signals to generate separate load signals for each byte.

Sample Transactions

The following figures illustrate typical PCI Bus read and write transactions that result from implementing the logic presented in this chapter, as well as several exceptional cases that can arise.
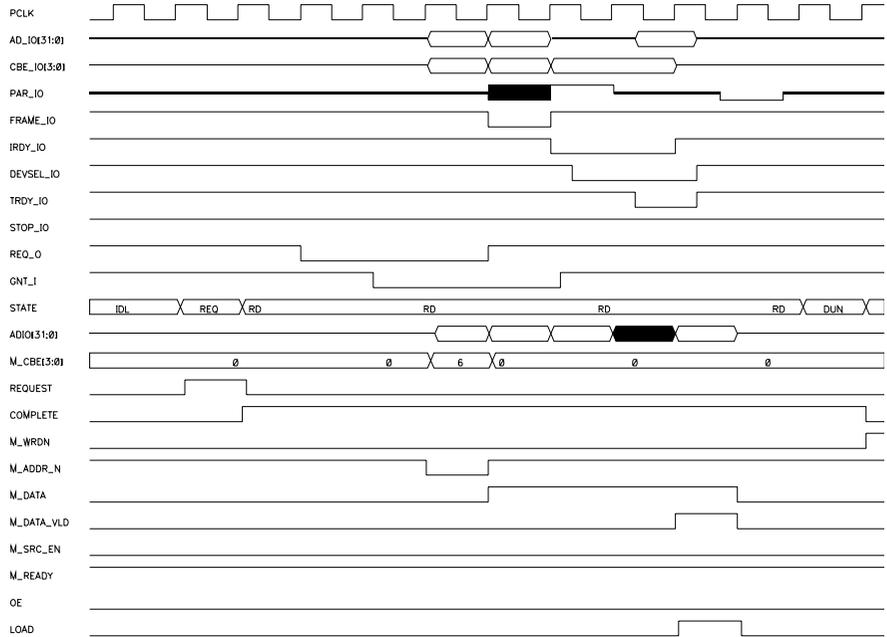
Figure 12-4   Initiator Read Transaction

Figure 12-4, "Initiator Read Transaction" demonstrates the initiator issuing a read command to a target. The transaction is terminated in a normal fashion by the initiator. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during an initiator read transaction.
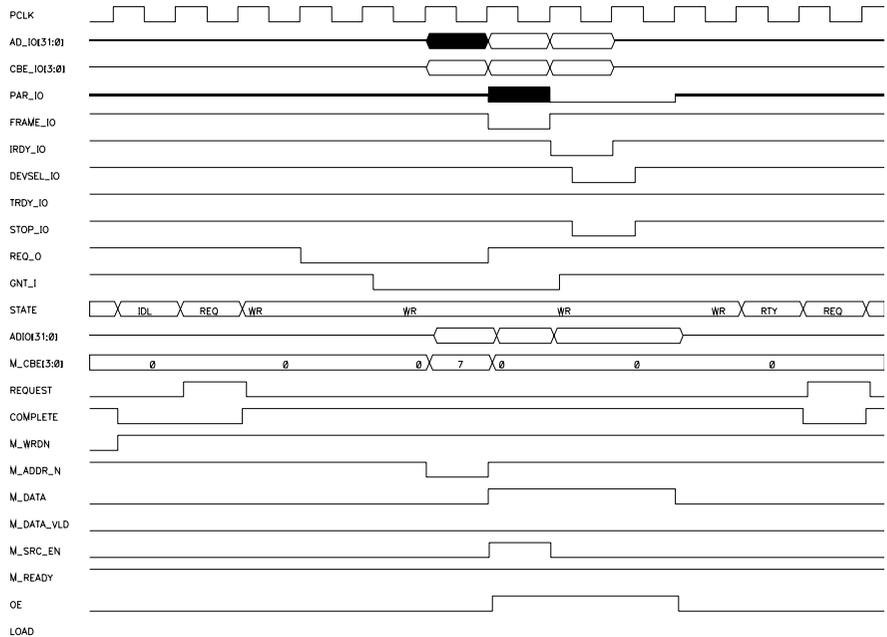
Figure 12-5   Initiator Write Retried by Target

Figure 12-5, "Initiator Write Retried by Target" shows the initiator issuing a write command to a target. The target terminates the transaction with retry. When this occurs, no data is transferred, and the initiator will try again.
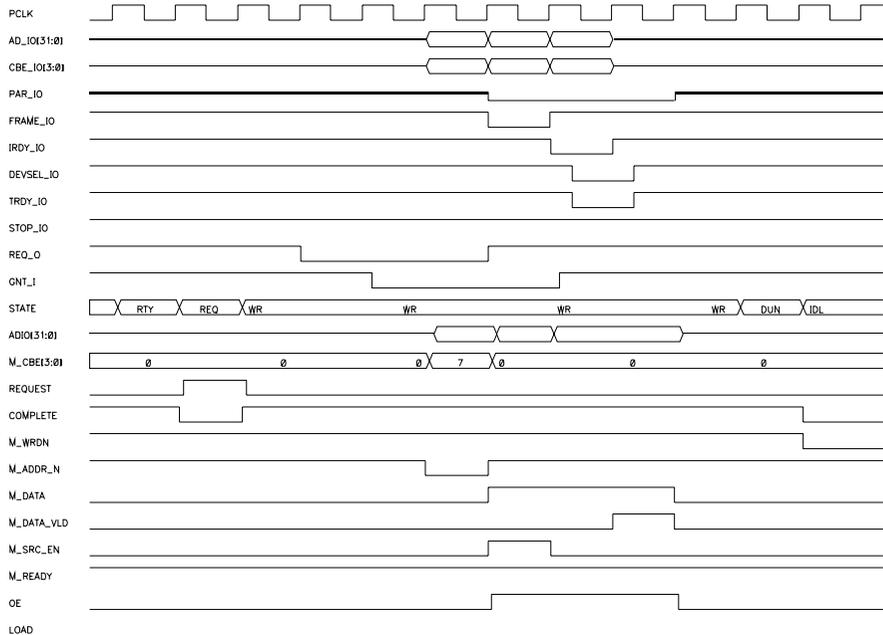
Figure 12-6   Initiator Write Retried and Completed

Figure 12-6, "Initiator Write Retried and Completed", shows the initiator re-issuing the original command. This time the target does not retry the transaction. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during an initiator write transaction.
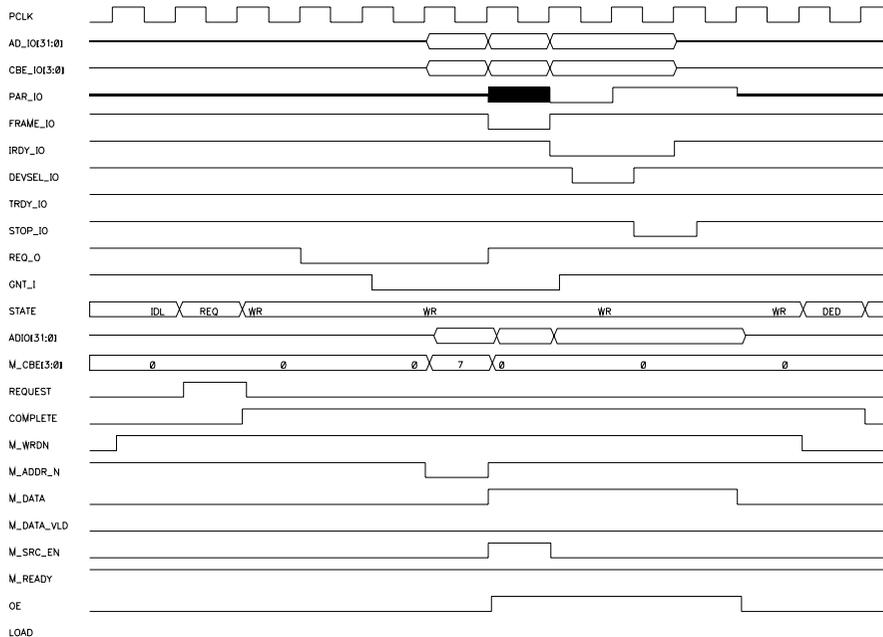
Figure 12-7   Target Abort

Figure 12-7, "Target Abort" demonstrates the behavior of the PCI interface when a target signals an abort (target abort). This event often occurs due to incorrect programming or serious system errors. It can also signify that the initiator has incorrectly attempted to burst data beyond the address space of the target. The user application must respond appropriately.
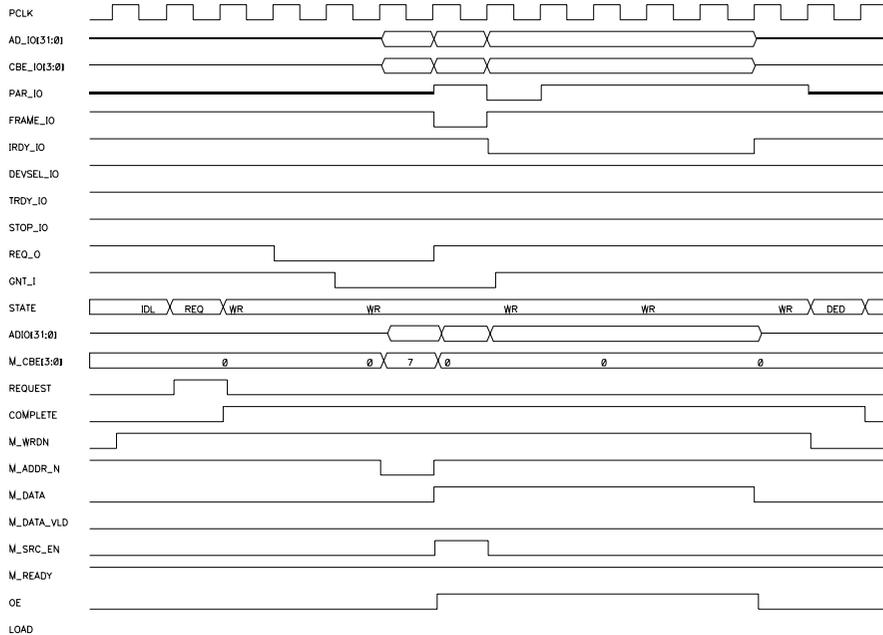
Figure 12-8   Master Abort

Figure 12-8, "Master Abort" demonstrates the behavior of the PCI interface when no target responds (master abort). This event is expected during some configuration transactions and special cycle broadcast cycles. However, during normal operation, this would occur due to incorrect programming or serious system errors. It is critical that the user application respond appropriately.

Chapter 13

Initiator Data Phase Control

This chapter discusses the mechanism by which the user application can control aspects of initiator transactions to accommodate its own ability to source or sink data.

The user application may insert wait states before the first data transfer to allow itself additional time if it is not ready. Additionally, the user application may control the number of data phases by indicating when to complete the transaction.

The generation of initial latency with wait states, while possible, is not recommended. This technique wastes valuable bus bandwidth. From a bandwidth perspective, it is far better for the user application to delay making a bus request until it is ready to perform a transaction.

Control Modes

Data phase control is achieved using the **M_READY** and **COMPLETE** signals. Combinations of the two control signals yield the following four modes:

- Wait Burst -- the wait burst mode inserts wait states at the beginning of a PCI Bus transaction (holds off the first data phase) by delaying the assertion of IRDY_IO by the PCI interface. This particular wait mode is for use with initiator burst transactions. Use of this mode indicates to the PCI interface that the user application is not ready and will attempt more than one data phase.

- Wait Single -- the wait single mode inserts wait states at the beginning of a PCI Bus transaction in the same manner as the wait burst mode. However, this mode is for use with single initiator transactions. Use of this mode indicates to the PCI interface

that the user application is not ready and will not attempt more than one data phase.

- Proceed -- the proceed mode allows PCI Bus data phase(s) to proceed without interruption. While the selected target may insert wait states or terminate the transaction prematurely, the user application must be prepared to transfer data at full speed. This is for use with multiple data phase transfers only.

- Finish -- the finish mode causes the PCI interface to complete the transaction as soon as possible. This is for use with both single and multiple data phase transfers.

Again, the exact disconnect sequence is affected by whether or not the selected target terminates the transaction. The PCI interface will automatically generate the correct behavior.

Table 13-1, "Data Phase Control Signals for Initiators", shows the four modes of operation and the corresponding **M_READY** and **COMPLETE** values.

Table 13-1   Data Phase Control Signals for Initiators

Condition	Bus Signals	From User Application	
		M_READY	**COMPLETE**
Wait Burst	IRDY_IO = 1 FRAME_IO = 0	Low	Low
Wait Single	IRDY_IO = 1 FRAME_IO = 1	Low	High
Proceed	IRDY_IO = 0 FRAME_IO = 0	High	Low
Finish	IRDY_IO = 0 FRAME_IO = 1	High	High

Changing from one mode to another must not be done in arbitrary sequence. In addition, the timing of mode transitions is critical to ensure precise control over the number of data phases that occur in a transaction. This is of particular importance when switching to the finish mode.

The permitted data phase control sequences for initiator designs using the PCI interface are shown in Figure 13-1, "Permitted Data Phase Control Sequences". The exact timing details are covered in subsequent sections of this chapter.
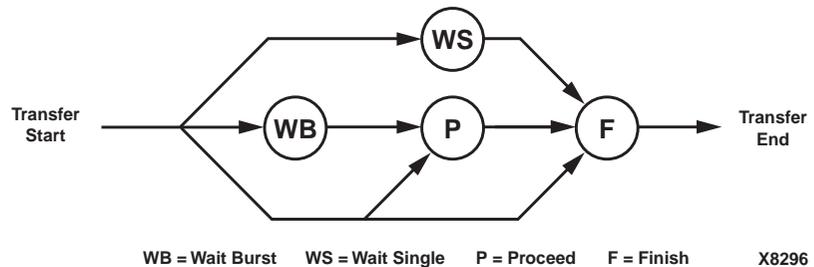

WB = Wait Burst   WS = Wait Single   P = Proceed   F = Finish   X8296

Figure 13-1   Permitted Data Phase Control Sequences

The control sequences shown in Figure 13-1, "Permitted Data Phase Control Sequences" assume that the user application is terminating the transaction. In practice, a transaction may end for reasons that the user application cannot control, such as a target termination or a timeout. When the initiator state machine becomes inactive after such a condition, the sequencing rules no longer apply.

Note that the wait modes cannot be used to insert wait states during arbitrary data phases in a transaction. They may only be used to delay the completion of the first data phase of an initiator transaction. This is called master data latency in the *PCI Local Bus Specification*.

Figure 13-1, "Permitted Data Phase Control Sequences" shows that there are four possible sequences for controlling the number of data phases in a transaction. These are:

•   Single transfers with no master data latency

•   Single transfers with master data latency

•   Burst transfers with no master data latency

•   Burst transfers with master data latency

All initiators are required to complete the first data phase of a transaction within 8 clocks from the assertion of FRAME_IO. The *PCI Local Bus Specification* strongly discourages the use of master data latency and states that there is generally no reason for using it. The user application is responsible for observing this requirement.

Note: During initiator state machine activity, do not violate the mode sequencing shown above. When the initiator state machine is inactive, this requirement does not apply.

Control Pipeline

In order to meet the stringent PCI Bus performance requirements, the PCI interface pipelines all of the bus control signals and the data path. Consequently, the **M_READY** and **COMPLETE** signals must be presented in advance of the desired effect.

The signals **M_READY** and **COMPLETE** connect to the initiator state machine through multiple levels of logic. For this reason, it is highly recommended that the logic driving these signals be kept as simple as possible. Although it is advantageous to drive these signals from flip-flops in the user application, this is typically not possible in all but the most simple (non-burst) designs.

The user application must present the correct initial data phase control mode no later than one cycle after asserting **REQUEST**. After this, the user application may change modes as long as it does not violate the sequencing shown in Figure 13-1, "Permitted Data Phase Control Sequences".

In all cases, once the user application decides to finish an initiator transaction, it must select the finish mode by setting **M_READY** and **COMPLETE** appropriately. Once the user application signals that it wishes to finish a transaction, it must hold **M_READY** and **COMPLETE** until the end of the transaction. The deassertion of **M_DATA** by the PCI interface indicates that the transfer is over.

Transaction Termination Rules

If the user application is sending or receiving a single data word, the user application must signal the wait single or finish mode no later than one cycle after asserting **REQUEST**. If the user application inserts wait states using the wait single mode, it must switch to the finish mode before it causes the PCI interface to violate the master data latency specification. Again, once the user application signals to finish the transaction, it must hold **M_READY** and **COMPLETE** through the end of the **M_DATA** state.

If the user application is sending or receiving two data words, the user application may start in either the wait burst or proceed modes.

If starting in the wait burst mode, the user application must switch to the proceed mode before it causes the PCI interface to violate the master data latency specification. Once in the proceed mode, the user application must switch to the finish mode when both of the following conditions have been met:

- The "proceed" mode has been signalled for at least one cycle.

- The signal **M_DATA** has been asserted for at least one cycle.

For burst transfers of three or more data words, the initial mode selection is the same as in the two transfer case. The user application should switch to the finish mode when three transfers remain and **M_DATA_VLD** is asserted.

Implementation

The above rule set appears quite complicated when presented textually. The following example demonstrates the logic required to drive **M_READY** and **COMPLETE** in a general case. This example builds on the example presented in an earlier chapter, enabling it to perform simple burst transfers. This code would replace the previous logic that generated **M_READY** and **COMPLETE**.

Most initiator designs will use a transfer counter to track the desired burst length. The following logic implements a transfer counter and generates three outputs which indicate how many transfers remain. The BURST_LENGTH and START signals are generated elsewhere in the user application. Note that for reads and writes, **M_DATA_VLD** is used to indicate a successful data transfer on the PCI Bus.

```
always @(posedge CLK or posedge RST)
begin : transfer_counter
    if (RST) XFER_CNT = 4'h0;
    else if (START) XFER_CNT = BURST_LENGTH;
    else if (M_DATA_VLD) XFER_CNT = XFER_CNT - 4'h1;
end

assign CNT3 = (XFER_CNT == 4'h3);
assign CNT2 = (XFER_CNT == 4'h2);
assign CNT1 = (XFER_CNT == 4'h1);
```

The following logic generates an initiator "ready" signal that is used later in the **M_READY** and **COMPLETE** logic. In user application designs that do not insert wait states as an initiator, this logic can be optimized away. The logic samples a signal called READY_FLAG, which is produced elsewhere in the user application. This signal indicates that the user application is ready to transfer data.

```
always @(posedge CLK or posedge RST)
begin : not_recommended
    if (RST)
    begin
        INIT_READY = 1'b1;
        INIT_WAITED = 1'b0;
    end
    else
    begin
        INIT_WAITED = !INIT_READY;
        if (START) INIT_READY = READY_FLAG;
        else if (!INIT_READY) INIT_READY = READY_FLAG;
    end
end
```

The final step is to generate **M_READY** and **COMPLETE**. Typically, initiator wait states are not used and the INIT_WAITED signal should be optimized out of the equations. It is critical to reduce the logic as much as possible for timing reasons.

```
assign FIN1 = CNT1 & REQUEST;
assign FIN2 = CNT2 & M_DATAQ & !INIT_WAITED;
assign FIN3 = CNT3 & M_DATA_VLD;

assign ASSERT_COMPLETE = FIN1 | FIN2 | FIN3;
assign COMPLETE = ASSERT_COMPLETE | HOLD_COMPLETE;
assign M_READY = INIT_READY;

always @(posedge CLK or posedge RST)
begin : finish_up
    if (RST) HOLD_COMPLETE = 1'b0;
```

```
                else if (M_DATA_FELL) HOLD_COMPLETE = 1'b0;

                else if (ASSERT_COMPLETE) HOLD_COMPLETE = 1'b1;

        end
```

Note: Do not forget that **M_READY** and **COMPLETE** must not be assigned static values.

Sample Transactions

The following figures illustrate typical initiator read and write burst transactions that result from implementing the logic presented in this chapter. Sample transactions for cases of single transfers were presented in earlier chapters.
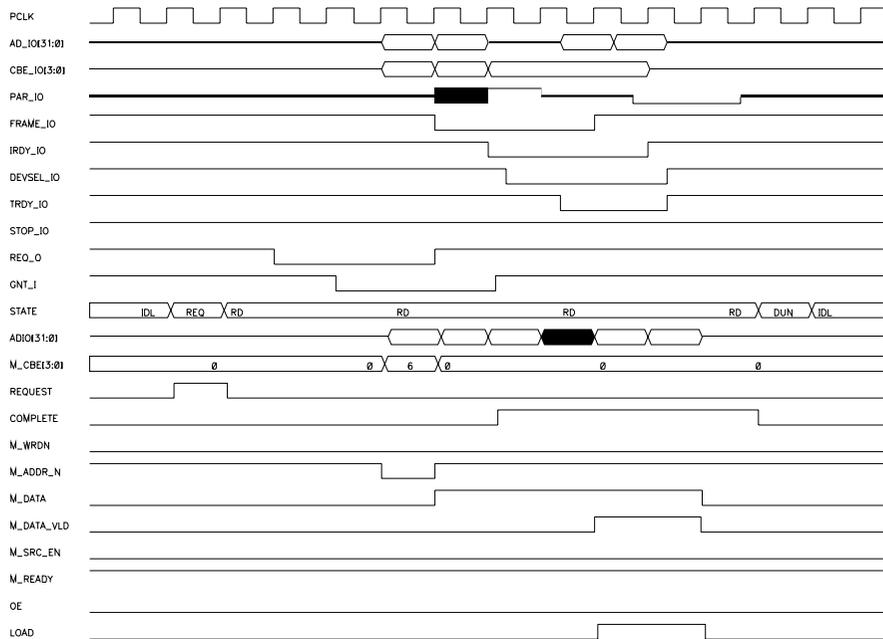

Figure 13-2   Two DWORD Initiator Read Transfer

Figure 13-2, "Two DWORD Initiator Read Transfer", demonstrates the PCI interface performing a burst transfer. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during the transaction.

PCLK
AD_IO[31:0]
CBE_IO[3:0]
PAR_IO
FRAME_IO
IRDY_IO
DEVSEL_IO
TRDY_IO
STOP_IO
REQ_O
GNT_I
STATE   IDL   REQ   WR   WR   WR   WR   DUN   IDL
ADIO[31:0]
M_CBE[3:0]   0   0   7   0   0   0
REQUEST
COMPLETE
M_WRDN
M_ADDR_N
M_DATA
M_DATA_VLD
M_SRC_EN
M_READY
OE
LOAD
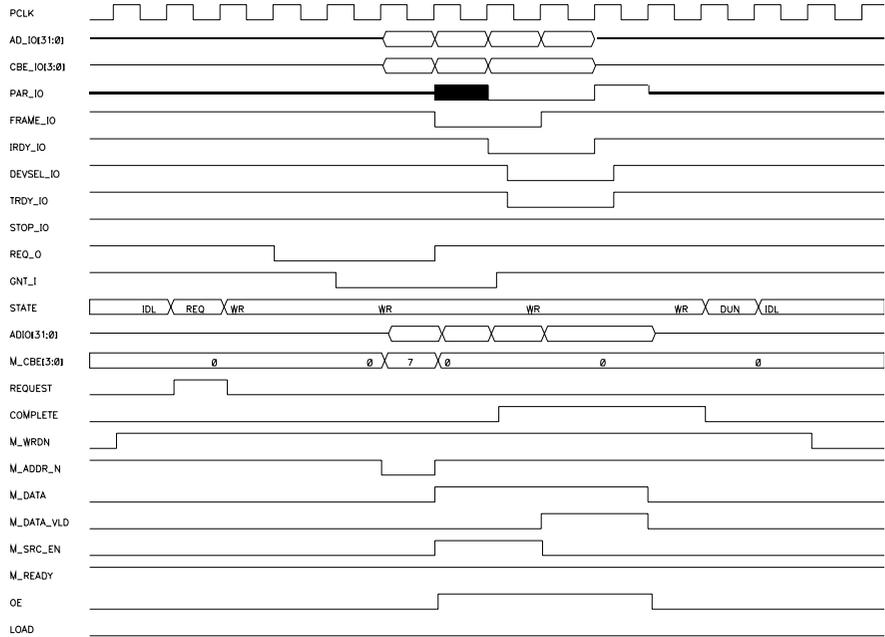
Figure 13-3   Two DWORD Initiator Write Transfer

Figure 13-3, "Two DWORD Initiator Write Transfer", demonstrates the PCI interface performing a burst transfer. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during the transaction.
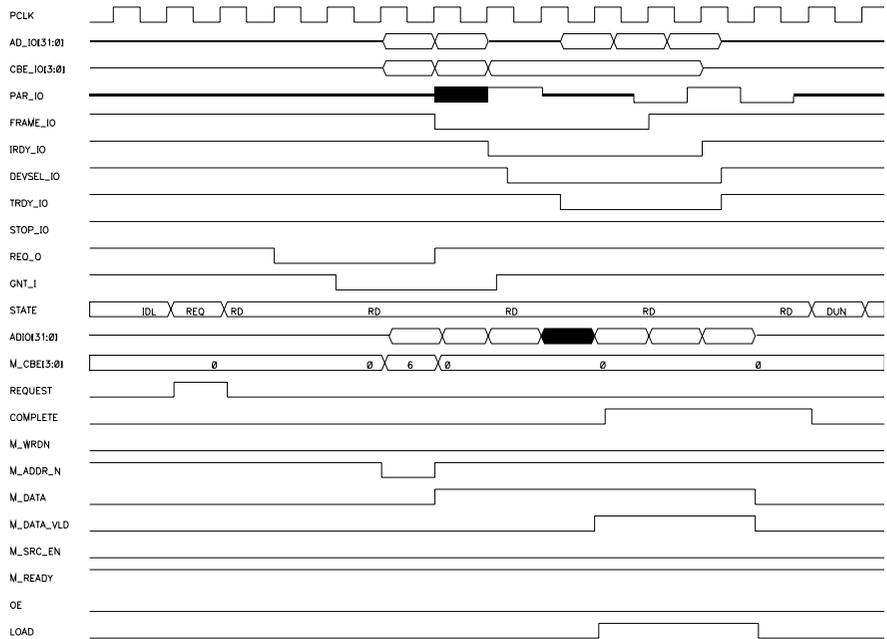
Figure 13-4   Three DWORD Initiator Read Transfer

Figure 13‑4, "Three DWORD Initiator Read Transfer", demonstrates the PCI interface performing a burst transfer.

Figure 13-5   Three DWORD Initiator Write Transfer

Figure 13-5, "Three DWORD Initiator Write Transfer", demonstrates the PCI interface performing a burst transfer. This figure also shows the timing relationship between selected user application signals and PCI Bus signals during the transaction.

Chapter 14
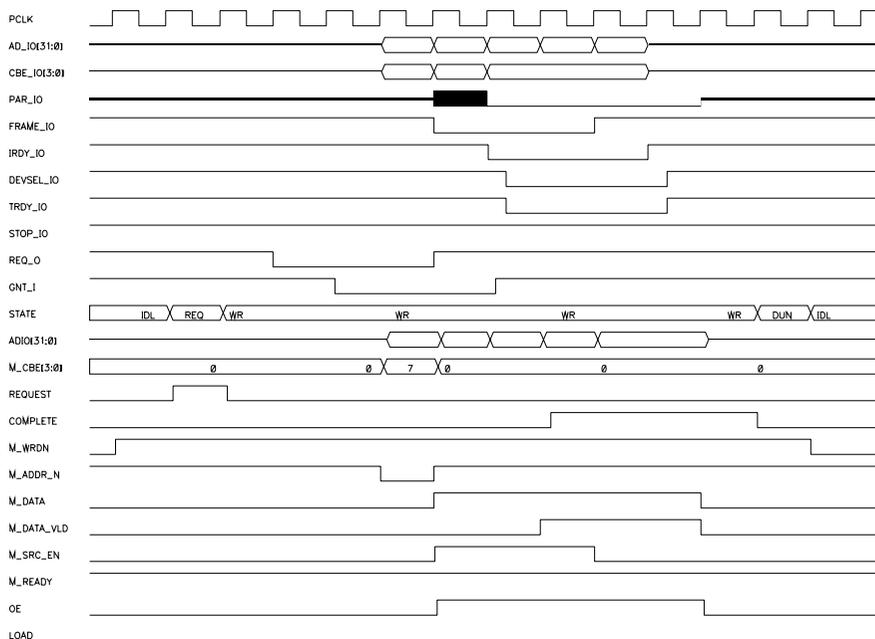
Initiator Burst Transfers

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

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

Keeping Track of the Address Pointer

In a PCI transaction, only the starting address is broadcast over the bus. For single transfers as an initiator, a register that holds the target address is sufficient.

For burst transfers, however, the user application must keep track of the current address because the target can terminate the transaction at any time. If the initiator is to continue the transfer during another transaction, it must resume at the appropriate address.

The initiator must always provide and track a full 32 bit address (the lowest two bits are always zero). In some applications, however, a smaller address counter and a larger data register are sufficient to track the current address. The actual size of the counter depends on the alignment and length of the transfers required by the user application. In the worst case, the initiator may require a full 30-bit, loadable binary counter. See Table 14-1, "Example Initiator Address Pointer".

Table 14-1   Example Initiator Address Pointer

31		2	1	0
30-bit loadable binary counter			0	0

Two methods may be used for incrementing the initiator address pointer. The first method is to use **M_DATA_VLD** as a increment enable signal. The **M_DATA_VLD** signal indicates successful data transfer for both initiator reads and initiator writes.

This method is simple and ensures that the initiator address pointer is always valid. It is most useful in user application designs that do not require explicit addresses during a transfer, e.g. bursting data to or from FIFOs. An example of this is presented later in this chapter.

Another method is to use **M_DATA_VLD** as an increment enable during initiator reads, and use **M_SRC_EN** during initiator writes. This method allows the initiator address pointer to serve as a local pointer used to index storage elements in the user application.

This is particularly useful in designs with addressable RAM or in other cases where explicit addresses are required. However, this method requires that the initiator address pointer be "backed up" in cases where the target terminates the transaction prematurely or the transfer spans multiple bus transactions. The relationship between **M_DATA_VLD** and **M_SRC_EN** is identical to that of **S_DATA_VLD** and **S_SRC_EN**.

Sinking Data in Burst Transfers

During initiator reads, the PCI interface transfers burst data using a pipelined data path. The data valid signal, **M_DATA_VLD**, is used to advance the initiator address pointer (and any other data pointers in the user application logic). At the same time the initiator address pointer is advanced, the user application also captures valid data from the internal **ADIO** bus.

Using **M_DATA_VLD** to capture burst data is very similar to the simple case of single transfers. The user application may enable different data sinks, if necessary, based upon the current initiator address

pointer. The generation of a local address pointer was discussed in the previous section.

An initiator burst read is shown in the waveform of Figure 14-1, "Initiator Burst Read Transaction". This waveform includes both PCI Bus signals and internal user application signals. In this figure, the initiator address pointer is implemented to provide a local pointer to index storage elements. During initiator reads, this pointer is incremented when **M_DATA_VLD** is asserted.
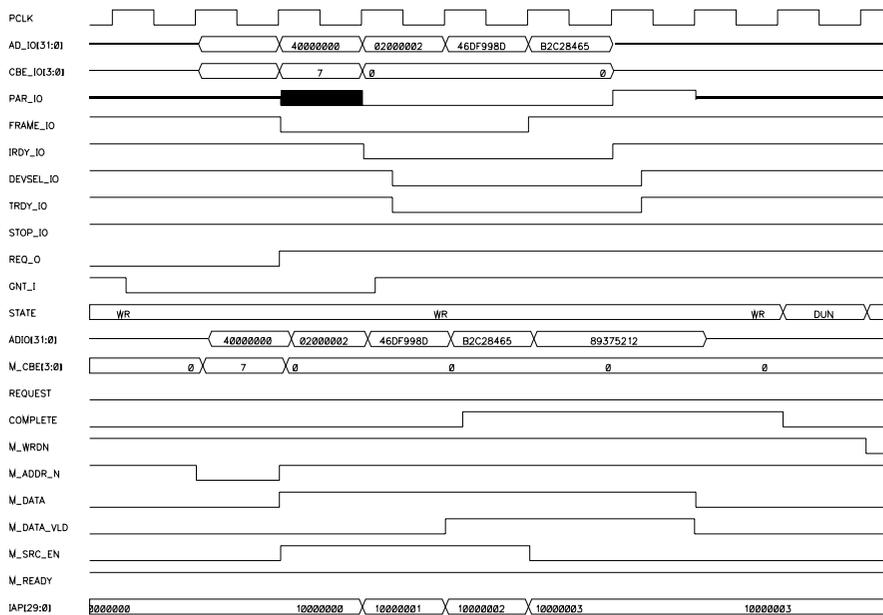

Figure 14-1   Initiator Burst Read Transaction

Sourcing Data in Burst Transfers

During initiator writes, the PCI interface transfers burst data using a pipelined data path. The data source enable signal, **M_SRC_EN**, is used to advance data pointers in the user application logic (and possibly the initiator address pointer). The result is that the user application drives new data onto the internal **ADIO** bus.

Internally, the PCI interface captures the data value provided by the user application on the **ADIO** bus and holds this value in the output

flip-flops driving the AD_IO pins on the PCI Bus. The user application then presents the next data word on **ADIO**, instead of holding the previous data word until the current data phase completes.

An initiator burst write is shown in the waveform of Figure 14-2, "Initiator Burst Write Transaction". This waveform includes both PCI Bus signals and internal user application signals. In this figure, the initiator address pointer is implemented to provide a local pointer to index storage elements. During initiator writes, this pointer is incremented when **M_SRC_EN** is asserted.

Figure 14-2   Initiator Burst Write Transaction

Using **M_SRC_EN** to present data for the next data phase may require additional control logic depending on the type of data source present in the user application. Keep in mind that the **M_SRC_EN** signal advances data pointers (and possibly the initiator address pointer) in anticipation of the next data phase, which may or may not complete with successful data transfer.

If a pointer is advanced, and the data is never transferred, then the user application must decide what to do with the non-transferred

data. In the case of prefetchable data sources, such as RAM or a register file, the data can be discarded. The original data remains in the RAM or the register file for future use.

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

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

Conditions requiring "back up" may arise at the end of an initiator write transfer where the target signals some form of disconnect, terminating the transaction before the initiator is able to complete the full transfer. In these cases, the user application is not immediately aware of the termination condition, and will have advanced the data source too many times. This condition can also arise during data transfers that span multiple bus transactions.

One way to determine the number of times the data pointer (and possibly the initiator address pointer) has been over-advanced during a burst write is to monitor the difference in the number of cycles **M_SRC_EN** and **M_DATA_VLD** have been asserted during a transaction. During initiator writes, the signal **M_DATA_VLD** represents the number of data phases that actually complete with data transfer.

Design Example

The following design example demonstrates burst transfers as an initiator using a non-prefetchable data source. In this particular example, explicit local addresses are not required, so the simple initiator address pointer scheme is used.

Non-prefetchable data sources, such as FIFOs, exhibit "side effects" from reads (that is, the state is altered or lost). Special care must be taken during initiator burst writes so that state information is not lost. The use of **M_SRC_EN** results in reading the data source ahead of the actual transfer. Unless precautions are taken, the data will be lost.

Figure 14-3, "Non-Prefetchable Data Source" shows a FIFO suitable for initiator burst transactions in a user application. To present a concise example, this example uses a single FIFO with both ports accessible through the PCI interface. In practice, the best structure for most applications is a dual-FIFO design with separate read and write FIFOs.
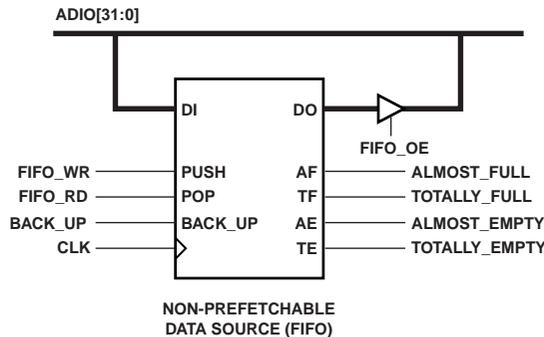
X8275

Figure 14-3   Non-Prefetchable Data Source

The most crucial element is the FIFO itself. The FIFO must have an additional control signal to back up, or "undo" up to two reads. A typical FIFO implemented in an FPGA consists of a circular buffer implemented with RAM, a read pointer, and a write pointer.

The back up feature can be incorporated in a FIFO by making the actual depth two less than the maximum possible, and by using a bidirectional read pointer. This prevents new data entering the FIFO from overwriting data in the FIFO that may need to be restored.

The FIFO must also have a set of flags that provide FIFO status information. These flags, and their uses, are listed below:

- TE -- this flag indicates a totally empty condition. If the FIFO is totally empty and an initiator write transaction is requested, the user application should either ignore the request, postpone it, or flag some sort of error.

- TF -- this flag indicates a totally full condition. If the FIFO is totally full and an initiator read transaction is requested, the user application should either ignore the request, postpone it, or flag some sort of error.

- AE -- this flag indicates an almost empty condition. When the FIFO becomes almost empty during an initiator write, the user application should signal that it wishes to complete the transaction because it is about to underrun the FIFO.

- AF -- this flag indicates an almost full condition. When the FIFO becomes almost full during an initiator read, the user application should signal that it wishes to complete the transaction because it is about to overrun the FIFO.

In most designs, the user application contains a transfer counter which specifies the length of the desired transfer. For designs with transfer lengths less than or equal to the FIFO size, it is not necessary to monitor the AE and AF flags from the FIFO. In this case, the transfer counter may be monitored to determine when the initiator should cease data transfer. That is, there is only one "terminating" condition which occurs when the transfer is complete.

For transfer lengths greater than the FIFO size, the AE and AF flags become important. In this case, the user application must be aware of two "terminating" conditions. One occurs when the transfer is complete, as determined by the transfer counter, and the other occurs when the FIFO becomes almost full (initiator read) or almost empty (initiator write).

In the second case, the FIFO must be emptied (initiator read) or refilled (initiator write) by logic in the user application before it requests another transaction to continue the transfer.

The FIFO design, as described above, may be coupled with a modified initiator control state machine, as discussed in the "Initiator Data Transfer and Control" chapter of this guide. Although the bulk of this example is similar, it is presented here in its entirety to present a complete picture. As before, the entire design breaks down into several distinct sections:

- Inputs to the state machine

- The state machine itself

- Outputs to the PCI interface

- Control signals

Inputs to the State Machine

The sample state machine requires one input to tell it to start, another input to indicate the direction of the desired transfer, and six additional inputs to effectively monitor the progression of the transaction. These signals are discussed below:

The START signal indicates to the state machine that it should begin a transaction. This signal is generated by the user application. The source of this signal will vary depending on the specific user application function. The logic in the user application that generates the START signal must not assert START in the following cases:

- Initiator write and FIFO empty

- Initiator read and FIFO full

Failure to observe this will lead to data loss or corruption. Once an initiator transaction is started, at least one data phase will complete.

The DIR signal indicates the direction of the desired transfer, and is generated by the user application. The DIR signal should not change from the time START is asserted until the end of the initiator transfer. Initiator writes are selected by driving DIR to logic high.

The signals **M_ADDR_N** and **M_DATA** are outputs of the PCI user interface that are used by the state machine. The signal M_DATA_FELL indicates that a falling edge has been detected on **M_DATA**. This is generated by the following code.

```
always @(posedge CLK or posedge RST)
begin : edge_detect
    if (RST) M_DATAQ = 1'b0;
    else M_DATAQ = M_DATA;
end


assign M_DATA_FELL = !M_DATA & M_DATAQ;
```

The FATAL signal is derived from PCI interface signals. The FATAL signal indicates that a master abort or target abort has occurred.

This information is obtained from the extended status signals, **CSR[39:32]**. The extended status information is valid one clock cycle after a particular event has occurred on the PCI Bus. Unless the status is used during that cycle, it must be registered to preserve it for

later use. The following code performs the task of preserving the status information from the final data phase.

```
always @(posedge CLK or posedge RST)
begin : watch_status
    if (RST) FATAL = 1'b0;
    else if (!M_ADDR_N) FATAL = 1'b0;
    else if (M_DATA) FATAL = CSR[39] | CSR[38];
end
```

In the previous example on single transfers, the transaction status was monitored to detect if the target signalled a retry. In this example, the state machine automatically requests transactions until the transfer counter is zero. The actual behavior can be modified to suit the user application by making changes to the state machine.

Note that the *PCI Local Bus Specification* requires initiators to repeat transactions terminated by the target with retry. However, for system robustness, it is advisable to have the initiator monitor the number of times it is retried, so that it can abort transfer attempts that are met with an excessive number of retries from the target.

The BACK_UP signal indicates that the FIFO must be backed up to compensate for speculative reads that can occur during initiator writes. The generation of this signal is discussed after the state machine is presented.

The final signal required by the state machine is the DONE signal. This signal indicates to the state machine that a transaction sequence has completed. This is asserted when the transfer length counter has reached zero.

```
always @(posedge CLK or posedge RST)
begin : transfer_counter
    if (RST) XFER_CNT = 4'h0;
    else if (START) XFER_CNT = BURST_LENGTH;
    else if (M_DATA_VLD) XFER_CNT = XFER_CNT - 4'h1;
end


assign DONE = (XFER_CNT == 4'h0);
```

This counter is loaded from a signal named BURST_LENGTH that is provided by the user application. Note that this counter does not need to be backed up, as it counts actual transfers, not anticipated transfers.

The State Machine

The state machine is shown in Figure 14-4, "User Application State Machine". This design is intended to perform initiator burst transactions of any length less than the size of the FIFO.
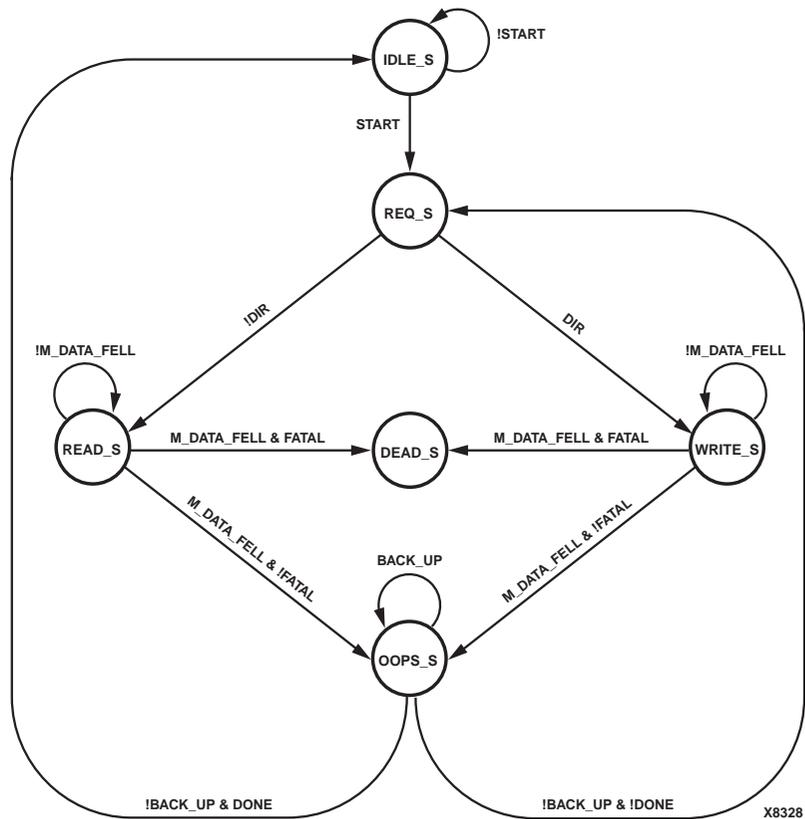

Figure 14-4   User Application State Machine

Each state performs a specific step in the sequence. The individual states and their purposes are listed below.

IDLE_S is the idle state. During this state, no initiator activity takes place, and the state machine waits for the user application to assert START. Once START has been asserted, the state machine proceeds to the REQ_S state.

REQ_S is the request state. During this state, a request is made to begin an initiator transaction. The state machine then branches based on the value of the DIR signal. If DIR indicates an initiator read, the next state is READ_S. Otherwise, the next state is WRITE_S.

READ_S and WRITE_S are data transfer states. The state machine stays in READ_S or WRITE_S until it detects that the transaction is over. The transaction is over when M_DATA_FELL is asserted. This can result from either premature target termination (retry or disconnect), the expiration of the internal latency timer, or natural transaction termination due to the assertion of **COMPLETE**. The **COMPLETE** signal is asserted by other logic that monitors FIFO status flags and the transfer counter.

These states are identical except that the state machine outputs differ. The output logic is discussed in the next section. Since DIR is available in the user application, these two states may be reduced into a single XFER_S state if DIR is used in the data register control equations. The states are separate in this example for clarity. In more elaborate designs, it is often desirable to split the state machine into separate read and write state machines.

Whether in READ_S or WRITE_S, the state machine branches after the transfer state based on the FATAL signal. If a fatal error has occurred, the state machine moves to the DEAD_S state. Otherwise, the state machine moves to the OOPS_S state.

DEAD_S is the terminal state for handling fatal errors. In practice, the user application should provide some method for resetting the state machine to handle such errors.

OOPS_S is the transaction evaluation state. If the FIFO must be backed up, it is done in this state. Upon exit from this state, the state machine evaluates the DONE signal. If DONE is asserted, the state machine transitions to IDLE_S, otherwise it transitions to the REQ_S state to request another transaction.

The state machine shown in Figure 14-4, "User Application State Machine" could be implemented as shown:

```
always @(posedge CLK or posedge RST)
```

```
begin : initiator_fsm
   if (RST) STATE = IDLE_S;
   else case (STATE)
       IDLE_S :  begin
                    if (START) STATE = REQ_S;
                    else STATE = IDLE_S;
                    end
       REQ_S :   begin
                    if (DIR) STATE = WRITE_S;
                    else STATE = READ_S;
                    end
       WRITE_S : begin
                    if (M_DATA_FELL)
                    begin
                        if (FATAL) STATE = DEAD_S;
                        else STATE = OOPS_S;
                    end
                    else STATE = WRITE_S;
                    end
       READ_S :  begin
                    if (M_DATA_FELL)
                    begin
                        if (FATAL) STATE = DEAD_S;
                        else STATE = OOPS_S;
                    end
                    else STATE = READ_S;
                    end
       DEAD_S :  STATE = DEAD_S;
       OOPS_S :  begin
                    if (BACK_UP) STATE = OOPS_S;
                    else if (DONE) STATE = IDLE_S;
                    else STATE = REQ_S;
                    end
```

```
          default : STATE = IDLE_S;
      endcase
end
```

Outputs to the PCI Interface

The state machine in the user application is responsible for driving several signals which are output to the PCI interface. These signals are discussed below along with the logic required to generate them. This discussion assumes the existence of a 32-bit START_ADDR signal provided by other logic in the user application. This signal represents the initial PCI Bus address for the desired target. The source of this signal will vary depending on the specific user application function.

Before requesting a transaction, the user application should indicate that it is ready to transfer data. Although it is possible to initiate a transaction and then insert wait states using the **M_READY** signal, bus bandwidth is conserved by requesting a transaction only after the user application is ready to transfer data. In this example, the user application is always ready.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized_a
   if (RST) M_READY = 1'b0;
   else M_READY = 1'b1;
end
```

The **M_READY** signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

The state machine initially requests access to the PCI Bus by asserting **REQUEST**. This is done in the REQ_S state and results in the assertion of REQ_O. The **REQUEST** signal must only be asserted for a single cycle to initialize a request. Note that the bus master enable bit in the command register must be set before the PCI interface is able to request the bus. This is the responsibility of the host bridge and system configuration software.

```
assign REQUEST = (STATE == REQ_S);
```

The **REQUESTHOLD** signal is not used in this example, and is simply assigned to logic zero.

```
assign REQUESTHOLD = 1'b0;
```

When the PCI interface indicates that it has received a bus grant, the user application must drive the target address on the **ADIO** bus. This address, stored in the initiator address pointer, is presented on the PCI Bus during the address phase.

```
always @(posedge CLK or posedge RST)
begin : init_addr_pointer
    if (RST) ADDRESS = 30'h0;
    else if (START) ADDRESS = START_ADDR[31:2];
    else if (M_DATA_VLD) ADDRESS = ADDRESS + 30'h1;
end


assign ADDR_OE = M_ADDR_N;
assign ADIO = ADDR_OE ? 32'bz : {ADDRESS, 2'b00};
```

Simultaneously, the user application must also drive **M_CBE** to indicate the desired PCI Bus command. At other times, the **M_CBE** signal is used to indicate byte enables. In this example, the initiator performs memory read and memory write transactions (commands 0x6 and 0x7, respectively), depending on the value of the DIR signal. All bytes are enabled. The bus command and byte enables are easily modified to accommodate the requirements of the user application.

```
assign COMMAND = {3'b011, DIR};
assign BYTE_ENABLE = 4'b0000;
assign M_CBE = M_ADDR_N ? BYTE_ENABLE : COMMAND;
```

Throughout the entire transaction, the user application must drive **M_WRDN** to indicate the desired transfer direction. This signal must remain constant during a transaction. By assigning it to track the DIR signal, which was stipulated to be constant during a transaction, this requirement is met.

```
assign M_WRDN = DIR;
```

The **M_WRDN** signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, although it is permitted to drive it from combinational logic.

Note: Never tie this signal to logic one or to logic zero.

The **COMPLETE** signal indicates to the PCI interface that it should finish the current transaction. This is determined by the current transfer counter value. The logic presented below is derived from the example in an earlier chapter on data phase control.

```
assign CNT3 = (XFER_CNT == 4'h3);
assign CNT2 = (XFER_CNT == 4'h2);
assign CNT1 = (XFER_CNT == 4'h1);
```

In this example, initiator wait states are not used. It is critical to reduce the logic as much as possible for timing reasons.

```
assign FIN1 = CNT1 & REQUEST;
assign FIN2 = CNT2 & M_DATAQ;
assign FIN3 = CNT3 & M_DATA_VLD;
assign ASSERT_COMPLETE = FIN1 | FIN2 | FIN3;


always @(posedge CLK or posedge RST)
begin : finish_up
    if (RST) HOLD_COMPLETE = 1'b0;
    else if (M_DATA_FELL) HOLD_COMPLETE = 1'b0;
    else if (ASSERT_COMPLETE) HOLD_COMPLETE = 1'b1;
end


assign COMPLETE = ASSERT_COMPLETE | HOLD_COMPLETE;
```

The **COMPLETE** signal must not be assigned a static value. For timing reasons, this signal should be driven from the output of a flip-flop, if possible. In practice, it is usually driven by combinational logic.

Note: Never tie this signal to logic one or to logic zero.

To determine the number of times the FIFO must be backed up after an initiator write, the user application must include a small state machine to monitor the difference between anticipated transfers and actual transfers.

Anticipated transfers are signalled by **M_SRC_EN**, qualified by the TE flag, as an empty FIFO cannot be advanced. Actual transfers are signalled by **M_DATA_VLD**.

```
assign ANTICIPATED = M_SRC_EN & !TE;

always @(posedge CLK or posedge RST)
begin : oops_counter
   if (RST) OOPS = 2'b00;
   else
   case({ANTICIPATED, M_DATA_VLD, BACK_UP, OOPS})
      5'b00000: OOPS = 2'b00;
      5'b00001: OOPS = 2'b01;
      5'b00010: OOPS = 2'b10;
      5'b00011: OOPS = 2'b11;
      5'b00100: OOPS = 2'b00;
      5'b00101: OOPS = 2'b00;
      5'b00110: OOPS = 2'b01;
      5'b00111: OOPS = 2'b10;
      5'b01000: OOPS = 2'b00;
      5'b01001: OOPS = 2'b00;
      5'b01010: OOPS = 2'b01;
      5'b01011: OOPS = 2'b10;
      5'b01100: OOPS = 2'b00;
      5'b01101: OOPS = 2'b00;
      5'b01110: OOPS = 2'b00;
      5'b01111: OOPS = 2'b01;
      5'b10000: OOPS = 2'b01;
      5'b10001: OOPS = 2'b10;
      5'b10010: OOPS = 2'b11;
      5'b10011: OOPS = 2'b11;
      5'b10100: OOPS = 2'b00;
      5'b10101: OOPS = 2'b01;
      5'b10110: OOPS = 2'b10;
      5'b10111: OOPS = 2'b11;
      5'b11000: OOPS = 2'b00;
      5'b11001: OOPS = 2'b01;
```

```
            5'b11010: OOPS = 2'b10;

            5'b11011: OOPS = 2'b11;

            5'b11100: OOPS = 2'b00;

            5'b11101: OOPS = 2'b00;

            5'b11110: OOPS = 2'b01;

            5'b11111: OOPS = 2'b10;

            default : OOPS = 2'b00;

        endcase

    end


    assign BACK_UP = (| OOPS) & (STATE == OOPS_S);
```

This state machine describes a two bit saturating up/down counter
with one increment input and two decrement inputs. As required, it
tracks the number of actual transfers versus the number of antici-
pated transfers. The BACK_UP signal is asserted when the state
machine is in the OOPS_S state and the OOPS counter is non-zero.
This backs up the FIFO and decrements the OOPS counter until this
counter is zero. Once BACK_UP is deasserted, the state machine
proceeds to the next state.

Control Signals

As a final consideration, the logic for the FIFO control signals as
shown in Figure 14-3, "Non-Prefetchable Data Source" are as follows:

```
    assign FIFO_WR = (STATE == READ_S) & M_DATA_VLD;

    assign FIFO_RD = (STATE == WRITE_S) & M_SRC_EN;

    assign FIFO_OE = (STATE == WRITE_S) & M_DATA;
```

The generation of the BACK_UP signal, which is also used by the state
machine, was presented previously.

Sample Transactions

The following figures illustrate typical burst read and write transac-
tions that result from implementing the logic presented in this
chapter.
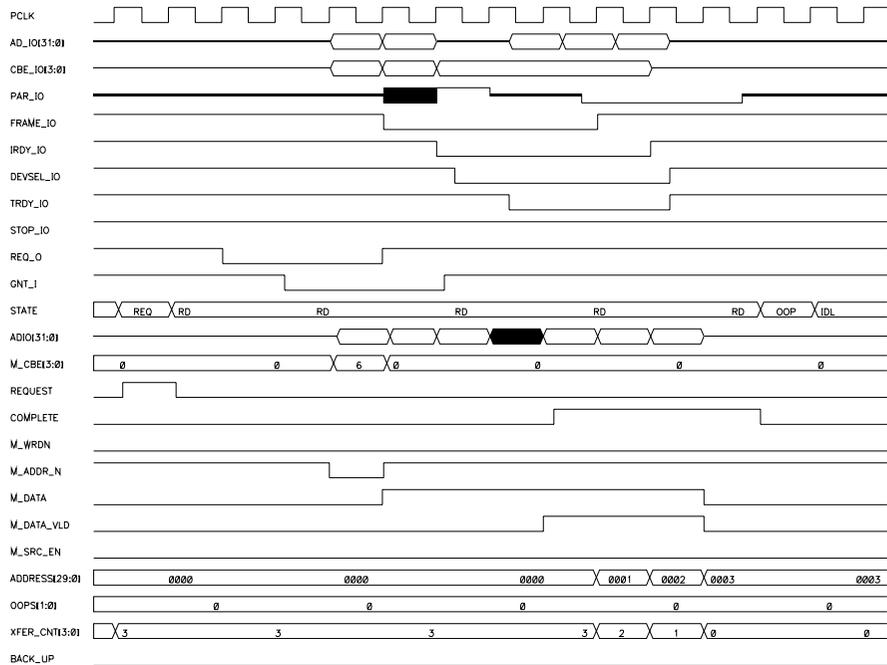
Figure 14-5   Burst Read Transaction with Normal Termination

Figure 14-5, "Burst Read Transaction with Normal Termination", demonstrates the PCI interface filling the FIFO in the user application. The target does not terminate the transaction. This figure also shows the timing relationship between selected user application signals and PCI Bus signals.
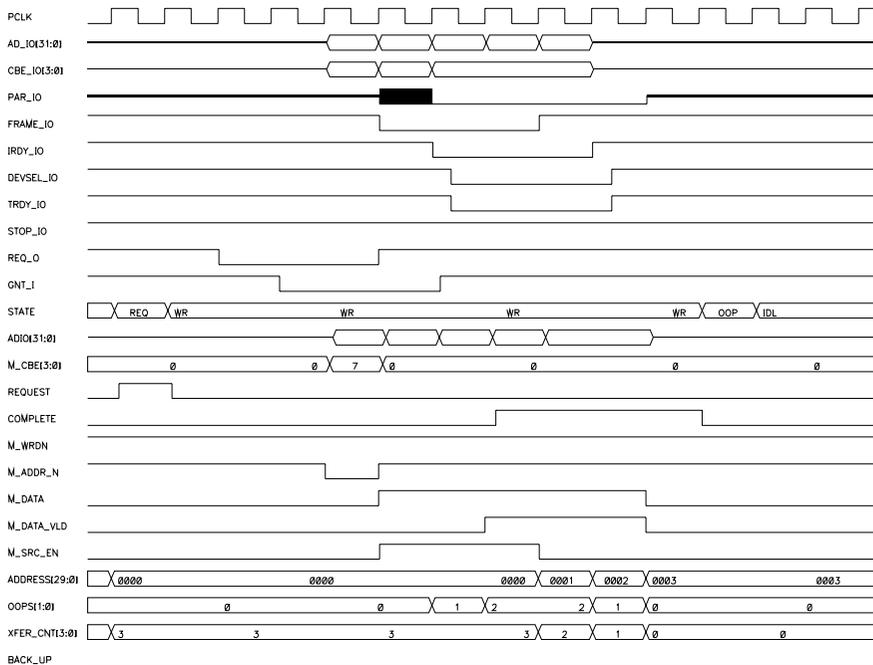
Figure 14-6   Burst Write Transaction with Normal Termination

Figure 14-6, "Burst Write Transaction with Normal Termination",
demonstrates the PCI interface emptying the FIFO in the user appli-
cation. The target does not terminate the transaction. This figure also
shows the timing relationship between selected user application
signals and PCI Bus signals.
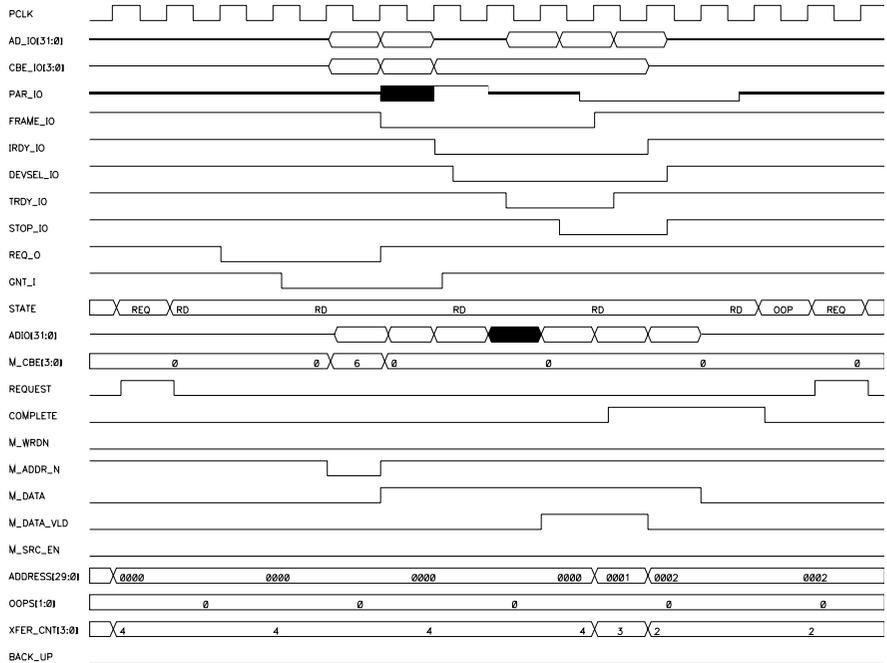
Figure 14-7   Burst Read with Target Disconnect

Figure 14-7, "Burst Read with Target Disconnect", demonstrates the PCI interface filling the FIFO in the user application. During the transaction, the target disconnects.
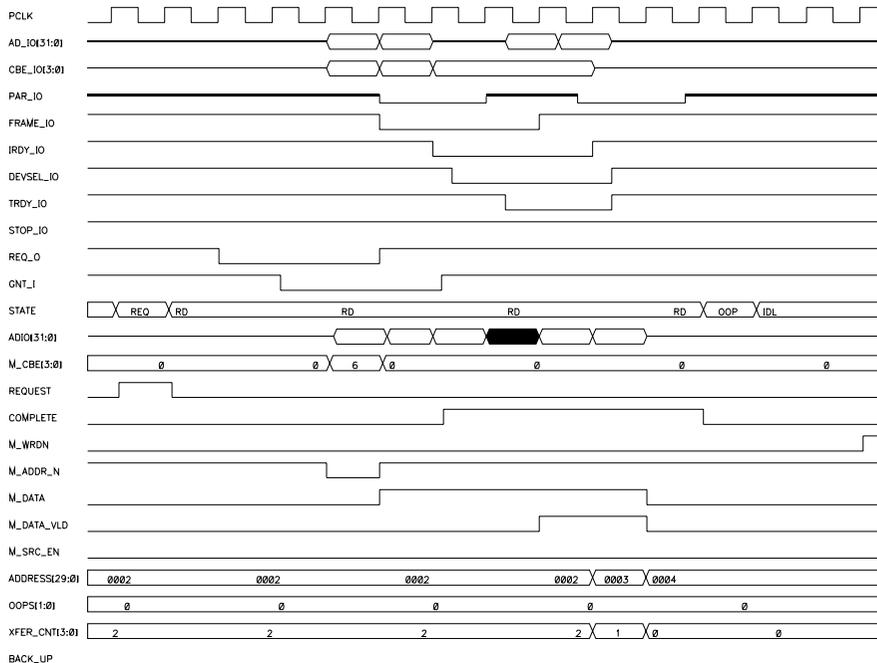
Figure 14-8   Burst Read Continues after Target Disconnect

Figure 14-8, "Burst Read Continues after Target Disconnect" shows the user application requesting bus access again to complete the transfer.
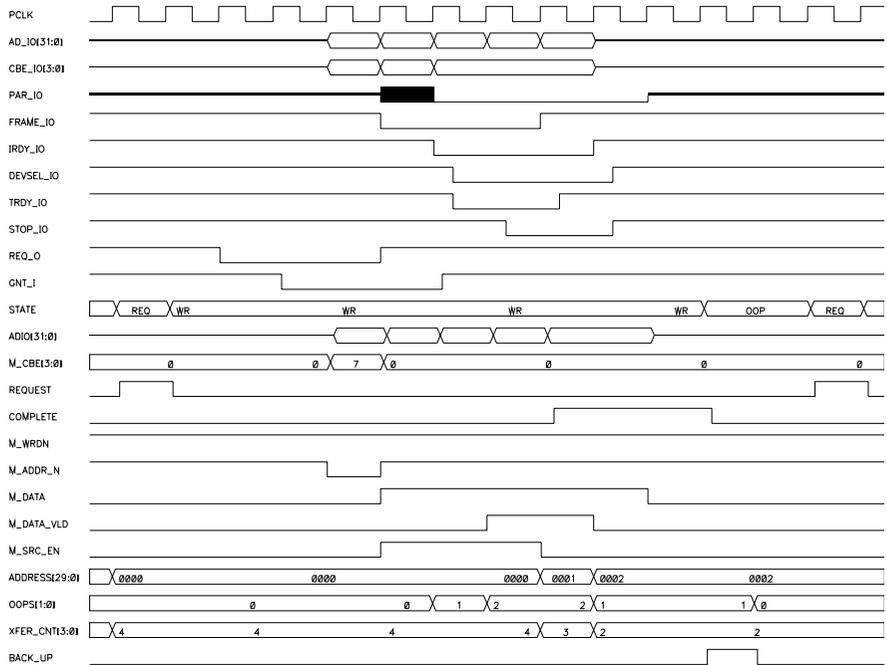
Figure 14-9   Burst Write with Target Disconnect

Figure 14-9, "Burst Write with Target Disconnect", demonstrates the PCI interface emptying the FIFO in the user application. During the transaction, the target disconnects. Notice that the FIFO must be backed up in this situation.
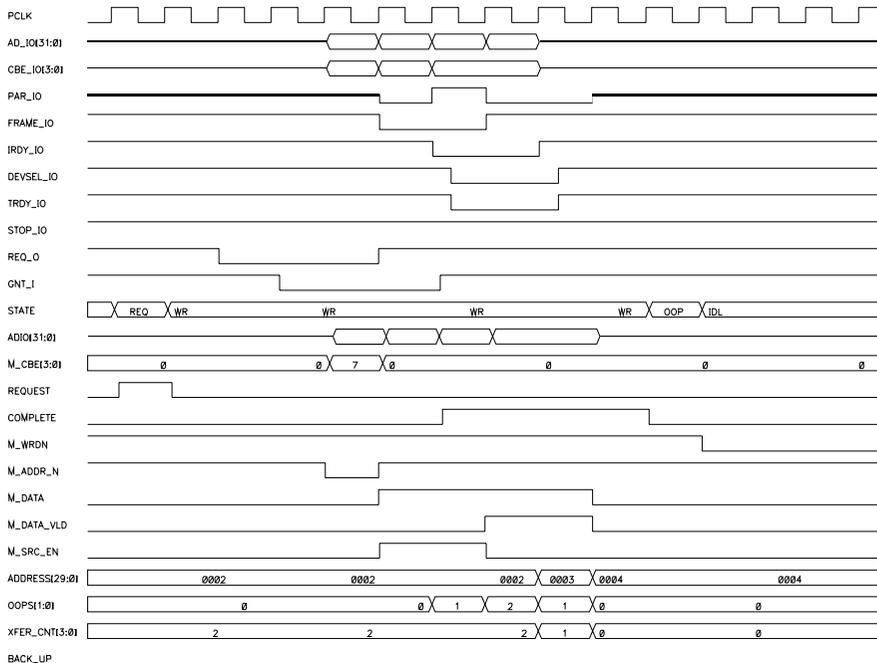
Figure 14-10   Burst Write Continues after Target Disconnect

Figure 14-10, "Burst Write Continues after Target Disconnect" shows the user application requesting bus access again to complete the transfer.

Chapter 15

Initiator 64-bit Extension

This chapter provides additional details about the initiator 64-bit extension. Implementation of a user application which is a 64-bit initiator is similar to the implementation of a 32-bit version. In addition to the wider data path, some provisions must be made for exceptional conditions which may arise during 64-bit transfers. Note that 64-bit transfers only apply to memory spaces. Other spaces do not support this capability.

Initiator Extension Signals

In addition to the initiator signals presented in the "Initiator Data Transfer and Control" chapter of this guide, there are a few additional signals present in 64-bit implementations of the LogiCORE PCI interface.

- **ADIO[63:32]** -- this bidirectional bus provides the means for 64-bit data and address transfer to and from the PCI interface. When the initiator start address is presented during the assertion of **M_ADDR_N** by the PCI interface, the high portion of the bus is reserved but must be driven with valid data. When driving data onto the bus, the entire bus width should be driven. Note that 64-bit addressing and dual address cycles are not supported.

- **M_CBE[7:4]** -- this output indicates the PCI command and byte enables during an initiator transaction. When the command is presented during the assertion of **M_ADDR_N** by the PCI interface, the high nibble is reserved but must be driven with valid data. During data phases, this bus should be driven with the byte enables for the extended data path.

- **SLOT64** -- this signal enables the 64-bit extension signals (AD_IO[63:32], CBE_IO[7:4], and PAR64_IO). This signal must remain static at all times except at power-on and immedi-

ately after a PCI Bus reset. The method for determining the appropriate value for this signal is design and device family dependent. Refer to the *LogiCORE PCI Implementation Guide* for more details about **SLOT64**.

• **REQUEST64** -- this output from the user application instructs the PCI interface to request the PCI Bus and to begin a 64-bit initiator transaction once GNT_I is asserted.

• **M_FAIL64** -- this input indicates that a 64-bit transfer attempt has encountered a 32-bit target. In such situations, the initiator will transfer at most two 32-bit words before terminating the transfer. This signal should be used to adjust the increment (step size) of any initiator address pointers.

Controlling 64-bit Transfers

The concepts presented in earlier chapters apply to 64-bit transfers. The steps for transfer requests and data phase control are the same. Extending the data path of the example presented in the "Initiator Data Phase Control" chapter of this guide yields the following waveforms.
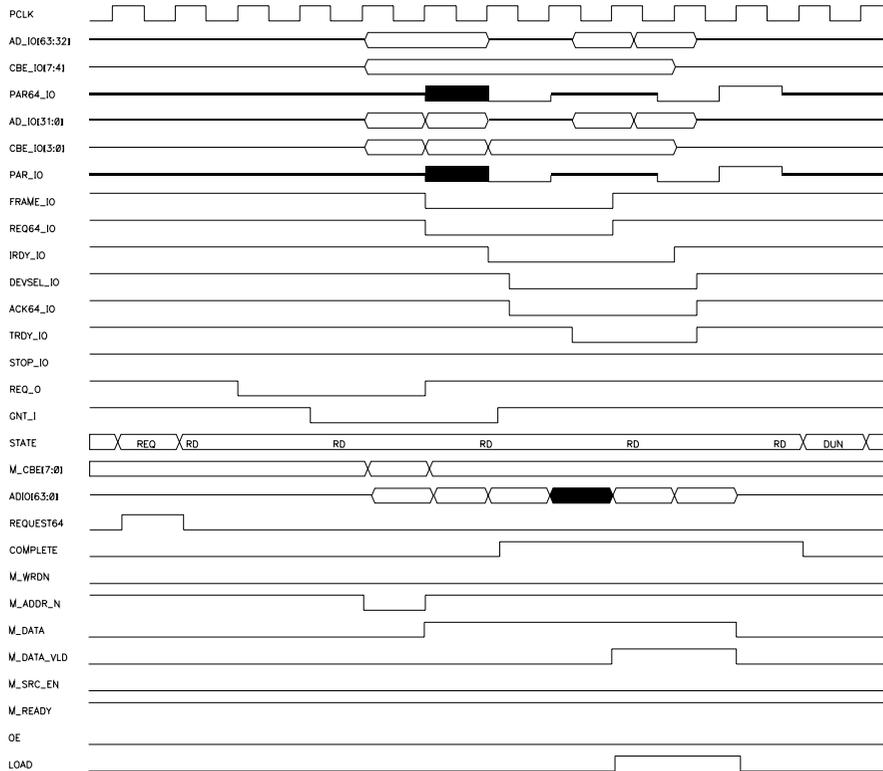
Figure 15-1   Two QWORD Initiator Read Transfer

Figure 15-1, "Two QWORD Initiator Read Transfer" demonstrates the PCI interface performing a burst read transfer from a 64-bit target. In this case, the **REQUEST64** signal is asserted instead of the **REQUEST** signal The **COMPLETE** logic remains the same, but the transfer counter (and any address pointer) now must track QWORDs instead of DWORDs.
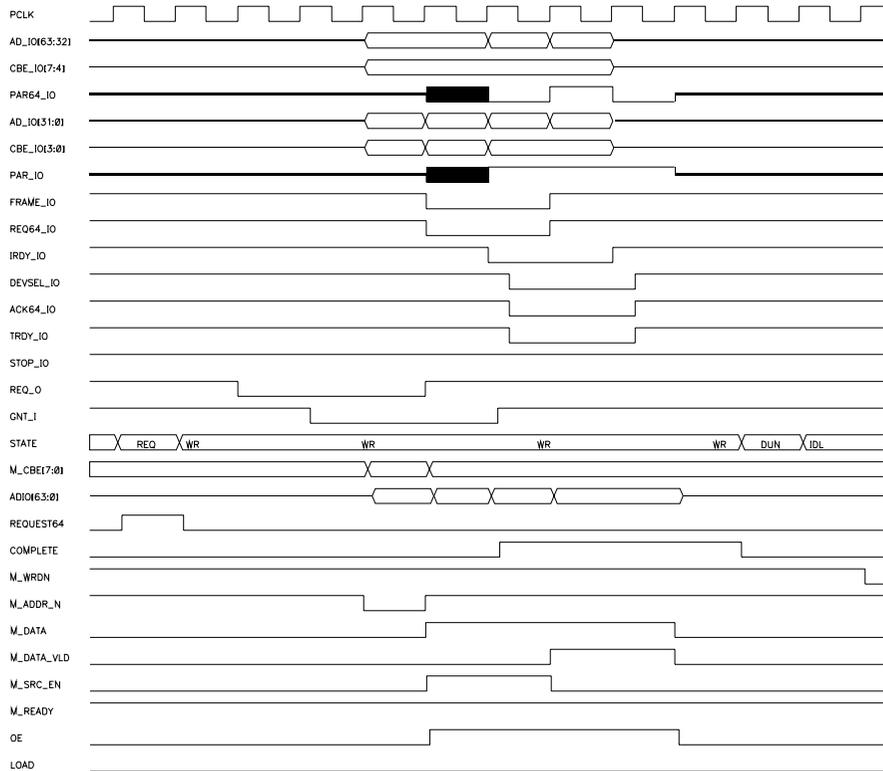
Figure 15-2   Two QWORD Initiator Write Transfer

Figure 15-2, "Two QWORD Initiator Write Transfer" demonstrates the PCI interface performing a burst write transfer to a 64-bit target. Again, the **REQUEST64** signal is used and the transfer counter (and any address pointer) tracks QWORDs instead of DWORDs.

Additional Considerations

The control of 64-bit initiator transfers is similar to the 32-bit case. However, the flexibility of the PCI interface opens the door to potential protocol violations by the user application. For this reason, the following points must be considered when designing an initiator control state machine.

Only Perform Burst Transfers

Do not request single data phase 64-bit transactions. The PCI interface does not know if a 64-bit or a 32-bit target will respond to a 64-bit transaction request. For this reason, the interface does not know when to deassert FRAME_IO and REQ64_IO for a single data phase transfer. Instead, request a 32-bit transfer and perform two data phases.

Only Use Aligned Addresses

The *PCI Local Bus Specification* forbids initiators from requesting unaligned 64-bit transfers. The user application should never initiate a 64-bit transfer with an address that is not aligned on a QWORD boundary. Specifically, the low three address bits must be zero.

Only Use Allowed Commands

The *PCI Local Bus Specification* states that 64-bit transfers apply to memory spaces only. The user application should never initiate a 64-bit transfer using non-memory commands. As a 64-bit initiator, the LogiCORE PCI interface supports the following commands:

- Memory Read

- Memory Write

- Memory Read Multiple

- Memory Read Line

Monitor the Target Response

Under ideal conditions, a 64-bit target will respond to a 64-bit transfer request by the PCI interface. Unfortunately, this response is not guaranteed in an open system. A 32-bit response may occur when:

- The target does not support 64-bit transfers

- The PCI interface is installed in a 32-bit system

In either case, the PCI interface perceives the target as a 32-bit agent on the bus. When the LogiCORE PCI interface encounters a 32-bit target as a 64-bit initiator, it will automatically terminate the transfer after two 32-bit data phases. Depending on the behavior of the target, different results are possible.

If the target terminates the transfer attempt with retry, the user application is required to retry the original 64-bit transaction exactly as it occurred the first time -- even though the user application may now be aware that the target is not 64-bit.

When the PCI interface detects a 32-bit target responding to a 64-bit transfer attempt, it asserts **M_FAIL64**. The user application must use this signal to adjust the increment (step size) for any internal counters or pointers. This includes address pointers, data pointers, and back up counters. Once the transfer completes, it is the responsibility of the user application to request another transfer and continue as a 32-bit agent.

The following waveforms are an example of possible situations which may arise when a 64-bit transfer request is met with a 32-bit target response. The data path signals in these figures are broken into high and low halves to demonstrate the actual data movement during transfers where **M_FAIL64** is asserted by the PCI interface.
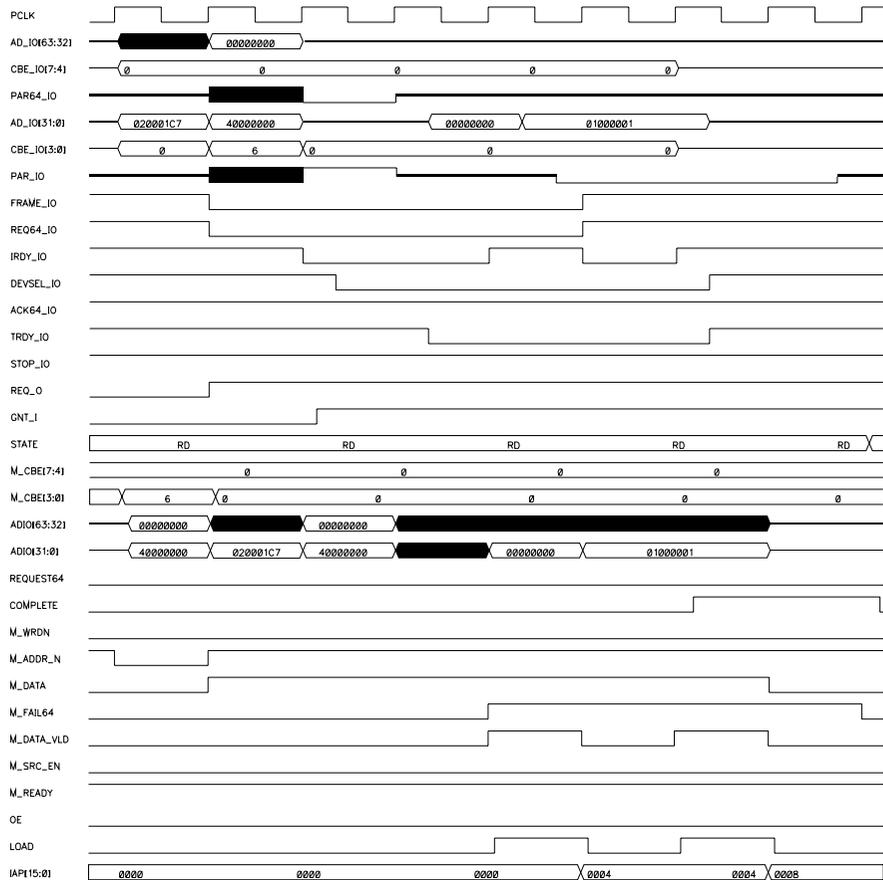
Figure 15-3   32-bit Target Responds to 64-bit Read

Figure 15-3, "32-bit Target Responds to 64-bit Read" shows the PCI interface requesting a 64-bit read burst. The target is a 32-bit agent and does not assert ACK64_IO in response to REQ64_IO.

Once the PCI interface detects that the target is a 32-bit agent, it asserts **M_FAIL64**. The PCI interface automatically inserts a wait state after the first data phase. During that wait state, the interface automatically copies the byte enables on CBE_IO[7:4] to CBE_IO[3:0]. Then the PCI interface completes the transfer on the second data phase. In this example, the target does not disconnect.

The user application must monitor **M_FAIL64** during a transfer to correctly interpret the transaction. If **M_FAIL64** is not asserted, the user application may complete the transfer using the full 64-bit data path. If **M_FAIL64** is asserted, the user application should treat the transfer as a 32-bit transfer. This implies:

• Capturing data from **ADIO[31:0]** and ignoring **ADIO[63:32]**

• Assembling data into 64-bit chunks, if required

• Adjusting the increment (step size) for any counters or pointers

At the end of this transfer, the bus address is QWORD aligned. Although the *PCI Local Bus Specification* permits this transfer to be continued by requesting another 64-bit transfer, it is unwise to do so from a bandwidth perspective. Once the target is known to be a 32-bit agent, the user application should request a 32-bit transfer.

Figure 15-4   32-bit Target Disconnects with Data on Read

Figure 15-4, "32-bit Target Disconnects with Data on Read" demonstrates a transaction that is similar to Figure 15-3, "32-bit Target Responds to 64-bit Read" with the exception that the target disconnects after the first data phase.

Once the PCI interface detects that the target is a 32-bit agent, it asserts `M_FAIL64`. Since the target disconnects, the PCI interface does not insert a wait state to multiplex the byte enables for the second data phase.

As before, the user application must monitor **M_FAIL64** during the transfer to correctly interpret the transaction. If **M_FAIL64** is not asserted, the user application may complete the transfer using the full 64-bit data path. If **M_FAIL64** is asserted, the user application should treat the transfer as a 32-bit transfer. This implies:

- Capturing data from **ADIO[31:0]** and ignoring **ADIO[63:32]**

- Assembling data into 64-bit chunks, if required

- Adjusting the increment (step size) for any counters or pointers

At the end of this transfer, the bus address is *not* QWORD aligned. The *PCI Local Bus Specification* stipulates that this transfer must be continued as a 32-bit transfer due to the misalignment.
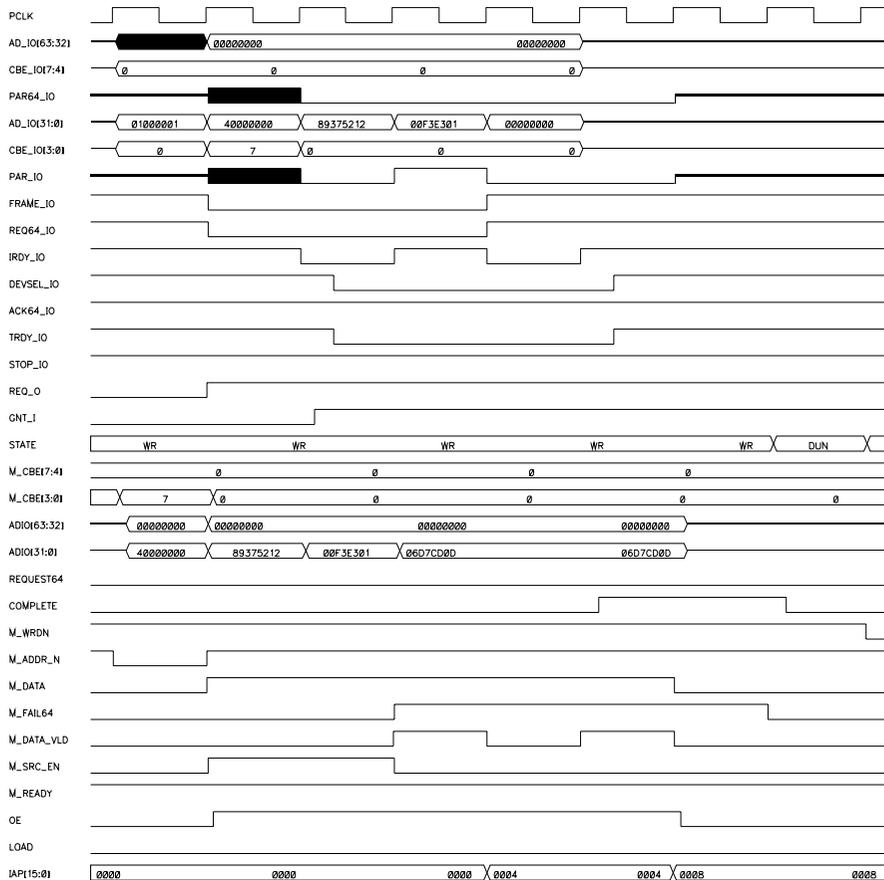
Figure 15-5   32-bit Target Responds to 64-bit Write

Figure 15-5, "32-bit Target Responds to 64-bit Write" shows the PCI interface requesting a 64-bit write burst. The target is a 32-bit agent and does not assert `ACK64_IO` in response to `REQ64_IO`.

Once the PCI interface detects that the target is a 32-bit agent, it asserts **M_FAIL64**. The PCI interface automatically inserts a wait state after the first data phase. During that wait state, the interface automatically copies the byte enables on `CBE_IO[7:4]` to `CBE_IO[3:0]`. In the next cycle, the interface copies the data from `AD_IO[63:32]` to `AD_IO[31:0]`. Then the PCI interface completes

the transfer on the second data phase. In this example, the target does not disconnect.

Since **M_FAIL64** is asserted, the user application must treat the transfer as a 32-bit transfer and adjust the increment (step size) for any counters or pointers. The PCI interface automatically handles the data multiplexing internally when performing initiator writes.

At the end of this transfer, the bus address is QWORD aligned. Although the *PCI Local Bus Specification* permits this transfer to be continued by requesting another 64-bit transfer, it is unwise to do so from a bandwidth perspective. Once the target is known to be a 32-bit agent, the user application should request a 32-bit transfer.
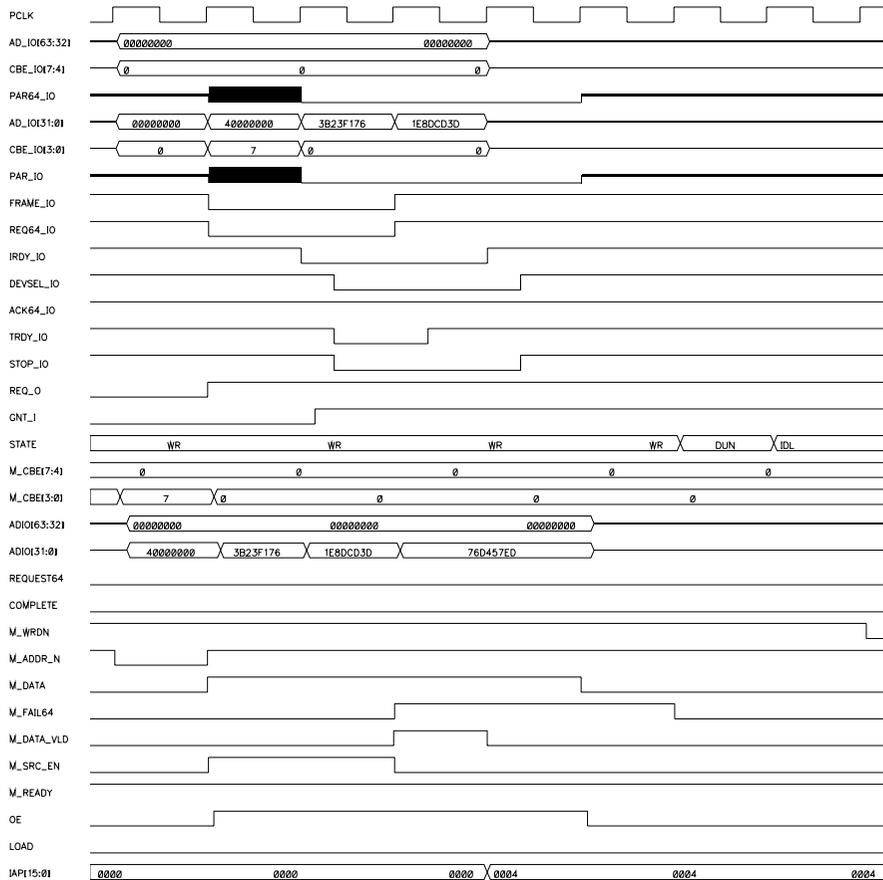
Figure 15-6   32-bit Target Disconnects with Data on Write

Figure 15-6, "32-bit Target Disconnects with Data on Write" demon-
strates a transaction that is similar to Figure 15-5, "32-bit Target
Responds to 64-bit Write" with the exception that the target discon-
nects after the first data phase.

Once the PCI interface detects that the target is a 32-bit agent, it
asserts **M_FAIL64**. Since the target disconnects, the PCI interface
does not insert a wait state to multiplex the data and byte enables for
the second data phase.

Since **M_FAIL64** is asserted, the user application must treat the transfer as a 32-bit transfer and adjust the increment (step size) for any counters or pointers. The PCI interface automatically handles the data multiplexing internally if the target does not disconnect.

At the end of this transfer, the bus address is *not* QWORD aligned. The *PCI Local Bus Specification* stipulates that this transfer must be continued as a 32-bit transfer due to the misalignment.

Chapter 16

Other Bus Cycles

The purpose of this chapter is to demonstrate the use of more esoteric commands with the PCI interface. In addition to supporting memory read and memory write commands, the LogiCORE PCI interface supports a host of other PCI Bus commands as both a target and an initiator. Many of these commands do not require additional design effort. The examples presented in earlier chapters of this book may be easily modified to support other commands.

As a target, the PCI interface can support I/O commands if at least one of the Base Address Registers is configured as an I/O space. Additionally, the PCI target can handle all extended memory commands such as memory read multiple, memory read line, and memory write and invalidate, as long as at least one Base Address Register is configured as a memory space.

As an initiator, the PCI interface can issue both I/O commands and all memory commands except memory write and invalidate. The user application simply presents the appropriate command to the PCI interface when initiating a transfer. Note that I/O commands may not be used with multiple data phase transfers, as required by the *PCI Local Bus Specification*.

Supported Commands

Table 16-1, "Supported PCI Bus Commands" provides a list of commands supported by the LogiCORE PCI interface. The first

portion of this chapter discusses target operations, while the second
portion discusses initiator operations.

Table 16-1   Supported PCI Bus Commands

Command		Support	
Code	Name	Target	Initiator
0000	Interrupt Acknowledge	Yes	Yes
0001	Special Cycle	Ignore	Yes
0010	I/O Read	Yes	Yes
0011	I/O Write	Yes	Yes
0100	Reserved	Ignore	No
0101	Reserved	Ignore	No
0110	Memory Read	Yes	Yes
0111	Memory Write	Yes	Yes
1000	Reserved	Ignore	No
1001	Reserved	Ignore	No
1010	Configuration Read	Yes	Yes
1011	Configuration Write	Yes	Yes
1100	Memory Read Multiple	Yes	Yes
1101	Dual Address Cycle	Ignore	No
1110	Memory Read Line	Yes	Yes
1111	Memory Write Invalidate	Yes	No

Note: Commands listed as "Yes" indicate full support. Those listed as
"Ignore" are not supported by the PCI target and are ignored. Those
listed as "No" are not supported by the PCI initiator and must not be
used.

Target Interrupt Acknowledge

Interrupt acknowledge commands issued on the PCI Bus are implicitly addressed to the system interrupt controller. These bus cycles are typically initiated by the host bridge.

In designs where the user application contains the system interrupt controller, it is necessary for the PCI interface to respond to interrupt acknowledge cycles as a target. This behavior is enabled using the "Interrupt Acknowledge" configuration option as discussed in the "Customizing the LogiCORE PCI Interface" chapter of this guide. Enabling this option forces the last available Base Address Register in the PCI interface to respond to interrupt acknowledge cycles.

```
always @(posedge CLK or posedge RST)
begin : decode_int_ack
    if (RST) INT_BAR = 1'b0;
    else if (BASE_HIT[2]) INT_BAR = 1'b1;
    else if (!S_DATA) INT_BAR = 1'b0;
end
```

For this example, assume the existence of a 32-bit signal named VECTOR which holds the interrupt vector. This signal is generated elsewhere in the user application.

```
assign ACK_OE = INT_BAR & S_DATA;
assign ADIO = ACK_OE ? VECTOR : 32'bz;
```

As a general rule, interrupt acknowledge cycles may be terminated like other bus cycles. For this example, however, the transfer is not terminated by the PCI interface.

```
always @(posedge CLK or posedge RST)
begin : cannot_be_optimized
    if (RST)
    begin
        S_ABORT = 1'b1;
        S_READY = 1'b0;
        S_TERM = 1'b1;
    end
    else
```

```
begin
    S_ABORT = 1'b0;
    S_READY = 1'b1;
    S_TERM = 1'b0;
end
end
```

The code above generates the waveform shown Figure 16-1, "Target Accepts Interrupt Acknowledge Cycle". In this situation, the interrupt vector is returned to the host bridge during the cycle.

Figure 16-1   Target Accepts Interrupt Acknowledge Cycle

Target Configuration Cycles

The *PCI Local Bus Specification* defines three separate address spaces:

- Memory space
- I/O space
- Configuration space

The LogiCORE PCI interface, by default, implements the first 64 bytes of a type zero configuration space header, shown in Table 16-2 "PCI Configuration Space Header". Additional configuration space is available if the "User Config Space" option is enabled in the configuration file.

Table 16-2 PCI Configuration Space Header

31	16	15	0	
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Rev ID	08h
BIST	Header Type	Lat. Timer	Line Size	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Base Address Register 2				18h
Base Address Register 3				1Ch
Base Address Register 4				20h
Base Address Register 5				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved			CapPtr	34h
Reserved				38h
Max_Lat	Min_Gnt	Intr. Pin	Intr. Line	3Ch
User Config Space Begins				40h

All unimplemented configuration space registers return a value of zero during configuration read cycles and no operation occurs during configuration write cycles.

Shaded address locations are not implemented in the LogiCORE PCI interface default configuration. These locations return zero during configuration read accesses. Other locations, such as the CapPtr, or Base Address Registers, will return zeros if disabled.

This section discusses some features of the configuration space that exist in the PCI interface, and presents methods for handling other configuration space accesses. Typically, user application designs will not need to extend the configuration space that is pre-implemented in the PCI interface.

This information is intended mainly for advanced users who wish to implement the user accessible area of the configuration space. This section is divided into the following sub-sections:

- Setup for Typical Applications

- User Definable Configuration Space

- Capabilities List Pointer

- Externally Supplied Subsystem Identification

Setup for Typical Applications

Configuration transactions are automatically handled by the PCI interface. The signals **C_TERM** and **C_READY** must be asserted by the user application at all times to allow the PCI interface to respond to the supported configuration space addresses. All unsupported addresses return zero when accessed, as mandated by the *PCI Local Bus Specification.*

For a typical application that uses only the pre-implemented configuration space, **C_TERM** and **C_READY** must always be asserted to ensure that the PCI interface terminates all configuration accesses by disconnecting with data on the first data phase. The PCI interface does not handle burst configuration cycles and must disconnect.

```
assign C_READY = 1'b1;
assign C_TERM = 1'b1;
```

Note: The **ADIO** bus is actively driven by the PCI interface during configuration cycles. Care should be taken as to avoid contention on the **ADIO** bus.

User Definable Configuration Space

The user definable region of configuration space resides at and above address 0x40, up to the top of configuration space at 0xFF. Access to this region is controlled by the "User Config Space" option in the configuration file.

With this option disabled, the PCI interface returns zeros when this region is accessed. If the option is enabled, the PCI interface does not automatically return zeros, but instead allows the user application to treat the configuration access like any other non-burst target access.

Data is transferred much like non-burst target accesses. A typical configuration register is interfaced as shown in Figure 16-2, "Example Configuration Register".
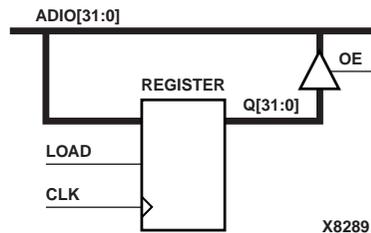
Figure 16-2   Example Configuration Register

The following signals are associated with configuration data transfer to and from the PCI interface. These signals are used in conjunction with other signals used for target data transfer. References to inputs and outputs are made with respect to the user application.

• **CFG_HIT** -- this input indicates that the PCI interface recognizes that it is the target of a current PCI configuration transaction. This signal is similar to the **BASE_HIT** signals used in target accesses.

• **CFG_VLD** -- this input indicates that a valid PCI configuration address is available on the **ADIO** bus, and may be used as a clock enable by the user application to capture a copy of this address. As the PCI interface does not support configuration burst transactions, the latched address present on the **ADDR[31:0]** bus should suffice. The **CFG_VLD** signal is asserted for a single cycle, coincident with **ADDR_VLD**.

- **C_READY** -- this output from the user application indicates that it is ready to transfer data, and can be used to insert wait states during the first data phase of a transaction. Together with **C_TERM**, it is also used to signal different types of target termination for configuration accesses. This signal has the same functionality as **S_READY** for target accesses.

- **C_TERM** -- this output from the user application indicates that data transfer should cease. It is also used with **C_READY** to signal different types of target termination for configuration accesses. This signal has the same functionality as **S_TERM** for target accesses.

The Verilog pseudocode to decode configuration transactions can be represented as follows:

```
always @(posedge CLK or posedge RST)
begin : decode_cfgspace
    if (RST)
    begin
        CFG_RD = 1'b0;
        CFG_WR = 1'b0;
    end
    else
    begin
        if (CFG_HIT)
        begin
            CFG_RD = !S_WRDN;
            CFG_WR = S_WRDN;
        end
        else if (!S_DATA)
        begin
            CFG_RD = 1'b0;
            CFG_WR = 1'b0;
        end
    end
end
```

This is nearly identical to the decoding used for normal target transactions. In this case, the PCI Bus command has been pre-decoded as a configuration read or configuration write command through the logic that generates **CFG_HIT**.

Configuration Writes

During a configuration write operation, data is captured from the **ADIO** bus to a data register in the user application by asserting the load input. The assignment for the load input of the register would be:

```
assign LOAD = CFG_WR & S_DATA_VLD & fn(ADDR[7:2]);
```

If the user application supports byte-addressable registers, separate load signals should be generated for each byte in the register by further gating the expression shown above. The **S_CBE** signals are available for this purpose.

Configuration Reads

During a target read operation, data from the user application is driven onto the **ADIO** bus. To do this, the user application must assert the output enable for the desired register. The assignment for the output enable would then be:

```
assign OE = CFG_RD & fn(ADDR[7:2]) & S_DATA;
```

Note: Do not drive the **ADIO** bus from the user application when the configuration transaction is not addressing user configuration space. The PCI interface responds to addresses below 0x40 and will drive **ADIO** during accesses of this region. When the address is 0x40 and above, always drive the entire **ADIO** bus with valid data. Unimplemented regions must return zero.

Configuration Data Phase Control

Data phase control is achieved using the **C_TERM** and **C_READY** signals. Combinations of the two control signals yield the following three modes:

• Wait -- the wait mode inserts wait states at the beginning of a configuration transaction.

• Retry -- this mode terminates the current PCI Bus transaction without data transfer on the final data phase.

- Disconnect with data -- this mode terminates the current PCI Bus transaction with data transfer on the first data phase.

Table 16-3, "Configuration Data Phase Control", shows the three modes of operation and the corresponding `C_TERM` and `C_READY` values. Never assert `C_READY` while `C_TERM` is deasserted, as the LogiCORE target does not support configuration bursts as a target.

Table 16-3   Configuration Data Phase Control

Condition	Bus Signals	From User Application	
		`C_TERM`	`C_READY`

Wait	`TRDY_IO = 1`
	`DEVSEL_IO = 0`
	`STOP_IO = 1`
		Low	Low

Retry	`TRDY_IO = 1`
	`DEVSEL_IO = 0`
	`STOP_IO = 0`
		High	Low

Disconnect With Data	`TRDY_IO = 0`
	`DEVSEL_IO = 0`
	`STOP_IO = 0`
		High	High

Note: The `C_TERM` and `C_READY` control signals affect termination for all of configuration space, not just user configuration space.

If user configuration space must behave differently from the pre-implemented configuration space in the PCI interface, use the example shown below as a guide.

```
always @(posedge CLK or posedge RST)
begin : cbl_timer
    if (RST) TIMER = 4'h0;
    else if (CFG_VLD) TIMER = BL_WAIT[3:0];
    else if (TIMER != 4'h0) TIMER = TIMER - 4'h1;
end

assign BLAT_RDY = (TIMER <= 4'h4);
assign USER_CFG = ADDR[7] | ADDR[6];
```

```
assign TERMINATE = (!USER_CFG | BLAT_RDY) & S_DATA;

always @(posedge CLK or posedge RST)
begin : keep_it_registered
   if (RST)
   begin
      C_READY = 1'b0;
      C_TERM = 1'b0;
   end
   else
   begin
      C_READY = (CFG_RD | CFG_WR) & TERMINATE;
      C_TERM = (CFG_RD | CFG_WR) & TERMINATE;
   end
end
```

In this example, the PCI interface inserts wait states until it decodes the address of the configuration access. Configuration space accesses are not bandwidth critical, and the method in this example is simple to implement.

Accesses to address 0x40 and above are delayed by the insertion of additional wait states. The number of wait states are specified by the value of the BL_WAIT signal. This example generates transactions like those shown below.

Figure 16-3   Configuration Read with Wait States

In Figure 16-3, "Configuration Read with Wait States", the PCI interface is delaying the completion of a configuration read. This particular transaction is targeting user configuration space.

Figure 16-4   Configuration Write with Wait States

In Figure 16-4, "Configuration Write with Wait States", the PCI inter-
face is delaying the completion of a configuration write. This partic-
ular transaction is also targeting user configuration space.

Capabilities List Pointer

The LogiCORE PCI interface supports a capabilities list through two
options in the configuration file. The capabilities bit in the PCI status
register indicates whether or not the design implements a linked list
of extended capabilities. This bit is set through the "Capabilities List
Enable" option in the configuration file.

If the capabilities bit in the status register is set, the design must
implement the Cap_Ptr register at configuration space address 0x34.
The Cap_Ptr register contains the configuration space address of the
first item in the capabilities list. The capabilities list is a linked list of
registers for implementing various features. Power management
registers are one such feature that are linked into this list.

The PCI interface implements the Cap_Ptr register. The value of this register is set through the "Capabilities List Pointer" option in the configuration file. Values 0x00 through 0x3F are not valid values for the Cap_Ptr register because they point into the standard PCI configuration header.

After the capabilities bit is enabled and the Cap_Ptr is set to an appropriate value, the user application must implement a capabilities list in user configuration space.

Externally Supplied Subsystem Identification

The Subsystem Vendor ID and Subsystem ID fields are used to further differentiate designs which are founded upon the same base design.

As a fictional example, consider a video controller produced by a semiconductor manufacturer. The semiconductor manufacturer sets the Vendor ID and Device ID to identify the video controller as their design, but provides a means for various OEMs to set their own Subsystem Vendor ID. This way, products from different OEMs may be distinguished from each other, even though they use the same graphics chip. The same idea applies to the Subsystem ID, which may be used to distinguish different "trim" levels of a particular product from any single OEM.

For most designs, these values are stored in a configuration ROM table, as specified through the **CFG** bus and the static configuration file. For applications that require a dynamic method of supplying this information, the **SUB_DATA** bus is available. The user should enable the "External Subsystem ID and Subvendor ID" option in the configuration file and drive the proper value onto **SUB_DATA**.

This feature is provided so that a single design (bitstream) may be used in multiple board designs where a different Subsystem Vendor ID or Subsystem ID (or both) are required on each board.

Typical storage implementations for the external value include pullup and pulldown resistors on user I/O pads or an external serial memory. Regardless of the storage method, the value on **SUB_DATA** must be stable and valid before the PCI interface can respond to configuration cycles.

Using pullup and pulldown resistors on user I/O pads to drive **SUB_DATA** provides stable and valid data shortly after power is

applied. In this case, **C_TERM** and **C_READY** may be permanently asserted.

If the user application drives **SUB_DATA** with a value loaded from an external serial memory, the data may not be immediately available. Depending on the design, the user application may have to generate configuration retries to allow time for **SUB_DATA** to become valid.

Initiator Interrupt Acknowledge

Interrupt acknowledge commands issued on the PCI Bus are implicitly addressed to the system interrupt controller. These bus cycles are typically initiated by the host bridge. If the PCI interface and user application are performing the functions of a host bridge, the user application may issue interrupt acknowledge cycles as an initiator.

The example presented in the "Initiator Data Transfer and Control" chapter of this guide is sufficient to issue interrupt acknowledge transactions. The only change required is to alter the COMMAND value to specify the correct bus command:

```
assign COMMAND = 4'b0000;
```

Note that the interrupt acknowledge command uses implied addressing (the system interrupt controller is the implied target). For this reason, the address presented during the address phase of the transaction is not important. However, the AD_IO bus must be driven with stable values. For this reason, do not neglect to drive the **ADIO** bus with valid data during the assertion of **M_ADDR_N**, even if initiating an interrupt acknowledge.

Figure 16-5, "Initiator Issues Interrupt Acknowledge Cycle" shows the PCI interface issuing such a transaction.

PCLK		
AD_IO[31:0]		DEAD8237
CBE_IO[3:0]	Ø	Ø	Ø
PAR_IO		
FRAME_IO		
IRDY_IO		
DEVSEL_IO		
TRDY_IO		
STOP_IO		
REQ_O		
GNT_I		
STATE	RD	RD	RD	RD	RD	RD
ADIO[31:0]		DEAD8237
M_CBE[3:0]	Ø	Ø	Ø	Ø	Ø
REQUEST		
COMPLETE		
M_WRDN		
M_ADDR_N		
M_DATA		
M_DATA_VLD		
M_SRC_EN		
OE		
LOAD		
Q[31:0]	Ø6B97BØD	Ø6B97BØD	Ø6B97BØD	DEAD8237
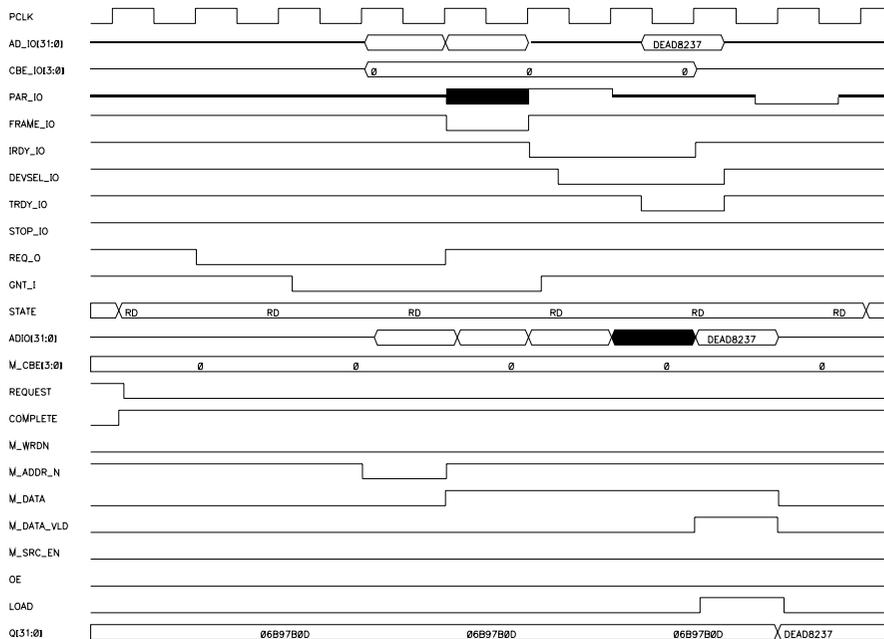
Figure 16-5   Initiator Issues Interrupt Acknowledge Cycle

The interrupt vector returned by the system interrupt controller is stored in a register named `Q`.

Initiator Special Cycle

Special cycle commands issued on the PCI Bus are broadcast cycles to one or more agents on the bus. These bus cycles are typically initiated to relay important system information across the bus. The user application may issue special cycles as an initiator. In order for this to be useful, there must be another agent on the bus capable of monitoring special cycles.

Note: The LogiCORE PCI target cannot monitor special cycles.

The example presented in the "Initiator Data Transfer and Control" chapter of this guide is sufficient to issue special cycle transactions. The only change required is to alter the `COMMAND` value to specify the correct bus command:

```
assign COMMAND = 4'b0001;
```

Special cycles are "addressed" to potentially every agent on the PCI Bus. For this reason, the address presented during the address phase of the transaction is not important. However, the AD_IO bus must be driven with stable values. For this reason, do not neglect to drive the **ADIO** bus with valid data during the assertion of **M_ADDR_N**.

No agent on the bus should respond to a special cycle by asserting DEVSEL_IO. Since no agent responds, the PCI interface must perform a master abort to end the transaction. Furthermore, because the PCI interface knows that it is issuing a special cycle (and is expecting a master abort) it should not set the master abort bit in the status register.

Figure 16-6, "Initiator Issues Special Cycle" shows the PCI interface issuing such a transaction.
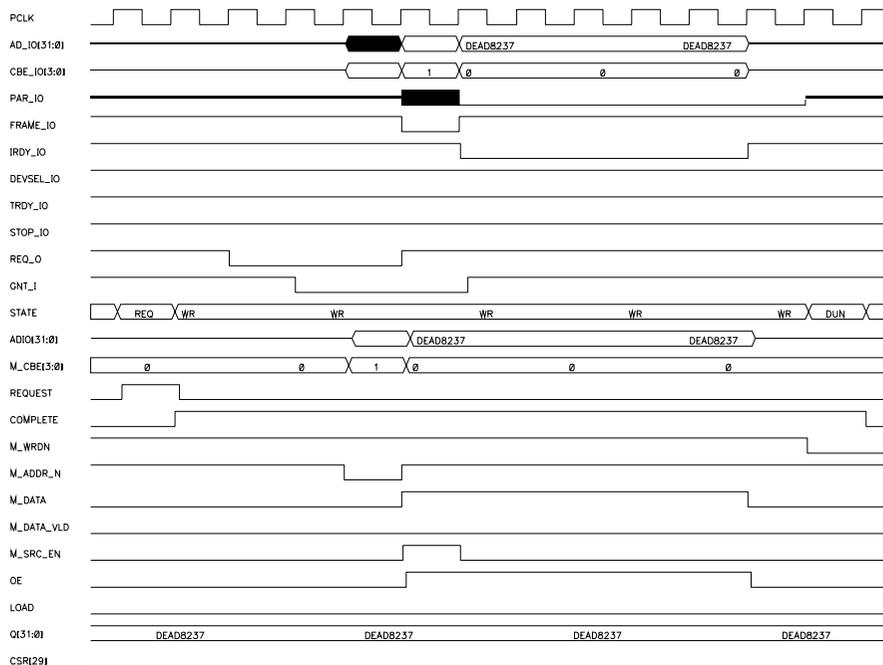
Figure 16-6   Initiator Issues Special Cycle

The message field contained in the register named Q is broadcast on the PCI Bus. Note that the master abort bit, **CSR[29]**, is not set even though the transaction terminates with master abort.

Initiator Configuration Cycles

In a typical PCI Bus system, reading and writing of configuration registers is performed by a host bridge. The host bridge, sometimes called a "north bridge," is the hardware that bridges a host processor bus and the PCI Bus.

The LogiCORE PCI interface is capable of acting as a host bridge. A host bridge is a subset of a central resource as defined in Section 2.4 of the *PCI Local Bus Specification*. A central resource, and a host bridge as part of a central resource, require additional design considerations beyond those for a generic PCI agent. Some of these design considerations are system dependent.

Note: This chapter does not cover the system dependent design considerations of a central resource. It is the responsibility of the designer to address issues such arbitration, interrupt handling, error handling, pullups, and the source of the system reset and clock signals. Designers of PC-AT central resources should carefully review Section 3.7.4 of the *PCI Local Bus Specification*, for additional details on compliance requirements specific to the PC architecture.

There are three items of particular interest when designing a host bridge with the LogiCORE PCI interface. The first is the generation of IDSEL signals, the second is the inclusion of appropriate logic in the user application to allow the PCI interface to configure itself, and the third is the generation of configuration cycles to external agents.

IDSEL Signal Generation

Section 3.7.4. of the *PCI Local Bus Specification* states that the method used to drive the IDSEL signals is left to the discretion of the host bridge designer. However, only a single IDSEL signal may be asserted during the address phase of a configuration transaction. One method mentioned in the specification is to use the upper word of the address bus as the IDSEL signals. Thus, IDSEL of device 0 is connected to AD[16], and so on. This allows for sixteen devices to reside on the PCI Bus.

Due to the extra loading on the AD bus by the IDSEL pins, the IDSEL pin is usually resistively coupled to the AD bus. One exception to this is noted in the electrical specification. If the input pin capacitance of the IDSEL pin of the agent is 8 pF or less, then direct coupling of the AD bus to the IDSEL pin is allowed.

An alternative method is to use separate IDSEL output pins on the host bridge and route them to the various agents on the bus using separate PCB traces.

If a resistor is used, then the signal driving the IDSEL pin may take a long time to reach a valid logic level. Host bridges may solve the timing problem by using address stepping to drive the address, but not assert FRAME#, for one or more clock cycles. This allows the IDSEL pin to reach a valid logic level. Although certain implementations of the LogiCORE PCI interface use address stepping, the PCI interface does not support this technique.

The designer should either use direct coupling or generate separate IDSEL signals. For embedded applications, direct coupling of IDSEL is recommended. The slight increase in bus propagation time may be acceptable in many closed systems. It is the responsibility of the designer to ensure that timing requirements are not violated.

Initiator Self Configuration Cycles

Since the initiator and target state machines in the LogiCORE PCI interface are independent, it is possible to use the initiator state machine to generate a configuration transaction aimed at the target state machine.

The steps for generating self configuration cycles are very similar to the steps for generating single data phase initiator transactions. The initiator control signals are driven the same as in any other initiator transaction, and the user application presents a configuration read or configuration write command on **M_CBE**.

The IDSEL signal routed to IDSEL_I on the LogiCORE PCI interface must be asserted during the address phase of the transaction. This is easily accomplished by using the direct coupling method for IDSEL generation described in the previous section. When the user application drives an address onto the **ADIO** bus during **M_ADDR_N** assertion, the bit pattern in the upper word of the address should result in IDSEL_I assertion for the interface.
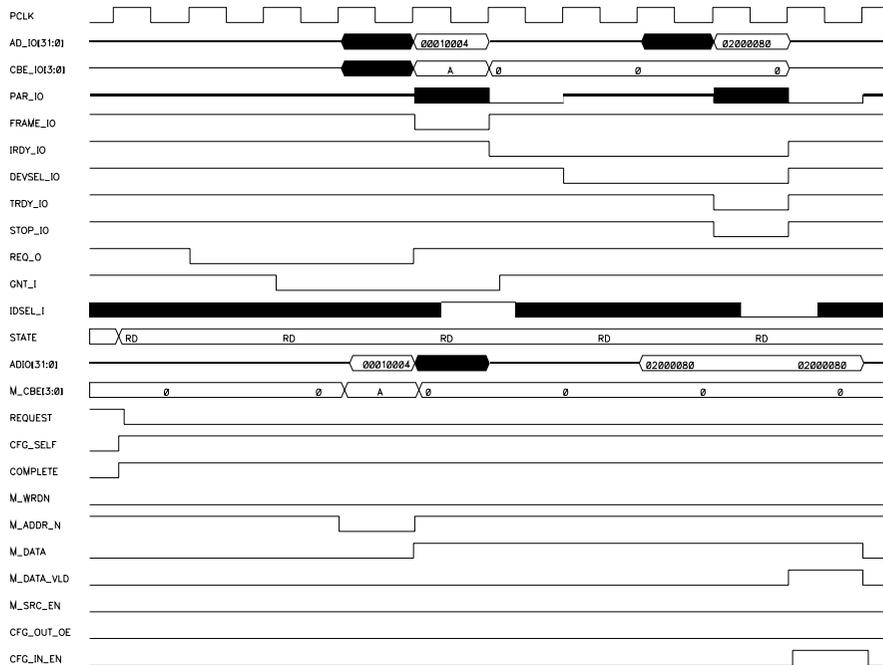
Figure 16-7 Self Configuration Read

A self configuration cycle must be signalled by the user application no later than one cycle after **REQUEST** is asserted. This is done by asserting **CFG_SELF**. This signal must remain asserted until the transaction is complete, which is signalled by the deassertion of **M_DATA**. Figure 16-7, "Self Configuration Read" shows the correct behavior and demonstrates the PCI interface reading its own command and status register.

Internally, the interface uses the **CFG_SELF** signal to temporarily force the value of the bus master enable bit of the command register. This allows the interface to generate configuration transactions as a bus master even when the bus master enable bit is cleared after a system reset.

The **CFG_SELF** signal also affects the internal data path, indicating to the PCI interface that data will be transferred between the user application and the interface over the **ADIO** bus. For this reason,

CFG_SELF must always be asserted during self configuration trans-actions, even after the bus master enable bit has been set.

During self configuration reads, the AD_IO bus is driven with invalid data during the data phase of the transfer. The valid data is available internally on the **ADIO** bus and is available to the user application. This is generally not a concern, as the LogiCORE PCI interface is both the initiator and the target.

No other agents will respond to the configuration transaction because no other agent will sample its IDSEL asserted. Any agent attempting to snoop the transaction (such as a bus analyzer) will not obtain meaningful data.
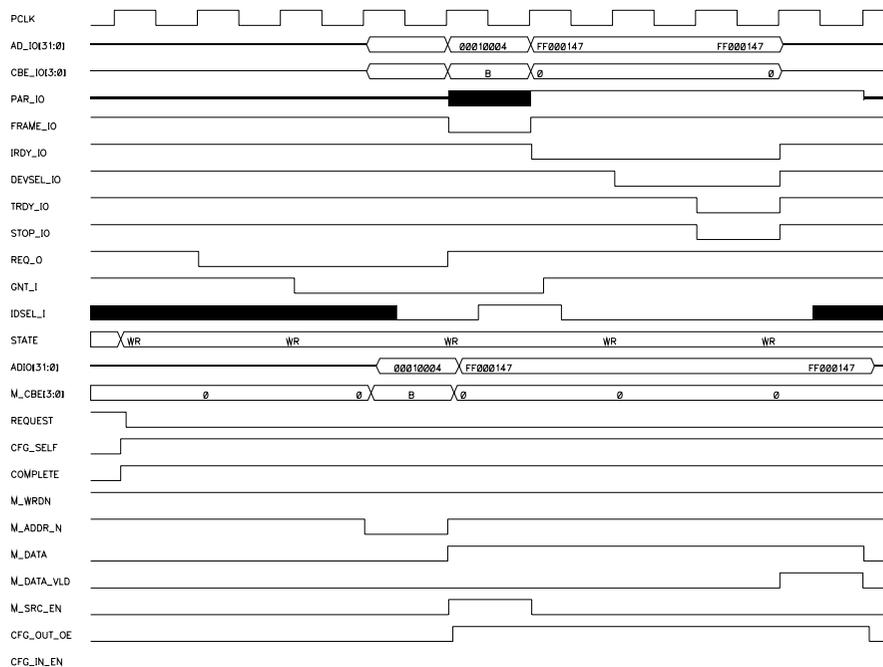
Figure 16-8   Self Configuration Write

Figure 16-8, "Self Configuration Write" also demonstrates the correct behavior. In this instance, the PCI interface is enabling itself by writing to its command and status register. Notice that in both figures, IDSEL_I is sampled during the address phase; at other times it is ignored.

Generation of Configuration Cycles to Other Agents

Once the host bridge is configured and enabled using self configuration cycles, it can be used to configure other agents on the PCI Bus.

The steps for generating configuration cycles are identical to the steps for generating other initiator transactions. The initiator control signals are driven the same as in any other initiator transaction, and the user application presents a configuration read or configuration write command on **M_CBE**.

The IDSEL signal for the configuration target must be asserted during the address phase of the transaction. Again, this may be accomplished by using the direct coupling method for IDSEL generation. When the user application drives an address onto the **ADIO** bus during **M_ADDR_N** assertion, the bit pattern in the upper word of the address should result in IDSEL assertion for the configuration target.

When configuring external agents, the signal **CFG_SELF** must not be asserted. Asserting this signal will result in a corrupt data transfer.

Other Considerations

While the *PCI Local Bus Specification* permits burst configuration transactions and the LogiCORE PCI interface is capable of issuing such transactions, it must respond to them with disconnect with data. Internally, the PCI interface does not keep an address pointer for configuration space addresses. To ensure this, tie the **C_TERM** and **C_READY** signals to logic high.

Do not attempt self configuration burst transactions. Additionally, it is recommended that burst cycles not be used for any configuration transactions. The use of burst configuration will greatly complicate the design of the user application.

Configuration transactions terminated with retry must be reissued, as is the case with all other transactions terminated with retry. The host bridge designer needs to determine the best order to perform retries.

One method, since configuration sequences are not usually time critical, is to retry the current transaction until it is completed or has timed out. Another method is to push this responsibility on to the configuration software by reporting retries to the host, and have the configuration software accept responsibility for reissuing the transaction at a later time.

During the polling sequence for external agents, it is fully expected that some master abort terminations will occur when polling empty PCI slots. The host bridge application needs to be able to report this condition to the host. Master abort terminations, as well as target abort terminations, should not disable the host bridge from performing further configuration cycles.

Error Detection and Reporting

The LogiCORE PCI interface generates parity, checks parity, and reports errors as required by the *PCI Local Bus Specification.* For more information regarding these specifications, refer to Section 3.8 of the *PCI Local Bus Specification.*

These functions are done in a method that is transparent to the user application. Certain classes of user applications, however, may need to know if an error has occurred. The purpose of this chapter is to demonstrate the behavior of the PCI interface when it encounters parity errors.

Consider the target write shown in the waveform of Figure 17-1, "Target Write with no Parity Errors". This waveform shows a target write which is not disrupted by parity errors. The subsequent sections discuss the response of the PCI interface when parity errors are introduced during the address phase and during data phases.
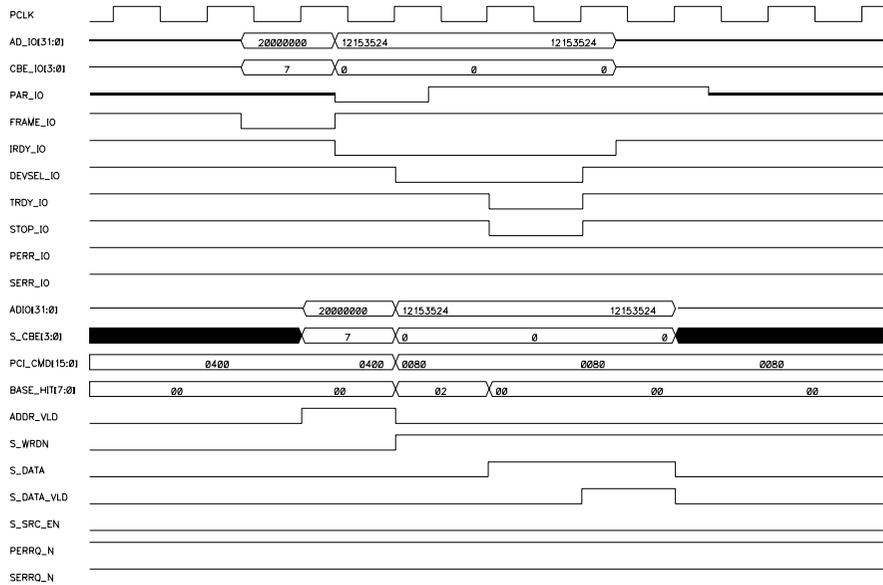
Figure 17-1   Target Write with no Parity Errors

The figure shows the error reporting signals on the PCI Bus (PERR_IO and SERR_IO) and the registered copies of these signals provided to the user application (**PERRQ_N** and **SERRQ_N**).

Address Parity Errors

The LogiCORE PCI interface checks for parity errors during address phases on the PCI Bus. As required by the *PCI Local Bus Specification*, the PCI interface reports address parity errors via SERR_IO. A registered version of this signal is available to the user application as **SERRQ_N**.

If an address parity error is detected, the PCI interface will either claim the transaction and issue a target abort, or will not claim the transaction at all. Figure 17-2, "Target Write with Address Parity Error" shows the same transaction presented in Figure 17-1, "Target Write with no Parity Errors" with an address parity error. Since the PCI interface does not respond to the transaction, the initiator terminates with master abort.

PCLK	
AD_IO[31:0]	20000000 12153524 12153524
CBE_IO[3:0]	7 0 0 0
PAR_IO	
FRAME_IO	
IRDY_IO	
DEVSEL_IO	
TRDY_IO	
STOP_IO	
PERR_IO	
SERR_IO	
ADIO[31:0]	20000000 12153524 12153524
S_CBE[3:0]	7 0 0 0
PCI_CMD[15:0]	0080 0080 0080 0080
BASE_HIT[7:0]	00 00 00 00 00
ADDR_VLD	
S_WRDN	
S_DATA	
S_DATA_VLD	
S_SRC_EN	
PERRQ_N	
SERRQ_N	
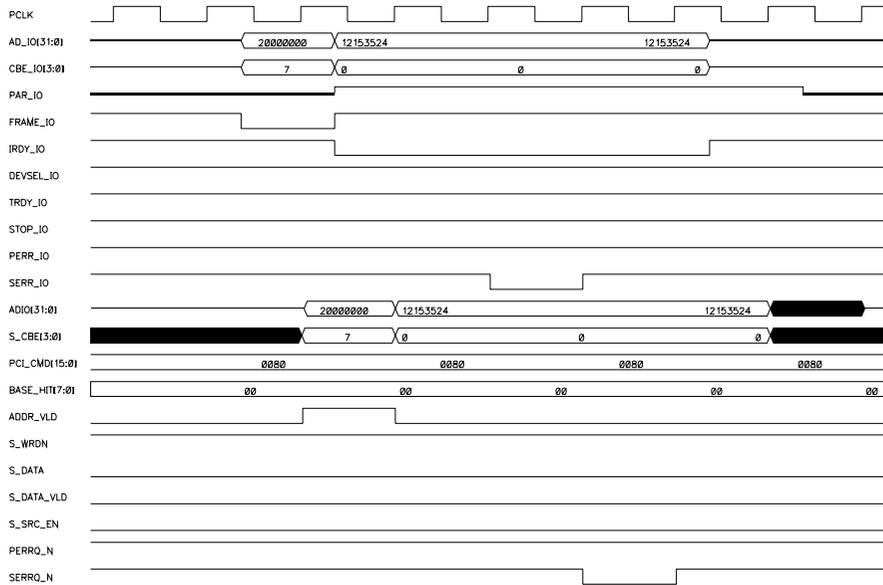
Figure 17-2   Target Write with Address Parity Error

The user application may monitor for system errors on the PCI Bus by sampling **SERRQ_N**. This signal indicates an address parity error or other serious system error. In addition, the command and status register state is available through the **CSR[39:0]** bus.

The **SERRQ_N** signal passed to the user application is a registered version of the system error signal, SERR_IO. This signal is for reporting catastrophic system errors. Typically, the assertion of this signal will result in a non-maskable interrupt being issued to the host. If the user application is a host bridge design, it should monitor the **SERRQ_N** signal and act accordingly.

Note: The SERR_IO signal is an open drain signal. It is passively deasserted by a pullup after synchronous assertion by a bus agent. This transition may take more than one clock cycle, which creates the possibility of a metastable output on **SERRQ_N**. The logic in the user application that generates a non-maskable interrupt should be designed with this in mind. Use of a synchronizer is recommended.

Data Parity Errors

The LogiCORE PCI interface checks for parity errors during data phases on the PCI Bus when it is receiving data. As required by the *PCI Local Bus Specification*, the PCI interface reports data parity errors via PERR_IO. A registered version of this signal is available to the user application as **PERRQ_N**.

If a data parity error is detected, the PCI interface will take no additional action other than signalling the error. However, if the user application may take any steps it deems necessary, including transfer termination. Figure 17-3, "Target Write with Data Parity Error" shows the same transaction presented in Figure 17-1, "Target Write with no Parity Errors" with a data parity error.
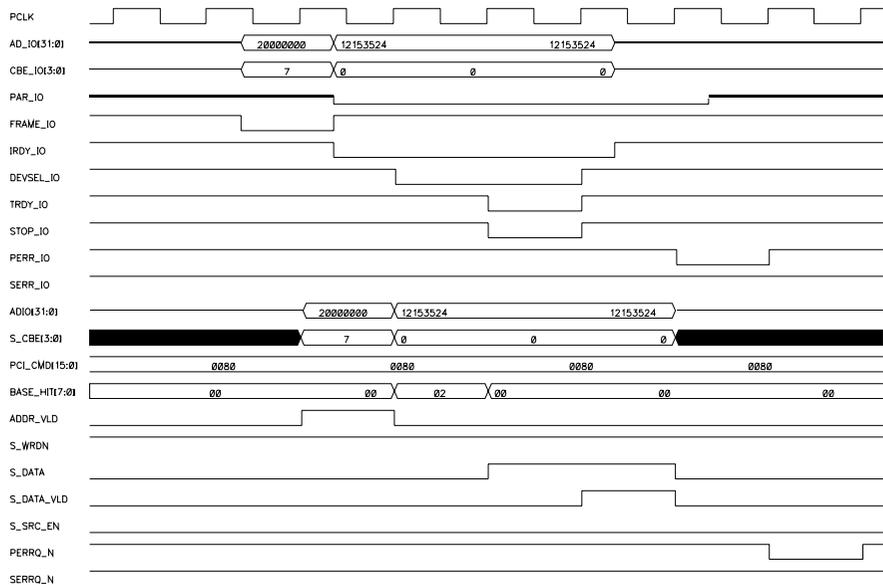
Figure 17-3   Target Write with Data Parity Error

The user application may monitor for parity errors on the PCI Bus by sampling **PERRQ_N**. This signal indicates an address parity error or other serious system error. In addition, the command and status register state is available through the **CSR[39:0]** bus.

Chapter 18

Design Constraints

To ensure that the PCI timing requirements are met when the user design is added, the PCI LogiCORE interface uses timing constraints during processing. The constraints file, used in combination with the appropriate guide file, guarantees the required PCI performance.

This chapter is intended to discuss the following:

- The function of the constraints in the user constraints file (`*.ucf`)

- The generation of additional constraints for the user application

- The use of the physical constraints file (`*.pcf`)

- The function of the guide file (`*.ncd`)

Note: Do not modify or remove the constraints found in the user constraints file. Customization of this file is allowed, but the modifications must not change the functionality of the supplied constraints. Make absolutely no modifications to the guide file. Xilinx guarantees PCI compliant timing only if the constraints file and guide file are used during processing.

Supplied User Constraints

While there are separate constraints files for every part and package combination, the constraints in each perform the same function. These functions are:

- Pinout definition constraints

- Absolute placement constraints

- Timing constraints

Pinout Definition Constraints

The pinout of the PCI interface is constrained in the user constraints file to match the suggested pin ordering in the *PCI Local Bus Specification* as closely as possible. In addition, the user constraints file prohibits some device pins from being used to reserve them for configuration and boundary scan.

Absolute Placement Constraints

A large number of absolute placement constraints are specified in the user constraints file. These are intended to group critical logic sections together. Additionally, the placement constraints align internal datapath registers and three-state buffers in columns which straddle the internal **ADIO** bus. The **ADIO** bus is formed by horizontal long lines in the FPGA.

Timing Constraints

Timing constraints are specified in three distinct steps. First, time-name (TNM) attributes are attached to specific instances in the design. Next, timegroups (TIMEGRP) are created from timenames, if necessary. Finally, timespecs (TIMESPEC) are formed by timing specifications between various timenames and timegroups.

Each device and package combination has unique timing constraints to ensure full PCI compliance. Do not remove any constraints.

Additional User Constraints

Depending on the nature of the customized user design, it may be necessary to modify or add constraints in the user constraints file.

- Change the instance name associated with USER_FFS to match the instance name of the user application.

- If the user application has input and output pads of its own, add a TNM for USER_PADS and specify two additional TIMESPECs for ALL_FFS to USER_PADS and USER_PADS to ALL_FFS.

  Pinout constraints may be added as long as they do not conflict with the pinout established for the PCI interface.

- If the default PERIOD constraint does not adequately constrain the user application, add TNMs, TIMEGRPs, and TIMESPECs as needed to sufficiently constrain the design.

  The PERIOD constraint has the lowest priority, so additional TIMESPECs will "carve" paths out of this default constraint.

- In complex designs, or designs involving multiple clock domains, it may be necessary to entirely remove the PERIOD constraint to effectively process the design.

  In this case, it is important to add at least three additional TIMESPECs to constrain paths within the PCI interface and those paths to and from the portion of the user application that communicates with the PCI interface. These TIMESPECs should have the same value as the PERIOD constraint:

  ```
  TS_UA2PCI = FROM : USER_FFS : TO : PCI_FFS
  ```

  ```
  TS_PCI2UA = FROM : PCI_FFS : TO : USER_FFS
  ```

  ```
  TS_PCI2PCI = FROM : PCI_FFS : TO : PCI_FFS
  ```

Note: If multiple clock domains are used, the USER_FFS flip-flops should be limited to those that connect to the PCI interface in the PCI clock domain.

Physical Constraints

The mapper generates a physical constraints file for use by the place and route tool. This physical constraints file contains translated constraints information from the user constraints file, as well as any constraints embedded in the design itself.

Note that while the syntax of a physical constraints file looks similar to that of a user constraints file, it can be significantly different. Additions to the physical constraints file should be performed by advanced users only.

In all designs, data transfer between the PCI interface and the user application occurs over the **ADIO** bus. As this bus has multiple data sources and data sinks, there are a great number of potential paths for timing analysis. Typically, registers in the user application do not transfer data between themselves. In large designs, it is often desirable to eliminate false paths which trace through the **ADIO** bus by the use of timing ignores (TIG) in the physical or user constraints files.

Guide Files

Guide files, when provided, contain routing and placement information for a few highly critical sections of the PCI interface. For this reason, guide files must not be modified.

The place and route tool has two guide modes. These modes are called leveraged and exact. The guide file for use with the PCI interface must be used with the exact guide mode.

Note: In order for PAR to identify exact matches between the customized user design and the guide file, the user design must be placed in the supplied "wrapper" file. This forces net names, instance names, and structure (hierarchy) in the customized user design to match those in the guide file.

Index

Numerics

A

B

C