Table 2-2. Major PCI Revision 2.1 Features

Feature	Description
Processor Independence	Components designed for the PCI bus are PCI-specific, not processor-specific, thereby isolating device design from processor upgrade treadmill.
Support for up to 256 PCI functional devices per PCI bus	Although a typical PCI bus implementation supports approximately ten electrical loads, each PCI device package may contain up to eight separate PCI functions. The PCI bus logically supports up to 32 physical PCI device packages, for a total of 256 possible PCI functions per PCI bus.
Support for up to 256 PCI buses	The specification provides support for up to 256 PCI buses.
Low-power consumption	A major design goal of the PCI specification is the creation of a system design that draws as little current as possible.
Burst used for all read and write transfers	Supports 132Mbytes per second peak transfer rate for both read and write transfers. 264Mbytes per second peak transfer rate for 64-bit PCI transfers. Transfer rates of up to 524Mbytes per second are achievable on a 66MHz PCI bus.
Bus speed	Revision 2.0 spec supports PCI bus speeds up to 33MHz. Revision 2.1 adds support for 66MHz bus operation.
64-bit bus width	Full definition of a 64-bit extension.
Fast access	As fast as 60ns (at a bus speed of 33MHz when an initiator parked on the PCI bus is writing to a PCI target.
Concurrent bus operation	High-end bridges support full bus concurrency with host bus, PCI bus, and the expansion bus simultaneously in use.
Bus master support	Full support of PCI bus initiators allows peer-to-peer PCI bus access, as well as access to main memory and expansion bus devices through PCI and expansion bus bridges. In addition, a PCI master can access a target that resides on another PCI bus lower in the bus hierarchy.
Hidden bus arbitration	Arbitration for the PCI bus can take place while another bus master is in possession of the PCI bus. This eliminates latency encountered during bus arbitration on buses other than PCI.
Low-pin count	Economical use of bus signals allows implementation of a functional PCI target with 47 pins and a functional PCI bus initiator with 49 pins.
Transaction integrity check	Parity checking on the address, command and data.
Three address spaces	Full definition of memory, I/O and configuration address space.
Auto-Configuration	Full bit-level specification of the configuration registers necessary to support automatic peripheral detection and configuration.
Software Transparency	Software drivers utilize same command set and status definition when communicating with PCI device or its expansion bus-oriented cousin.
Expansion Cards	The specification includes a definition of PCI connectors and add-in cards.
Expansion Card Size	The specification defines three card sizes: long, short and variable-height short cards.

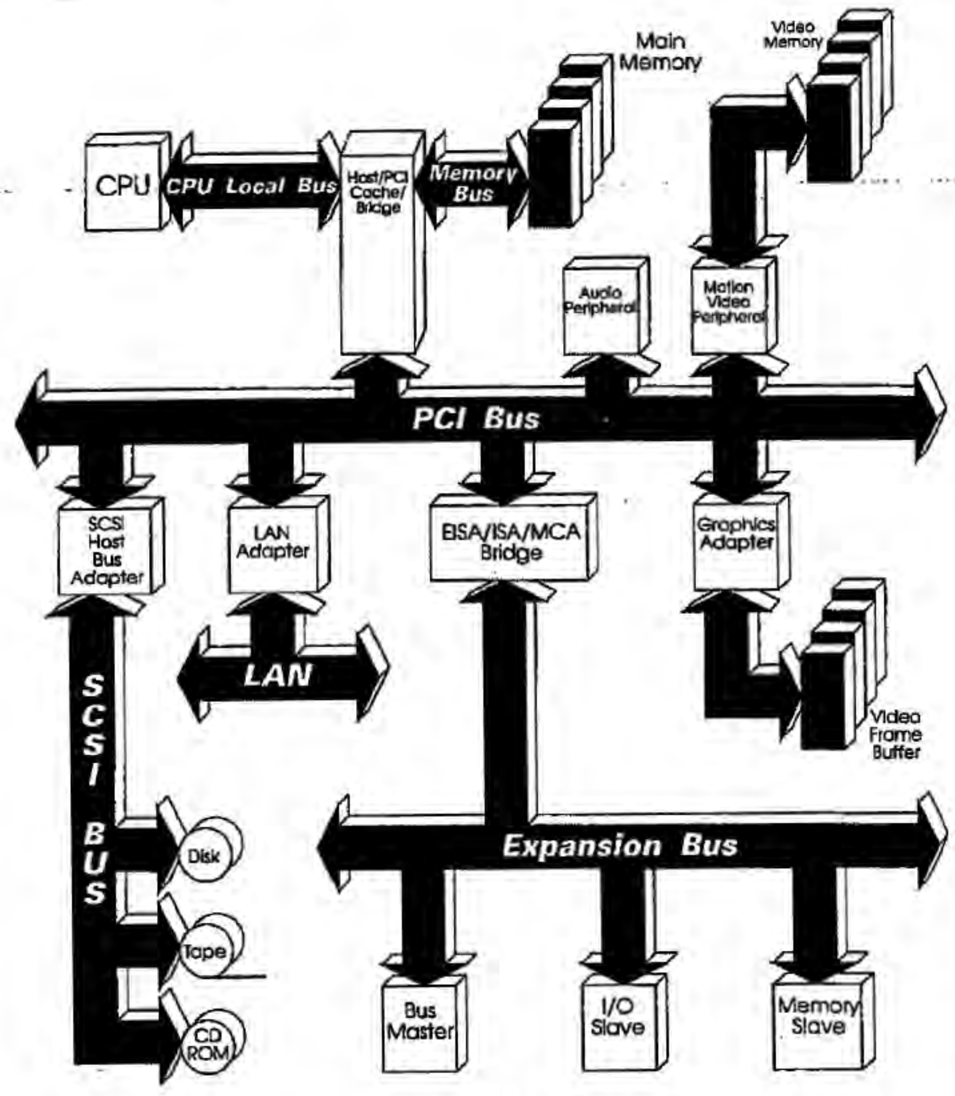

Figure 2-4. The PCI Bus

Market Niche for PCI and VESA VL

Many in the industry are using their crystal balls to predict the outcome of this "bus war," but this will not be a win/lose situation. VL is a good, cost-effective approach for low-end machines that require fast data transfer capability with one subsystem at a time in order to achieve acceptable system performance. Due to the complexity of the PCI chip sets when compared to the logic required by VL 1.0, PCI-based systems are slightly more expensive. Balancing this added cost with PCI's superior performance in supporting bus concurrency, auto-configuration and multiple bus masters, PCI-based machines will dominate the mid and high-end machine market niches.

It should be noted, however, that a machine can be designed without any bridges. All components, including the processor and main memory, would interface directly to the PCI bus. Due to the reduction in logic yielded by the deletion of the bridge logic, this PCI machine would be very price-competitive with a VESA VL-based machine.

PCI Device

The typical PCI device consists of a complete peripheral adapter encapsulated within an IC package or integrated onto a PCI expansion card. Typical examples would be a network, display or SCSI adapter. During the initial period after the introduction of the PCI specification, many vendors chose to interface pre-existent, non-PCI compliant devices to the PCI bus. This can be easily accomplished using programmable logic arrays (PLAs). Figure 2-5 illustrates ten PCI-compliant devices attached to the PCI bus on the system board. It should also be noted that each PCI-compliant package (VLSI component or add-in card) may contain up to eight PCI functions.

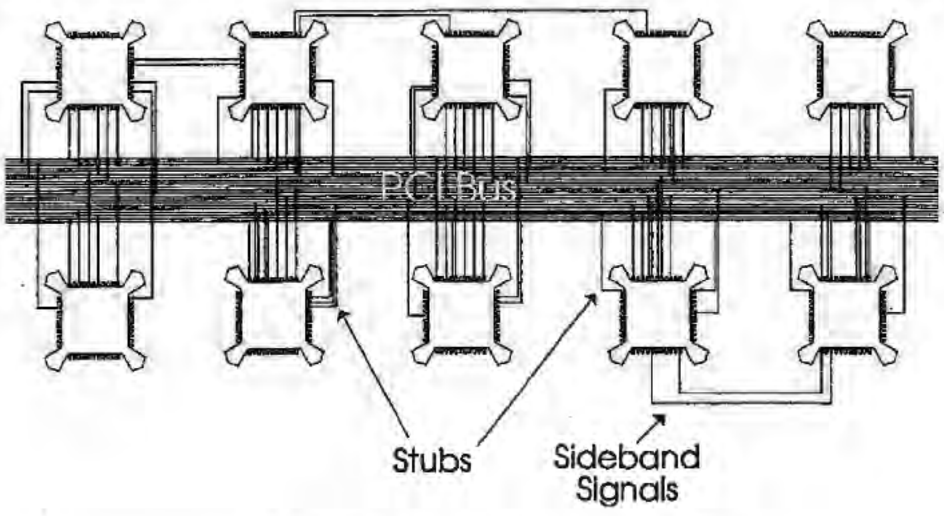

Figure 2-5. PCI Devices Attached to the PCI Bus

Specifications Book is Based On

This book is based on the documents indicated in table 2-3.

Table 2-3. This Book is Based On

Document Title	Revision
PCI Local Bus Specification	2.1
PCI-to-PCI Bridge Specification	1.0
PCI System Design Guide	1.0
PCI BIOS Specification	2.1

Obtaining PCI Bus Specification(s)

The PCI bus specification, version 1.0, was developed by Intel Corporation. The specification is now managed by a consortium of industry partners known as the PCI Special Interest Group (SIG). MindShare, Inc. is a member of the SIG. The specifications are commercially available for purchase. The latest revision of the specification (as of this printing) is 2.1. For information regarding the specifications and/or SIG membership, contact:

PCI Special Interest Group
P.O. Box 14070
Portland, OR 97214
Tel. (503) 797-4207 (International)
Fax (503) 234-6762
(800) 433-5177 (in U.S.)

Part II

Revision 2.1 Essentials

Chapter 3

The Previous Chapter

The previous chapter introduced the local bus concept, the VESA VL bus and the PCI bus.

In This Chapter

This chapter provides an introduction to the PCI transfer mechanism, including a definition of the following basic concepts: burst transfers, the initiator, targets, agents, single and multi-function devices, the PCI bus clock, the address phase, claiming the transaction, the data phase, transaction completion and the return of the bus to the idle state.

The Next Chapter

Unlike most buses, the PCI bus does not incorporate termination resistors at the physical end of the bus to absorb voltage changes and prevent the wavefront caused by the voltage change from being reflected back down the bus. Rather, PCI uses reflections to advantage. The next chapter provides an introduction to reflected-wave switching.

Burst Transfer

A burst transfer is one consisting of a single address phase followed by two or more data phases. The bus master only has to arbitrate for bus ownership one time. The start address and transaction type are issued during the address phase. The target device latches the start address into an address counter and is responsible for incrementing the address from data phase to data phase.

In the 486, EISA and Micro Channel environments, the ability to perform burst transfers is the product of negotiation between the bus master and the target device. If either or both of them do not support burst mode transfers, the data packet can only be transferred utilizing a series of separate bus transactions. The bus master must arbitrate for ownership of the bus to perform each individual transaction that comprise the series. Another bus master may acquire bus ownership after the master completes any transaction in the series. This can severely impact the bus master's data throughput.

Most PCI data transfers are accomplished using burst transfers. Most PCI bus masters and target devices are designed to support burst mode. It should be noted that a PCI target may be designed such that it can only handle single data phase transactions. When a bus master attempts to perform a burst transaction, the target terminates the transaction at the completion of the first data phase. This forces the master to re-arbitrate for the bus to attempt resumption of the burst with the next data item. The target terminates each burst transfer after the first data phase completes. This would yield very poor performance, but may be the correct approach for a device that doesn't require high throughput. Each burst transfer consists of the following basic components:

- The address and transfer type are output during the address phase.
- A data object (up to 32-bits in a 32-bit implementation or 64-bits in a 64-bit implementation) may then be transferred during each subsequent data phase.

Assuming that neither the initiator nor the target device inserts wait states in each data phase, a data object may be transferred on the rising-edge of each PCI clock cycle. At a PCI bus clock frequency of 33MHz, a transfer rate of 132Mbytes/second may be achieved. A transfer rate of 264Mbytes/second may be achieved in a 64-bit implementation when performing 64-bit transfers during each data phase. A 66MHz PCI bus implementation can achieve 264 or 524Mbytes/second transfer rates using 32 or 64-bit transfers. This chapter introduces the burst mechanism used in performing transfers over the PCI bus.

Initiator, Target and Agents

There are two participants in every PCI burst transfer: the initiator and the target. The initiator, or bus master, is the device that initiates a transfer. The terms bus master and initiator can be used interchangeably, but the PCI specification strictly adheres to the term initiator.

The target, or slave, is the device currently addressed by the initiator for the purpose of performing a data transfer. The terms target and slave can be used interchangeably, but the PCI specification strictly adheres to the term target.

All PCI initiator and target devices are commonly referred to as PCI-compliant agents.

Single vs. Multi-Function PCI Devices

A PCI physical device package may take the form a component integrated onto the system board or the form of a PCI add-in card. Each PCI package may incorporate from one to eight separate functions. This is analogous to a multi-function card found in any ISA, EISA or Micro Channel machine. A package containing one function is referred to as a single-function PCI device, while a package containing two or more PCI functions is referred to as a multi-function device.

PCI Bus Clock

All actions on the PCI bus are synchronized to the PCI CLK signal. The frequency of the CLK signal may be anywhere from 0MHz to 33MHz. The revision 1.0 specification stated that all devices must support operation from 16 to 33MHz, while recommending support for operation down to 0MHz. The revision 2.x PCI specification indicates that **ALL** PCI devices **MUST** support PCI operation within the 0MHz to 33MHz range. Support for operation down to 0MHz provides low-power and static debug capability. The PCI CLK frequency may be changed at any time and may be stopped (but only in the low state). Components integrated onto the system board may operate at a single frequency and may require a policy of no frequency change. Devices on add-in cards must support operation from 0 through 33MHz (because the card must operate in any platform that it may be installed in).

The revision 2.1 specification also defines PCI bus operation at speeds of up to 66MHz. The chapter entitled "66MHz PCI Implementation" describes the operational characteristics of the 66MHz PCI bus, embedded devices and add-in cards.

All PCI bus transactions consist of an address phase followed by one or more data phases. The exception is a transaction wherein the initiator uses 64-bit addressing delivered in two address phases. An address phase is one PCI

CLK in duration. The number of data phases depends on how many data transfers are to take place during the overall burst transfer. Each data phase has a minimum duration of one PCI CLK. Each wait state inserted in a data phase extends it by an additional PCI CLK.

Address Phase

As stated earlier, every PCI transaction (with the exception of a transaction using 64-bit addressing) starts off with an address phase one PCI CLK period in duration. During the address phase, the initiator identifies the target device and the type of transaction. The target device is identified by driving a start address within its assigned range onto the PCI address/data bus. At the same time, the initiator identifies the type of transaction by driving the command type onto the PCI Command/Byte Enable bus. The initiator asserts the FRAME# signal to indicate the presence of a valid start address and transaction type on the bus. Since the initiator only presents the start address for one PCI clock cycle, it is the responsibility of every PCI target device to latch the address so that it may subsequently be decoded.

By decoding the address latched from the address bus and the command type latched from the Command/Byte Enable bus, a target device can determine if it is being addressed and the type of transaction in progress. It's important to note that the initiator only supplies a start address to the target (during the address phase). Upon completion of the address phase, the address/data bus is then used to transfer data in each of the data phases. It is the responsibility of the target to latch the start address and to auto-increment it to point to the next group of locations during each subsequent data transfer.

Claiming the Transaction

When a PCI target determines that it is the target of a transaction, it must claim the transaction by asserting DEVSEL# (device select). If the initiator doesn't sample DEVSEL# asserted within a predetermined amount of time, it aborts the transaction.

Data Phase(s)

The data phase of a transaction is the period during which a data object is transferred between the initiator and the target. The number of data bytes to be transferred during a data phase is determined by the number of Com mand/Byte Enable signals that are asserted by the initiator during the data phase.

Both the initiator and the target must indicate that they are ready to complete a data phase, or the data phase is extended by a wait state one PCI CLK period in duration. The PCI bus defines ready signal lines used by both the initiator (IRDY#) and the target (TRDY#) for this purpose.

Transaction Duration

The initiator identifies the overall duration of a burst transfer with the FRAME# signal. FRAME# is asserted at the start of the address phase and remains asserted until the initiator is ready (asserts IRDY#) to complete the final data phase.

Transaction Completion and Return of Bus to Idle State

The initiator indicates that the last data transfer (of a burst transfer) is in progress by deasserting FRAME# and asserting IRDY#. When the last data transfer has been completed, the initiator returns the PCI bus to the idle state by deasserting its ready line (IRDY#).

If another bus master had previously been granted ownership of the bus by the PCI bus arbiter and were waiting for the current initiator to surrender the bus, it can detect that the bus has returned to the idle state by detecting FRAME# and IRDY# both deasserted.

"Green" Machine

In keeping with the goal of low power consumption, the specification calls for low-power, CMOS output drivers and receivers to be used by PCI devices.

The next chapter describes the reflected-wave switching used in the PCI bus environment to permit low-power, CMOS drivers to successfully drive the bus.

If the address/data bus signals attached to the CMOS input receivers are permitted to float (around the switching region of input buffers) for extended periods of time, the receiver inputs would oscillate and draw excessive current. To prevent this from happening, it is a rule in PCI that the address/data bus must not be permitted to float for extended periods of time. Since the bus is normally driven most of the time, it may be assumed that the pre-charged bus will retain its state while not being driven for brief periods of time during turnaround cycles (turnaround cycles are described in the chapter entitled "The Read and Write Transfer."

The section entitled "Bus Parking" in the chapter on bus arbitration describes the mechanism utilized to prevent the address/data bus from floating when the bus is idle. The chapter entitled "The Read and Write Transfer" describes the mechanism utilized during data phases with wait states. The chapter entitled "The 64-Bit Extension" describes the mechanism utilized to keep the upper 32 bits of the address/data bus from floating when they are not in use (during a 32-bit transfer).

Chapter 5

The Previous Chapter

The previous chapter provided an introduction to reflected-wave switching.

This Chapter

This chapter divides the PCI bus signals into functional groups and describes the function of each signal.

The Next Chapter

When a PCI bus master requires the use of the PCI bus to perform a data transfer, it must request the use of the bus from the PCI bus arbiter. The next chapter provides a detailed discussion of the PCI bus arbitration timing. The PCI specification defines the timing of the request and grant handshaking, but not the procedure used to determine the winner of a competition. The algorithm used by a system's PCI bus arbiter to decide which of the requesting bus masters will be granted use of the PCI bus is system-specific and outside the scope of the specification.

Introduction

This chapter introduces the signals utilized to interface a PCI-compliant device to the PCI bus. Figures 5-1 and 5-2 illustrate the required and optional signals for master and target PCI devices, respectively. A PCI device that can act as the initiator or target of a transaction would obviously have to incorporate both initiator and target-related signals. In actuality, there is no such thing as a device that is purely a bus master and never a target. At a minimum, a device must act as the target of configuration reads and writes.

Each of the signal groupings are described in the following sections. It should be noted that some of the optional signals are not optional for certain types of

PCI agents. The sections that follow identify the circumstances where signals must be implemented.

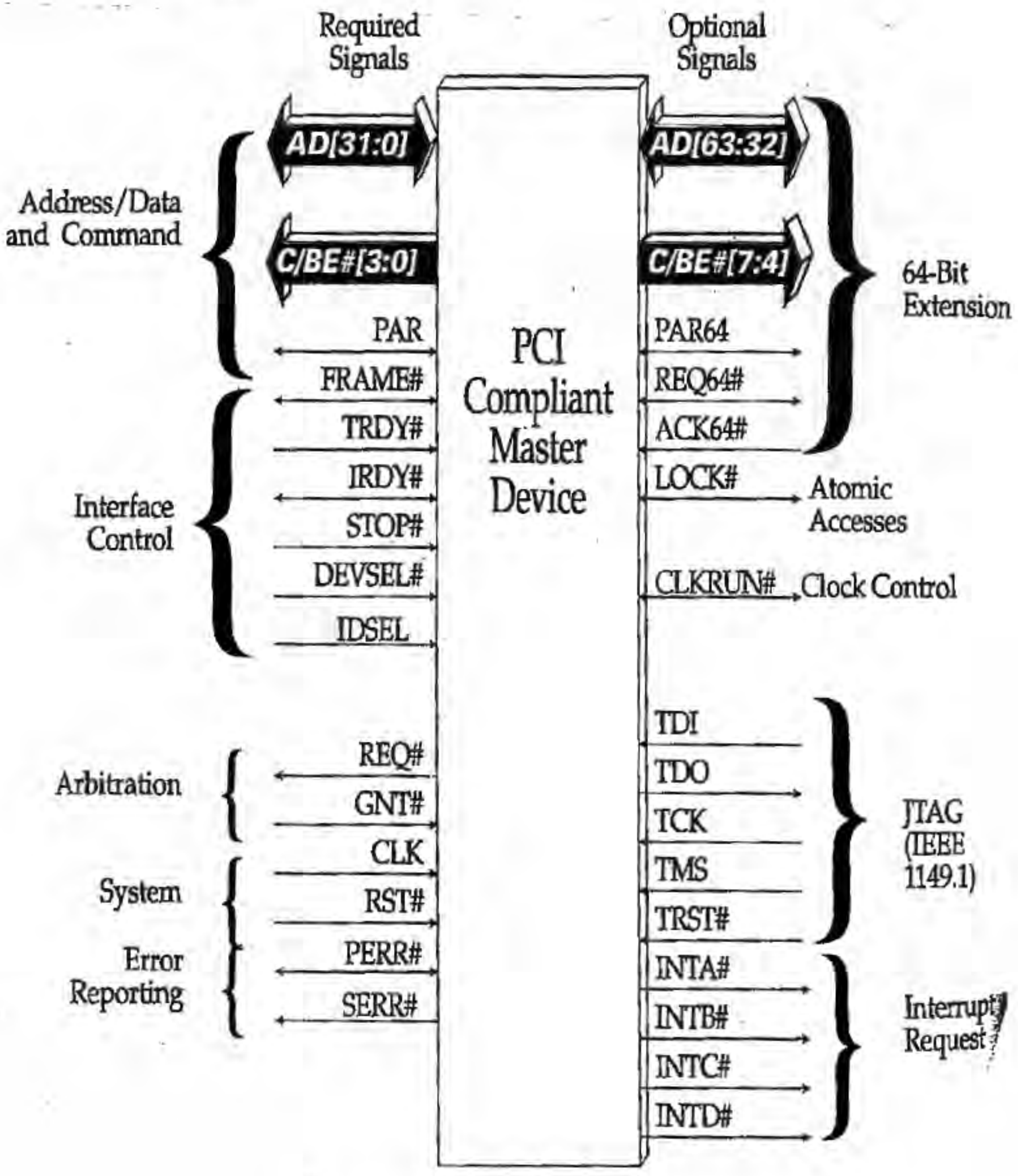

Figure 5-1. PCI-Compliant Master Device Signals

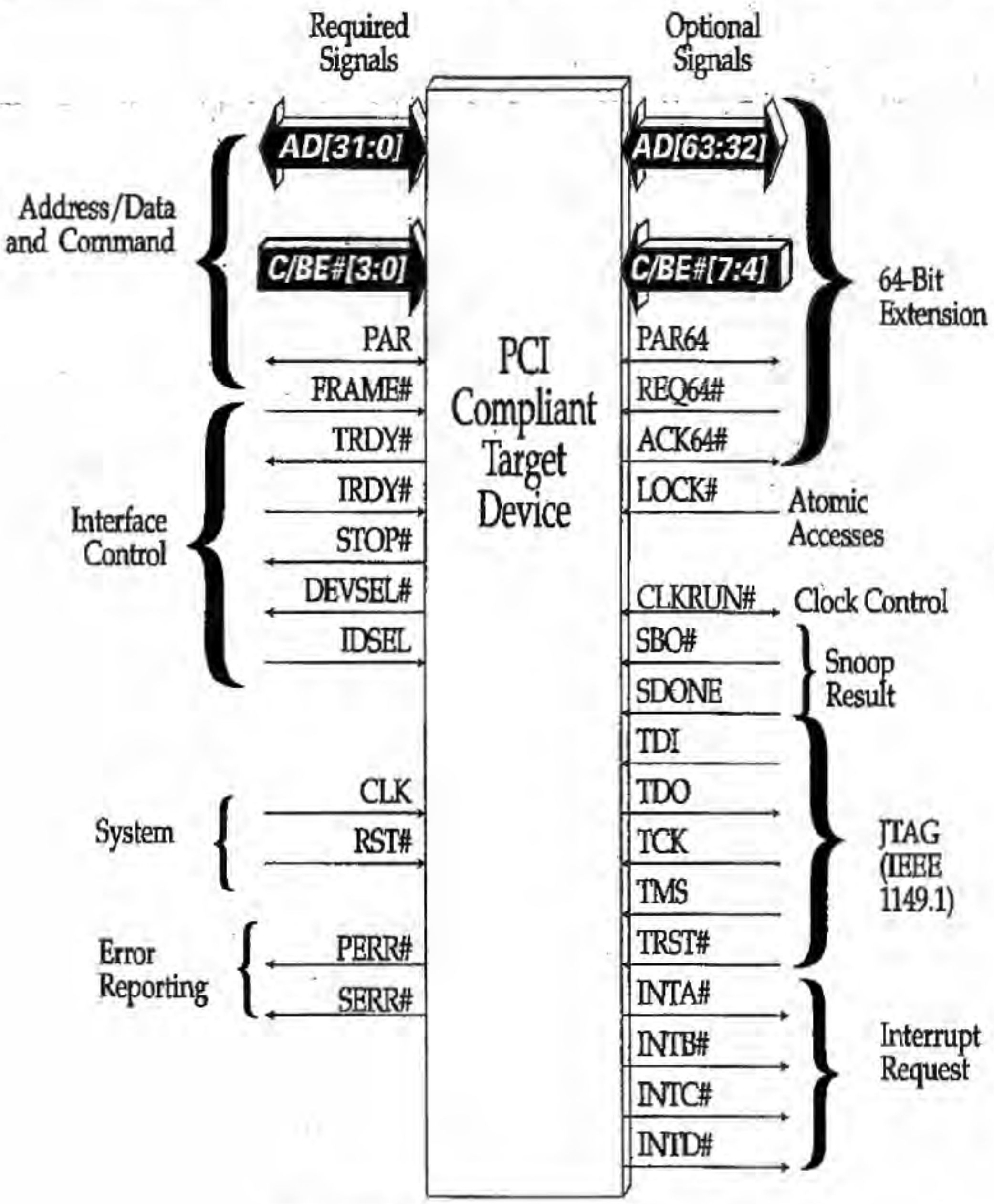

Figure 5-2. PCI-Compliant Target Device Signals

System Signals

PCI Clock Signal (CLK)

The CLK signal is an input to all devices residing on the PCI bus. It provides timing for all transactions, including bus arbitration. All inputs to PCI devices are sampled on the rising edge of the CLK signal. The state of all input signals are don't-care at all other times. All PCI timing parameters are specified with respect to the rising-edge of the CLK signal.

All actions on the PCI bus are synchronized to the PCI CLK signal. The frequency of the CLK signal may be anywhere from 0MHz to 33MHz. The revision 1.0 PCI specification stated that all devices must support operation from 16 to 33MHz and it strongly recommended support for operation down to 0MHz for static debug and low power operation. The revision 2.x PCI specification indicates that **ALL** PCI devices (with one exception noted below) **MUST** support PCI operation within the 0MHz to 33MHz range.

The clock frequency may be changed at any time as long as:

- The clock edges remain clean.
- The minimum clock high and low times are not violated.
- There are no bus requests outstanding.
- LOCK# is not asserted.

The clock may only be stopped in a low state (to conserve power).

As an exception, components designed to be integrated onto the system board may be designed to operate at a fixed frequency (of up to 33MHz) and may only operate at that frequency.

For a discussion of 66MHz bus operation, refer to the chapter entitled "66MHz PCI Implementation."

CLKRUN# Signal

General

The CLKRUN# signal is optional and is defined for the mobile (i.e., portable) environment. It is not available on the PCI add-in connector. This section provides an introduction to this subject. A more detailed description of the mobile environment and the CLKRUN# signal's role can be found in the document entitled *PCI Mobile Design Guide* (available from the SIG).

Although the PCI specification states that the clock may be stopped or its frequency changed, it does not define a method for determining when to stop (or slow down) the clock, or a method for determining when to restart the clock.

A portable system includes a central resource that includes the PCI clock generation logic. With respect to the clock generation logic, the CLKRUN# signal is a sustained tri-state input/output signal. The clock generation logic keeps CLKRUN# asserted when the clock is running normally. During periods when the clock has been stopped (or slowed), the clock generation logic monitors CLKRUN# to recognize requests from master and target devices for a change to be made in the state of the PCI clock signal. The clock cannot be stopped if the bus is not idle. Before it stops (or slows down) the clock frequency, the clock generation logic deasserts CLKRUN# for one clock to inform PCI devices that the clock is about to stopped (or slowed). After driving CLKRUN# high (deasserted) for one clock, the clock generation logic tri-states its CLKRUN# output driver. The keeper resistor on CLKRUN# then assumes responsibility for maintaining the deasserted state of CLKRUN# during the period in which the clock is stopped (or slowed).

The clock continues to run unchanged for a minimum of four clocks after the clock generation logic deasserts CLKRUN#. After deassertion of CLKRUN#, the clock generation logic must monitor CLKRUN# for two possible cases:

1. After the clock has been stopped (or slowed), a master (or multiple masters) may require clock restart in order to request use of the bus. Prior to issuing the bus request, the master(s) must first request clock restart. This is accomplished by assertion of CLKRUN#. When the clock generation logic detects the assertion of CLKRUN# by another party, it turns on (or speeds up) the clock and turns on its CLKRUN# output driver to assert CLKRUN#. When the master detects that CLKRUN# has been asserted for

two rising-edges of the PCI CLK signal, the master may then tri-state its CLKRUN# output driver.

2. When the clock generation logic has deasserted CLKRUN#, indicating its intention to stop (or slow) the clock, the clock must continue to run for a minimum of four clocks. During this period of time, a target (or master) that requires continued clock operation (e.g., in order to perform internal housekeeping after the completion of a transaction), may reassert CLKRUN# for two PCI clock cycles to request continued generation of CLK. When the clock generation logic samples CLKRUN# reasserted, it continues to generate the clock (rather than stopping it or slowing it down). The specification doesn't define the period of time that the clock will continue to run after a request for continued operation. The author interprets this as implying that the period is system design-specific.

Reset Signal (RST#)

When asserted, the reset signal forces all PCI configuration registers, master and target state machines and output drivers to an initialized state. RST# may be asserted or deasserted asynchronously to the PCI CLK edge. The assertion of RST# also initializes other, device-specific functions, but this subject is beyond the scope of the PCI specification. All PCI output signals must be driven to their benign states. In general, this means they must be tri-stated. Exceptions are:

- SERR# is floated.
- If SBO# and SDONE cannot be tri-stated, they will be driven low.
- To prevent the AD bus, the C/BE bus and the PAR signals from floating during reset, they may be driven low by a central resource during reset.

Refer to the chapter entitled "The 64-Bit PCI Extension" for a discussion of the REQ64# signal's behavior during reset.

Address/Data Bus

The PCI bus uses a time-multiplexed address/data bus. During the address phase of a transaction:

- The **AD bus**, AD[31:0], carries the start address. The resolution of this address is on a doubleword boundary (address divisible by four) during a memory or a configuration transaction, or a byte-specific address during an I/O read or write transaction. Additional information on memory and I/O addressing can be found in the chapter entitled "The Read and Write Transfer." Additional information on configuration addressing can be found in parts III and IV of this book.

- The **Command or Byte Enable bus, C/BE#[3:0]**, defines the type of transaction. The chapter entitled "The Commands" defines the transaction types.
- The **Parity signal, PAR**, is driven by the initiator one clock after completion of the address phase and completion of each data phase of write transactions. It is driven by the currently-addressed target one clock after the completion of each data phase of read transactions. One clock after completion of the address phase, the initiator drives PAR either high or low to ensure even parity across the address bus, AD[31:0], and the four Command/Byte Enable lines, C/BE#[3:0]. Refer to the chapter entitled "Error Detection and Handling" for a discussion of parity.

During each data phase:

- The data bus, **AD[31:0]**, is driven by the initiator (during a write) or the currently-addressed target (during a read).
- **PAR** is driven by either the initiator (during a write) or the currently-addressed target (during a read) one clock after completion of the data phase and ensures even parity across AD[31:0] and C/BE#[3:0]. If all four data paths are not being used during a data phase, the agent driving the data bus (the master during a write or the target during a read) must ensure that valid data is being driven onto all data paths (including those not being used to transfer data). This is necessary because PAR must reflect even parity across the entire AD and C/BE buses.
- The Command/Byte Enable bus, **C/BE#[3:0]**, is driven by the initiator to indicate the bytes to be transferred within the currently-addressed doubleword and the data paths to be used to transfer the data. Table 5-1 indicates the mapping of the byte enable signals to the data paths and to the locations within the currently-addressed doubleword. Table 5-2 defines the interpretation of the byte enable signals during each data phase. Any combination of byte enables is considered valid and the byte enables may change from data phase to data phase.

Table 5-1. Byte Enable Mapping To Data Paths and Locations Within the Currently-Addressed Doubleword

Byte Enable Signal	Maps To
C/BE3#	Data path 3, AD[31:24], and the fourth location in the currently-addressed doubleword.
C/BE2#	Data path 2, AD[23:16], and the third location in the currently-addressed doubleword.
C/BE1#	Data path 1, AD[15:8], and the second location in the currently-addressed doubleword.
C/BE0#	Data path 0, AD[7:0], and the first location in the currently-addressed doubleword.

Table 5-2. Interpretation of the Byte Enables During a Data Phase

C/BE3#	C/BE2#	C/BE1#	C/BE0#	Meaning
0	0	0	0	The initiator intends to transfer all four bytes within the currently-addressed doubleword using all four data paths.
0	0	0	1	The initiator intends to transfer the upper three bytes within the currently-addressed doubleword using the upper three data paths.
0	0	1	0	The initiator intends to transfer the upper two bytes and the first byte within the currently-addressed doubleword using the upper two data paths and the first data path.
0	0	1	1	The initiator intends to transfer the upper two bytes within the currently-addressed doubleword using the upper two data paths.
0	1	0	0	The initiator intends to transfer the upper byte and the lower two bytes within the currently-addressed doubleword using the upper data path and the lower two data paths.

C/BE3#	C/BE2#	C/BE1#	C/BE0#	Meaning
0	1	0	1	The initiator intends to transfer the second and the fourth bytes within the currently-addressed doubleword using the second and fourth data paths.
0	1	1	0	The initiator intends to transfer the first and the fourth bytes within the currently-addressed doubleword using the first and the fourth data paths.
0	1	1	1	The initiator intends to transfer the upper byte within the currently-addressed doubleword using the upper data path.
1	0	0	0	The initiator intends to transfer the lower three bytes within the currently-addressed doubleword using the lower three data paths.
1	0	0	1	The initiator intends to transfer the middle two bytes within the currently-addressed doubleword using the middle two data paths.
1	0	1	0	The initiator intends to transfer the first and third bytes within the currently-addressed doubleword using the first and the third data paths.
1	0	1	1	The initiator intends to transfer the third byte within the currently-addressed doubleword using the third data path.
1	1	0	0	The initiator intends to transfer the lower two bytes within the currently-addressed doubleword using the lower two data paths.
1	1	0	1	The initiator intends to transfer the second byte within the currently-addressed doubleword using the second data path.
1	1	1	0	The initiator intends to transfer the first byte within the currently-addressed doubleword using the first data path.

C/BE3#	C/BE2#	C/BE1#	C/BE0#	Meaning
1	1	1	1	The initiator does not intend to transfer any of the four bytes within the currently-addressed doubleword and will not use any of the data paths. This is a null data phase.

Preventing Excessive Current Drain

If the inputs to CMOS input receivers are permitted to float for long periods, the receivers tend to oscillate and draw excessive current. In order to prevent this phenomena and preserve the green nature of the PCI bus, several rules are applied:

- When the bus is idle and no bus masters are requesting ownership, either the bus arbiter or a master that has the bus parked on it must enable its AD, C/BE and PAR output drivers and drive a stable pattern onto these signal lines. This issue is discussed in the chapter entitled "PCI Bus Arbitration" under the heading "Bus Parking."
- During a data phase in a write transaction, the initiator must drive a stable pattern onto the AD bus when it is not yet ready to deliver the next set of data bytes. This subject is covered in the chapter entitled "The Read and Write Transfers."
- During a data phase in a read transaction, the target must drive a stable pattern onto the AD bus when it is not yet ready to deliver the next set of data bytes. This subject is covered in the chapter entitled "The Read and Write Transfers."
- A 64-bit card plugged into a 32-bit expansion slot must keep its AD[63:32], C/BE#[7:4] and PAR64 input receivers from floating. This subject is covered in the chapter entitled "The 64-bit PCI Extension."

Transaction Control Signals

Table 5-3 provides a brief description of each signal used to control a PCI transfer.

Table 5-3. PCI Interface Control Signals

Signal	Master	Target	Description
FRAME#	In/Out	In	**Cycle Frame** is driven by the current initiator and indicates the start (when it's first asserted) and duration (the duration of its assertion) of a transaction. In order to determine that bus ownership has been acquired, the master must sample FRAME# and IRDY# both deasserted on the same rising-edge of the PCI CLK signal. A transaction may consist of one or more data transfers between the current initiator and the currently-addressed target. FRAME# is deasserted when the initiator is ready to complete the final data phase.
TRDY#	In	Out	**Target Ready** is driven by the currently-addressed target. It is asserted when the target is ready to complete the current data phase (data transfer). A data phase is completed when the target is asserting TRDY# and the initiator is asserting IRDY# at the rising-edge of the CLK signal. During a read, TRDY# asserted indicates that the target is driving valid data onto the data bus. During a write, TRDY# asserted indicates that the target is ready to accept data from the master. Wait states are inserted in the current data phase until both TDRY# and IRDY# are sampled asserted.
IRDY#	In/Out	In	**Initiator Ready** is driven by the current bus master (the initiator of the transaction). During a write, IRDY# asserted indicates that the initiator is driving valid data onto the data bus. During a read, IRDY# asserted indicates that the initiator is ready to accept data from the currently-addressed target. In order to determine that bus ownership has been acquired, the master must sample FRAME# and IRDY# both deasserted on the same rising-edge of the PCI CLK signal. Also refer to the description of TRDY# in this table.

Signal	Master	Target	Description
STOP#	In	Out	The target asserts **STOP#** to indicate that it wishes the initiator to stop the transaction in progress on the current data phase.
IDSEL	In	In	**Initialization Device Select** is an input to the PCI device and is used as a chip select during an access to one of the device's configuration registers. For additional information, refer to the chapter entitled "Configuration Transactions."
LOCK#	In/Out	In	Used by the initiator to lock the currently-addressed memory target during an atomic transaction series (e.g., during a semaphore read/modify/write operation). Refer to the description (in this chapter) under the heading "Resource Locking" and to the chapter entitled "Shared Resource Acquisition."
DEVSEL#	In	Out	**Device Select** is asserted by a target when the target has decoded its address. It acts as an input to the current initiator and the subtractive decoder in the expansion bus bridge. If a master initiates a transfer and does not detect DEVSEL# active within six CLK periods, it must assume that the target cannot respond or that the address is unpopulated. A master-abort results.

Arbitration Signals

Each PCI master has a pair of arbitration lines that connect it directly to the PCI bus arbiter. When a master requires the use of the PCI bus, it asserts its device-specific REQ# line to the arbiter. When the arbiter has determined that the requesting master should be granted control of the PCI bus, it asserts the GNT# (grant) line specific to the requesting master. In the PCI environment, bus arbitration can take place while another master is still in control of the bus. This is known as "hidden" arbitration. When a master receives a grant from the bus arbiter, it must wait for the current initiator to complete its transfer before initiating its own transfer. It cannot assume ownership of the PCI bus until FRAME# is sampled deasserted (indicating the start of the last data phase) and IRDY# is then sampled deasserted (indicating the completion of the last data phase). This indicates that the current transaction has been completed and the bus has been returned to the idle state. Bus arbitration is discussed in more detail in the chapter entitled "PCI Bus Arbitration."

While RST# is asserted, all masters must tri-state their REQ# output drivers and must ignore their GNT# inputs. In a system with PCI add-in connectors, the arbiter may require a weak pullup on the REQ# inputs that are wired to the add-in connectors. This will keep them from floating when the connectors are unoccupied.

Interrupt Request Signals

PCI agents that must generate requests for service can utilize one of the PCI interrupt request lines, INTA#, INTB#, INTC# or INTD#. A description of these signals can be found in the chapters entitled "Interrupt-Related Issues."

Error Reporting Signals

The sections that follow provide an introduction to the PERR# and SERR# signals. The chapter entitled "Error Detection and Handling" provides a more detailed discussion of error detection and handling.

Data Parity Error

The **generation of parity information is mandatory for all PCI devices** that drive address or data information onto the AD bus. This is a requirement because the agent driving the AD bus must assume that the agent receiving the data and parity will check the validity of the parity and may either flag an error or even fail the machine if incorrect parity is received.

The **detection and reporting of parity errors by PCI devices is generally required.** The specification is written this way to indicate that, in some cases, the designer may choose to ignore parity errors. An example might be a video frame buffer. The designer may choose not to verify the correctness of the data being written into the video memory by the initiator. In the event that corrupted data is received and written into the frame memory, the only effect will be one or more corrupted video pixels displayed on the screen. Although this may have a deleterious effect on the end user's peace of mind, it will not corrupt programs or the data structures associated with programs.

Implementation of the PERR# pin is required on all add-in PCI cards (and is generally required on system board devices). The data parity error signal, PERR#, may be pulsed by a PCI device under the following circumstances:

- In the event of a data parity error detected by a PCI target during a write data phase, the target must set the DATA PARITY SIGNALED bit in its PCI configuration status register and must assert PERR# (if the PARITY RESPONSE ENABLE bit in its configuration command register is set to one). It may then either continue the transaction or may assert STOP# to terminate the transaction prematurely. During a burst write, the initiator is responsible for monitoring the PERR# signal to ensure that each data item is not corrupted in flight while being written to the target.
- In the event of a data parity error detected by the PCI initiator during a read data phase, the initiator must set the DATA PARITY SIGNALED bit in its PCI configuration status register and must assert PERR# (if the PARITY RESPONSE ENABLE bit in its configuration command register is set to one). The platform designer may include third-party logic that monitors PERR# or may leave error reporting up to the initiator.

To ensure that correct parity is available to any PCI devices that perform parity checking, all PCI devices must generate even parity on AD[31:0], C/BE#[3:0] and PAR for the address and data phases. PERR# is implemented as an output on targets and as both an input and an output on masters. The initiator of a transaction has responsibility for reporting the detection of a data parity error to software. For this reason, it must monitor PERR# during write data phases to determine if the target has detected a data parity error. The action taken by an initiator when a parity error is detected is design-dependent. It may perform retries with the target or may choose to terminate the transaction and generate an interrupt to invoke its device-specific interrupt handler. If the initiator reports the failure to software, it must also set the DATA PARITY REPORTED bit in its PCI configuration status register. PERR# is only driven by one device at time.

A detailed discussion of data parity error detection and handling may be found in the chapter entitled "Error Detection and Handling."

System Error

The System Error signal, SERR#, may be pulsed by any PCI device to report address parity errors, data parity errors during a special cycle, and critical errors other than parity. SERR# is required on all add-in PCI cards that perform address parity checking or report other serious errors using SERR#. This signal is considered a "last-recourse" for reporting serious errors. Non-catastrophic and correctable errors should be signaled in some other way. In a PC-compatible machine, SERR# typically causes an NMI to the system processor (although the designer is not constrained to have it generate an NMI). In a PowerPC™ PREP-compliant platform, assertion of SERR# is reported to the host processor via assertion of TEA# or MC# and causes a machine check interrupt. This is the functional equivalent of NMI in the Intel world. If the designer of a PCI device does not want an NMI to be initiated, some means other than SERR# should be used to flag an error condition (such as setting a bit in the device's status register and generating an interrupt request). SERR# is an open-drain signal and may be driven by more than one PCI agent at a time. When asserted, the device drives it low for one clock and then tri-states its output driver. The keeper resistor on SERR# is responsible for returning it to the deasserted state.

A detailed discussion of system error detection and handling may be found in the chapter entitled "Error Detection and Handling."

Cache Support (Snoop Result) Signals

Table 5-4 provides a brief description of the optional PCI cache support signals. The chapter entitled "PCI Cache Support" provides a more detailed explanation of cache support implementation.

Table 5-4. Cache Snoop Result Signals

Signal	Description
SBO#	*Snoop Back Off.* This signal is an output from the PCI cache/bridge and an input to cacheable memory subsystems residing on the PCI bus. It is asserted by the bridge to indicate that the PCI memory access in progress is about to read or update stale information in memory. SBO# is qualified by and only has meaning when the SDONE signal is also asserted by the bridge. When SDONE and SBO# are sampled asserted, the currently-addressed cacheable PCI memory subsystem should respond by signaling a retry to the current initiator.
SDONE	*Snoop Done.* This signal is an output from the PCI cache/bridge and an input to cacheable memory subsystems residing on the PCI bus. It is deasserted by the bridge while the processor's cache(s) snoops a memory access started by the current initiator. The bridge asserts SDONE when the snoop has been completed. The results of the snoop are then indicated on the SBO# signal. SBO# sampled deasserted indicates that the PCI initiator is accessing a clean line in memory and the PCI cacheable memory target is permitted to accept or supply the indicated data. SBO# sampled asserted indicates that the PCI initiator is accessing a stale line in memory and should not complete the data access. Instead, the memory target should terminate the access by signaling a retry to the PCI initiator.

The specification recommends that systems that do not support cacheable memory on the PCI bus should supply pullups on the SDONE and SBO# pins at each add-in connector.

In order to guarantee proper operation in systems that do not support cacheable memory on the PCI bus, cacheable PCI memory targets must ignore SDONE and SBO# after reset is deasserted. If the system supports cacheable PCI memory, the configuration software will write the system cache line size into the target's cache line size configuration register.

64-bit Extension Signals

The PCI specification provides a detailed definition of a 64-bit extension to its baseline 32-bit architecture. Systems that implement the extension support the transfer of up to eight bytes per data phase between a 64-bit initiator and a 64-bit target. The signals involved are defined in table 5-5. A more detailed explanation can be found in the chapter entitled "The 64-Bit PCI Extension."

Table 5-5. The 64-Bit Extension

Signal	Description
AD[63:32]	Upper four data lanes. In combination with AD[31:0], extends the width of the data bus to 64 bits. These pins aren't used during the address phase of a transfer (unless 64-bit addressing is being used).
C/BE#[7:4]	**Byte Enables** for data lanes four-through-seven. Used during the data transfer phase, but not during the address phase (unless 64-bit addressing is being used.)
REQ64#	**Request 64-bit Transfer.** Generated by the current initiator to indicate its desire to perform transfers using one or more of the upper four data paths. REQ64# has the same timing as the FRAME# signal. Refer to the chapter entitled "The 64-Bit PCI Extension" for more information.
ACK64#	**Acknowledge 64-bit Transfer.** Generated by the currently-addressed target (if it supports 64-bit transfers) in response to a REQ64# assertion by the initiator. ACK64# has the same timing as the DEVSEL# signal.
PAR64	**Parity for the upper doubleword.** This is the even parity bit associated with AD[63:32] and BE#[7:4]. For additional information, refer to the chapters entitled "The 64-bit PCI Extension" and "Error Detection and Handling."

Resource Locking

The LOCK# signal should be utilized by a PCI initiator that requires exclusive access to a target memory device during two or more separate transactions. The intended use of this function is to support read/modify/write memory semaphore operations. It is **not** intended as a mechanism that permits an initiator to dominate a target device or the bus in general.

If a PCI device implements executable memory or memory that contains system data (managed by the operating system), it must implement the locking function. It is recommended that a host/PCI bridge that has system memory on the host side implement the locking function. Some host bus architectures, however, do support memory locking. For this reason, the specification recommends but does not require that a host/PCI bridge support locking when acting as the target of a system memory access by a PCI master. Since the device driver associated with a PCI master cannot depend on the ability to lock system memory, the specification recommends that the driver use some type of software protocol to gain exclusive access to code or data structures shared with other processors in the system.

An initiator requiring exclusive access to a target may use the LOCK# signal if it isn't currently being driven by another initiator. When the target device is addressed and LOCK# is deasserted by the initiator during the address phase and then asserted during the data phase, the target device is reserved for as long as the LOCK# signal remains asserted. If the target is subsequently addressed by another initiator while the lock is still in force, the target issues a retry to the initiator. While a target is locked, other bus masters (that don't require exclusive access to a target) are permitted to acquire the bus to access targets other than the locked target.

A more detailed description of the PCI locking capability can be found in the chapter entitled "Shared Resource Acquisition."

JTAG/Boundary Scan Signals

The designer of a PCI device may optionally implement the IEEE 1149.1 Boundary Scan interface signals to permit in-circuit testing of the PCI device. The related signals are defined in table 5-6. A detailed discussion of boundary scan is beyond the scope of this publication.

Table 5-6. Boundary Scan Signals

Signal	Description
TCK	***Test Clock.*** Used to clock state information and data into and out of the device during boundary scan.
TDI	***Test Input.*** Used (in conjunction with TCK) to shift data and instructions into the Test Access Port (TAP) in a serial bit stream.
TDO	***Test Output.*** Used (in conjunction with TCK) to shift data out of the Test Access Port (TAP) in a serial bit stream.
TMS	***Test Mode Select.*** Used to control the state of the Test Access Port controller.
TRST#	***Test Reset.*** Used to force the Test Access Port controller into an initialized state.

Interrupt Request Lines

The PCI interrupt request signals (INTA#, INTB#, INTC# and INTD#) are discussed in the chapters entitled "Interrupt-Related Issues" and "The Configuration Registers."

Sideband Signals

A sideband signal is defined as a signal that is not part of the PCI bus standard and interconnects two or more PCI agents. This signal only has meaning for the agents it interconnects. The following are some examples of sideband signals:

- A PCI bus arbiter could monitor a "busy" signal from a PCI device (such as an EISA or Micro Channel™ expansion bus bridge) to determine if the device is available before granting the PCI bus to a PCI initiator.
- PC compatibility signals like A20GATE, CPU RESET, etc.

Signal Types

The signals that comprise the PCI bus are electrically defined in one of the following fashions:

- **IN** defines a signal as a standard input-only signal.
- **OUT** defines a signal as a standard output-only signal.
- **T/S** defines a signal as a bi-directional, tri-state input/output signal.
- **S/T/S** defines a signal as a sustained tri-state signal that is driven by only one owner at a time. An agent that drives an s/t/s pin low must actively drive it high for at least one clock before tri-stating it. A pullup resistor is required to sustain the inactive state until another agent takes ownership of and drives the signal. The resistor is supplied as a central resource in the system design. The next owner of the signal cannot start driving the s/t/s signal any sooner than one clock after it is released by the previous owner.

O/D defines a signal as an open drain. It is wire-ORed with other agents. The signaling agent asserts the signal, but returning the signal to the inactive state is accomplished by a weak pull-up resistor. The deasserted state is maintained by the pullup resistor. The pullup may take two or three PCI clock periods to fully restore the signal to the deasserted state. Table 5-7 defines the PCI signal types.

Table 5-7. PCI Signal Types

Signal(s)	Type
CLK	IN
RST#	IN
AD[31:0]	T/S
C/BE#[3:0]	T/S
PAR	T/S
FRAME#	S/T/S
TRDY#	S/T/S
IRDY#	S/T/S
STOP#	S/T/S
LOCK#	S/T/S
IDSEL	IN
DEVSEL#	S/T/S
REQ#	T/S
GNT#	T/S
PERR#	S/T/S
SERR#	O/D
SBO#	IN or OUT
SDONE	IN or OUT
AD[63:32]	T/S
C/BE#[7:4]	T/S
REQ64#	S/T/S
ACK64#	S/T/S
PAR64	T/S
TCK	IN
TDI	IN
TDO	OUT
TMS	IN
TRST#	IN
INTA# - INTD#	O/D

Central Resource Functions

Any platform that implements the PCI bus must supply a toolbox of support functions necessary for the proper operation of all PCI devices. Some examples would include:

- **PCI bus arbiter.** The arbiter is necessary to support PCI masters. The PCI specification does not define the decision-making process utilized by the PCI bus arbiter. The design of the arbiter is therefore platform-specific.

- **Pullup resistors** on signals that are not always driven to a valid state. This would include: all of the s/t/s signals; AD[63:32]; C/BE#[7:4]; PAR64; and SERR#.
- **Error logic** responsible for converting SERR# to the platform-specific signal (e.g., NMI in an Intel-based platform or TEA# in a PowerPC™-based platform) utilized to alert the host processor that an error has occurred.
- Central resource to **generate** the proper **IDSEL** signal when a PCI device's configuration space is being addressed (this function is typically performed by the host/PCI bridge).
- System board logic to **assert REQ64# during reset**. A detailed description of this function is provided in the chapter entitled "The 64-Bit PCI Extension."
- **Subtractive decoder**. Each PCI target device must implement positive decode. In other words, it must decode any address placed on the PCI bus to determine if it is the target of the current transaction. Only one agent on the PCI bus may implement subtractive decode. This is typically the expansion bus (e.g., EISA, ISA, or Micro Channel) bridge.

Subtractive Decode

Background

The expansion bus bridge can claim transactions in one of two fashions:

1. When a transaction is not claimed by any other PCI device within a specified period of time, the PCI/expansion bus bridge may assert DEVSEL# and pass the transaction through to the expansion bus. It can determine that no other PCI device has claimed a transaction by monitoring the state of the DEVSEL# signal generated by the other PCI-compliant devices. If DEVSEL# is not sampled asserted within four clock periods after the start of a transaction, no other PCI device has claimed the transaction. The expansion bus bridge may then claim the transaction by asserting DEVSEL# during the period between the fifth and sixth clocks of the transaction. This is referred to as subtractive decode. Additional information regarding subtractive decode can be found in the chapter entitled "Premature Transaction Termination" in the section entitled "Master Abort."
2. Since this would result in very poor access time when accessing expansion bus devices, the expansion bus bridge may employ positive address decode. During system configuration, the bridge is configured to recognize certain memory and/or IO address ranges. Upon recognizing an address

within this pre-assigned range, the bridge may assert DEVSEL# immediately (without waiting for the DEVSEL# timeout) to claim the transaction. The bridge then passes the transaction through onto the expansion bus.

The ISA bus environment is one that depends heavily on subtractive decoding to claim transactions. Because most ISA bus devices are not plug and play-capable, the configuration software cannot automatically detect their presence and assign address ranges to their address decoders. The ISA bridge uses subtractive decode to claim all transactions that meet the following criteria:

- No other PCI device has claimed the transaction. By definition, all PCI device address decoders are fast (decodes address and asserts DEVSEL# during the clock cell immediately following completion of the address phase), medium (asserts DEVSEL# during the second clock cell after completion of the address phase) or slow (asserts DEVSEL# during the third clock after completion of the address phase). If the ISA bridge does not detect DEVSEL# asserted by any other PCI device (and the target address "makes sense" for the ISA environment), the bridge asserts DEVSEL# during the fourth clock after completion of the address phase. The transaction is then initiated on the ISA bus.
- The target address is one that falls within the overall ISA memory or I/O address ranges. Any memory address below 16MB that goes unclaimed by PCI devices is claimed and passed through to the ISA bus. Any I/O address in the lower 64KB of I/O space that goes unclaimed by PCI devices is claimed and passed through to the ISA bus.

Tuning Subtractive Decoder

This means that a transaction initiated by the host processor (or any other bus master) does not appear on the ISA bus until four or five clocks after the completion of the address phase on the PCI bus. The processor's performance when accessing ISA devices is therefore substantially degraded. In order to minimize the effect of subtractive decode on performance, the ISA bridge designer can permit the subtractive decoder to be "tuned." During the configuration process, the configuration software reads the configuration status register for every device on the PCI bus. One of the required fields in the status register is the DEVSEL# timing field, indicating whether the device has a fast, medium or slow address decoder. As an example, if every device on the PCI bus indicates that it has a fast decoder, the software can program the subtractive decoder to assert DEVSEL# and claim the transaction during the second clock after the completion of the address phase (if it doesn't detect DEVSEL# asserted during the first clock after the address phase).

Reading Timing Diagrams

Figure 5-3 illustrates a typical PCI timing diagram. When a PCI signal is asserted or deasserted by a PCI device, the output driver utilized is typically a weak CMOS driver. This being the case, the driver isn't capable of transitioning the signal line past the logic threshold for a logic high or low in one step. The voltage change initiated on the signal line propagates down the trace until it hits the physical end of the trace. As it passes the stub for each PCI device along the way, the wavefront has not yet transitioned past the new logic threshold, so the change isn't detected by any of the devices. When reflected back along the trace, however, the reflection doubles the voltage change on the line, causing it to cross the logic threshold. As the doubled wavefront propagates back down the length of the trace, the signal's new state is detected by each device it passes. The time it takes the signal to travel the length of the bus and reflect back is referred to as the Tprop, or propagation delay. This delay is illustrated in the timing diagrams.

As an example, a master samples FRAME# and IRDY# deasserted (bus idle) and its GNT# asserted on the rising-edge of clock one, indicating that it has bus acquisition. The master initiates the transaction during clock cell one by asserting the FRAME# signal to indicate the start of the transaction. In the timing diagram, FRAME# isn't shown transitioning from high-to-low until sometime after the rising-edge of clock one and before the rising-edge of clock two, thereby illustrating the propagation delay. Coincident with FRAME# assertion, the initiator drives the start address onto the AD bus during clock cell one, but the address change isn't valid until sometime after the rising-edge of clock one and before the rising-edge of clock two.

The address phase ends on the rising-edge of clock two and the initiator begins to turn off its AD bus drivers. The time that it takes the driver to actually cease driving the AD bus is illustrated in the timing diagram (the initiator has not successfully disconnected from the AD bus until sometime during clock cell two).

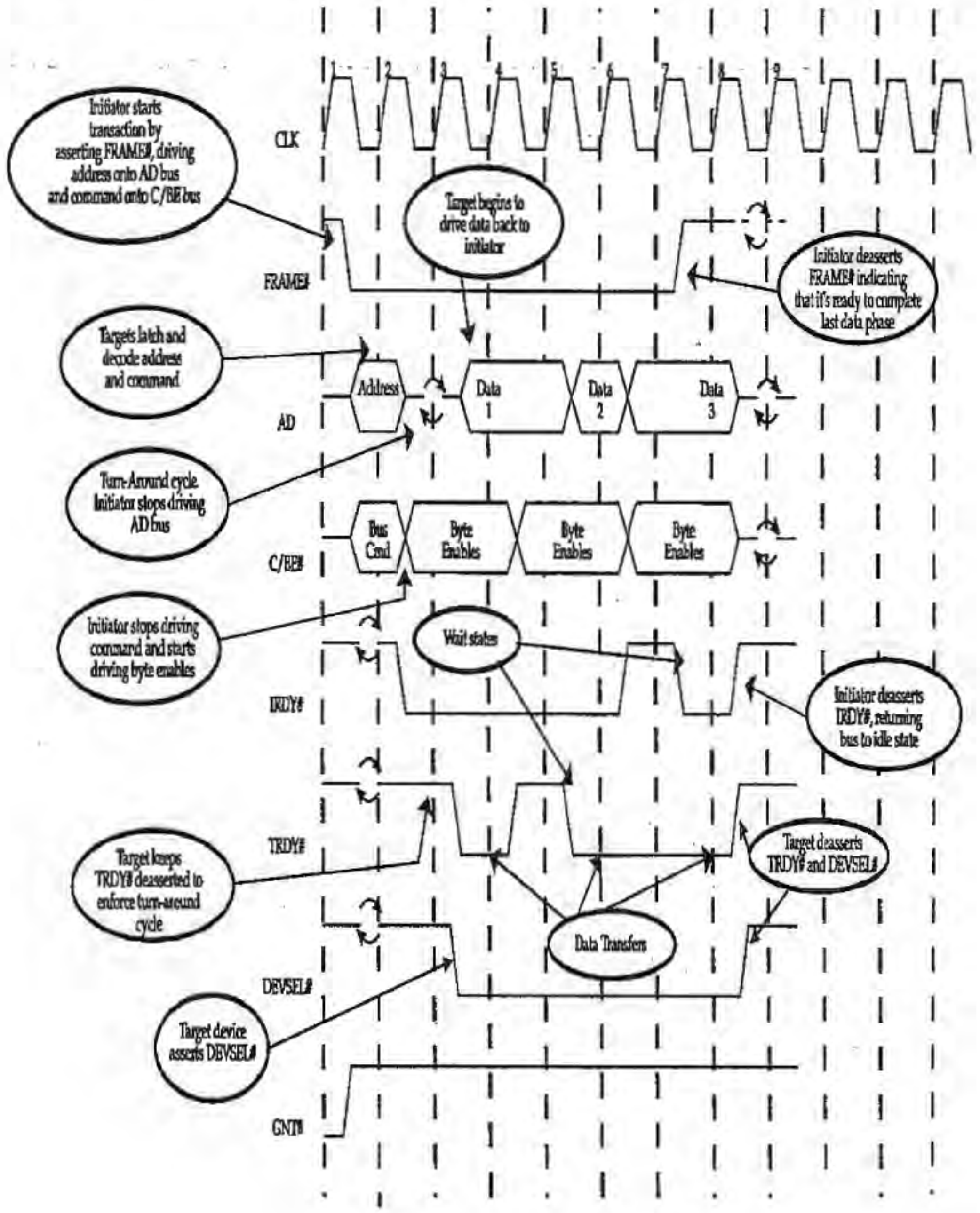
1
2
3
4
5
6
7
8
9
CLK
FRAME#
AD
C/BE#
IRDY#
TRDY#
DEVSEL#
GNT#
Address
Data 1
Data 2
Data 3
Bus Cmd
Byte Enables
Byte Enables
Byte Enables
Initiator starts transaction by asserting FRAME#, driving address onto AD bus and command onto C/BE bus
Target begins to drive data back to initiator
Initiator deasserts FRAME# indicating that it's ready to complete last data phase
Targets latch and decode address and command
Turn-Around cycle. Initiator stops driving AD bus
Initiator stops driving command and starts driving byte enables
Wait states
Initiator deasserts IRDY#, returning bus to idle state
Target keeps TRDY# deasserted to enforce turn-around cycle
Target deasserts TRDY# and DEVSEL#
Data Transfers
Target device asserts DEVSEL#

Figure 5-3. Typical PCI Timing Diagram

Chapter 6

The Previous Chapter

The previous chapter provided a detailed description of the PCI functional signal groups.

This Chapter

When a PCI bus master requires the use of the PCI bus to perform a data transfer, it must request the use of the bus from the PCI bus arbiter. This chapter provides a detailed discussion of the PCI bus arbitration timing. The PCI specification defines the timing of the request and grant handshaking, but not the procedure used to determine the winner of a competition. The algorithm used by a system's PCI bus arbiter to decide which of the requesting bus masters will be granted use of the PCI bus is system-specific and outside the scope of the specification.

The Next Chapter

The next chapter describes the transaction types, or commands, that the initiator may utilize when it has successfully acquired PCI bus ownership.

Arbiter

At a given instant in time, one or more PCI bus master devices may require use of the PCI bus to perform a data transfer with another PCI device. Each requesting master asserts its REQ# output to inform the bus arbiter of its pending request for the use of the bus. Figure 6-1 illustrates the relationship of the PCI masters to the central PCI resource known as the bus arbiter. In this example, there are seven possible masters connected to the PCI bus arbiter in the illustration. Each master is connected to the arbiter via a separate pair of REQ#/GNT# signals. Although the arbiter is shown as a separate component, it usually is integrated into the PCI chip set; specifically, it is typically integrated into the host/PCI or the PCI/expansion bus bridge chip.

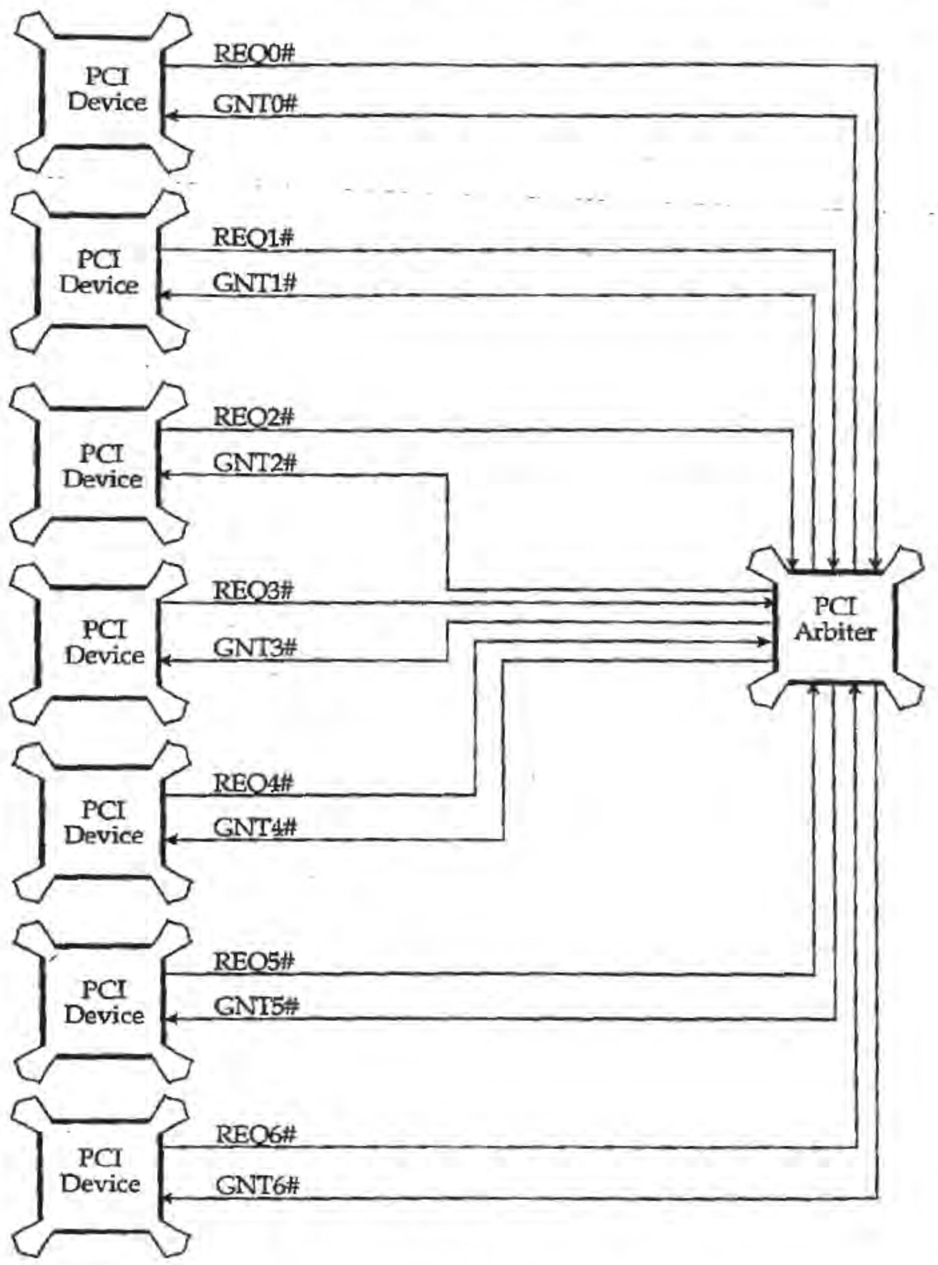

Figure 6-1. The PCI Bus Arbiter

Arbitration Algorithm

As stated at the beginning of this chapter, the PCI specification does not define the scheme used by the PCI bus arbiter to decide the winner of the competition when multiple masters simultaneously request bus ownership. The arbiter may utilize any scheme, such as one based on fixed or rotational priority or a combination of the two (rotational between one group of masters and fixed within another group). The 2.1 specification states that the arbiter is required to implement a fairness algorithm to avoid deadlocks. The exact verbiage that is used is:

> The central arbiter is required to implement a fairness algorithm to avoid deadlocks. Fairness means that each potential bus master must be granted access to the bus independent of other requests. However, this does not mean that all agents are required to have equal access to the bus. By requiring a fairness algorithm there are no special conditions to handle when LOCK# is active (assuming a resource lock) or when cacheable memory is located on PCI. A system that uses a fairness algorithm is still considered fair if it implements a complete bus lock instead of a resource lock. However, the arbiter must advance to a new agent if the initial transaction attempting to establish a lock is terminated with retry.

While the statements made regarding lock are clear, the definition of fairness contained in the above text was not clear to the author. Fairness is defined as a policy that ensures that high-priority masters will not dominate the bus to the exclusion of lower-priority masters when they are continually requesting the bus.

The specification contains an example arbiter implementation that does clarify the intent of the specification. The example follows this section.

Ideally, the bus arbiter should be programmable by the system. The startup configuration software can determine the priority to be assigned to each member of the bus master community by reading from the maximum latency (Max_Lat) configuration register associated with each bus master. The bus master designer hardwires this register to indicate, in increments of 250ns, how quickly the master requires access to the bus in order to achieve adequate performance.

In order to grant the PCI bus to a bus master, the arbiter asserts the device's respective GNT# signal. This grants the bus to the master for one transaction (consisting of one or more data phases).

If a master generates a request, is subsequently granted the bus and does not initiate a transaction (assert FRAME#) within 16 PCI clocks after the bus goes idle, the arbiter may assume that the master is malfunctioning. In this case, the action taken by the arbiter would be system design-dependent.

Example Arbiter with Fairness

A system may divided the overall community of bus masters on a PCI bus into two categories:

1. Bus masters that require fast access to the bus or high throughput in order to achieve good performance. Examples might be the video adapter, an ATM network interface or an FDDI network interface.
2. Bus masters that don't require very fast access to the bus or high throughput in order to achieve good performance. Examples might be a SCSI host bus adapter or a standard expansion bus master.

The arbiter would segregate the REQ#/GNT# signals into two groups with greater precedence given to those in one group. Assume that bus masters A, B and C are in the group that requires fast access, while masters X, Y and Z are in the other group. The arbiter can be programmed or designed to treat each group as rotational priority within the group and rotational priority between the two groups. This is pictured in figure 6-2.

Assume the following conditions:

- Master A is the next to receive the bus in the first group.
- Master X is the next to receive it in the second group.
- A master in the first group is the next to receive the bus.
- All masters are asserting REQ# and wish to perform multiple transactions (i.e., they keep their respective REQ# asserted after starting a transaction).

The order in which the masters would receive access to the bus is:

1. Master A.
2. Master B.
3. Master X.

4. Master A.
5. Master B.
6. Master Y.
7. Master A.
8. Master B.
9. Master Z.
10. Master A.
11. Master B.
12. Master X, etc.

The masters in the first group are permitted to access the bus more frequently than those that reside in the second group.

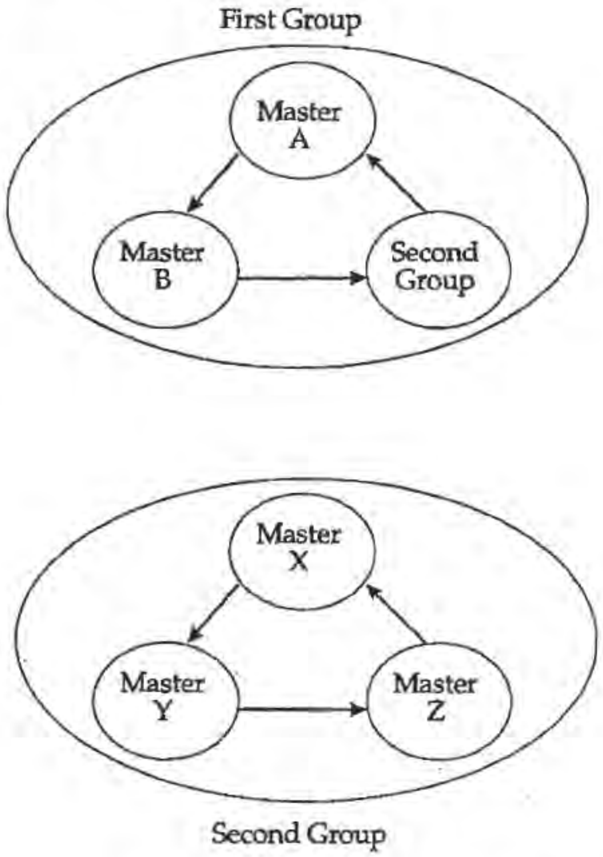

Figure 6-2. Example Arbitration Scheme

Master Wishes To Perform More Than One Transaction

If the master has another burst to perform immediately after the one it just initiated, it should keep its REQ# line asserted after it asserts FRAME# to begin the current transaction. This informs the arbiter of its desire to maintain ownership of the bus after completion of the current transaction. Depending on other pending requests, the arbiter may or may not permit the master to maintain bus ownership after the completion of the current transaction. In the event that ownership is not maintained, the master should keep its REQ# asserted until it is successful in acquiring bus ownership again.

At a given instant in time, only one bus master may use the bus. This means that no more than one GNT# line will be asserted by the arbiter during any PCI clock cycle.

Hidden Bus Arbitration

Unlike some arbitration schemes, the PCI scheme allows bus arbitration to take place while the current initiator is performing a data transfer. If the arbiter decides to grant ownership of the bus for the next transaction to a master other than the initiator of the current transaction, it removes the GNT# from the current initiator and issues GNT# to the next owner of the bus. The next owner cannot assume bus ownership, however, until the bus is idled by the current initiator. No bus time is wasted on a dedicated period of time to perform an arbitration bus cycle. This is referred to as hidden arbitration.

Bus Parking

A master must only assert its REQ# output to signal a current need for the bus. In other words, a master must not use its REQ# output to "park" the bus on itself. If a system designer implements a bus parking scheme, the bus arbiter design should indicate a default bus owner by asserting the device's GNT# signal when no request from any bus masters are currently pending. In this manner, a REQ# from the default bus master is granted immediately (if no other bus masters require the use of the PCI bus).

If the bus arbiter is designed to implement bus parking, it asserts GNT# to a default bus master when none of the REQ# lines are active. In this manner, the bus is immediately available to the default bus master if it should require the use of the bus (and no other higher-priority request is pending). If the master that the bus is parked on subsequently requires access to the PCI bus, it needn't assert its REQ#. Upon sampling bus idle (FRAME# and IRDY# deasserted) and its GNT# asserted, it can immediately initiate a transaction. The choice of which master to park the bus on is defined by the designer of the bus arbiter. Any process may be used, such as the last bus master to use the bus or a predefined default bus master.

There are two possible scenarios regarding the method utilized when implementing bus parking:

1. The arbiter may monitor FRAME# and IRDY# to determine if the bus is busy before parking the bus. Assume that a master requests the bus, receives its GNT# and starts a multiple data phase burst transaction. If it doesn't have another transaction to run after this one completes, it deasserts its REQ# when it asserts FRAME#. In this case, the arbiter may be designed to recognize that the bus is busy and, as a result, will not deassert the current master's grant to park the bus on another master.
2. The arbiter may not monitor for bus idle. Assume that a master requests the bus, receives its GNT# and starts a multiple data phase burst transaction. If it doesn't have another transaction to run after this one completes, it deasserts its REQ# when it asserts FRAME#. In this case, the arbiter may, in the absence of any requests from other masters, take away GNT# from the current master and issue GNT# to the master it intends to park the bus on. When the current master has exhausted its master latency timer and determines that it has lost its grant, it is forced to relinquish the bus, wait two clocks, and then rearbitrate for it again to resume the transaction at the point where it left off.

The specification recommends that the bus be parked on the last master that acquired the bus. In case two, then, the arbiter would continue to issue GNT# to the burst master and it can continue its transaction until either it is completed or until a request is received from another master.

When the arbiter parks the bus on a master (by asserting its grant) and the bus is idle, that master becomes responsible for keeping the AD bus, C/BE bus and PAR from floating (to keep the CMOS input buffers on all devices from oscillating and drawing excessive current). The master must enable its AD[31:0], C/BE#[3:0], and (one clock later) its PAR output drivers. The master doesn't have to turn on all of its output drivers in a single clock (it may take up to eight clocks, but two to three clocks is recommended). This procedure ensures that the bus doesn't float during bus idle periods. If the arbiter is not designed to park the bus, the arbiter itself should drive the AD bus, C/BE# lines and PAR during periods when the bus is idle.

Request/Grant Timing

When the arbiter determines that it is an master's turn to use the bus, it asserts the master's GNT# line. The arbiter may deassert a master's GNT# on any PCI clock. A master must ensure that its GNT# is asserted on the rising clock edge on which it wishes to start a transaction. If GNT# is deasserted, the transaction must not proceed. Once asserted by the arbiter, GNT# may be deasserted under the following circumstances:

- If GNT# is deasserted and FRAME# is asserted the transfer is valid and will continue. The deassertion of GNT# by the arbiter indicates that the master will no longer own the bus at the completion of the transaction currently in progress. The master keeps FRAME# asserted while the current transaction is still in progress. It deasserts FRAME# when it is ready to complete the final data phase.
- The GNT# to one master can be deasserted simultaneously with the assertion of another master's GNT# **if the bus isn't in the idle state**. The idle state is defined as a clock cycle during which both FRAME# and IRDY# are deasserted. If the bus were idle, the master whose GNT# is being removed may be using stepping to drive the bus (even though it hasn't asserted FRAME# yet; stepping is covered in the chapter entitled "The Read and Write Transfers"). The coincidental deassertion of its GNT# along with the assertion of another master's GNT# could result in contention on the AD bus. The other master could immediately start a transaction (because the bus is technically idle). The problem is prevented by delaying grant to the other master by one cycle. Table 6-1 defines the bus state as indicated by the current state of FRAME# and IRDY#.
- GNT# may be deasserted during the final data phase (FRAME# is deasserted) in response to the current bus master's REQ# being deasserted.

Table 6-1. Bus State

FRAME#	IRDY#	Description
deasserted	deasserted	Bus Idle.
deasserted	asserted	Initiator is ready to complete the last data transfer of a transaction, but it has not yet completed.
asserted	deasserted	A transaction is in progress and the initiator is not ready to complete the current data phase.
asserted	asserted	A transaction is in progress and the initiator is ready to complete the current data phase.

Example of Arbitration Between Two Masters

Figure 6-3 illustrates bus usage between two masters arbitrating for access to the PCI bus. The following assumptions must be made in order to interpret this example correctly:

- Bus master **A** requires the bus to perform two transactions. The first consists of a three data phase write and the second transaction type is a single data phase write.
- The arbitration scheme is fixed and bus master **B** has a higher priority than bus master **A**, or the scheme is rotational and it is **B**'s turn next.
- Bus master **B** only requires the bus to execute a single transaction consisting of one data phase.

It is important to remember that all PCI signals are sampled on the rising-edge of the PCI CLK signal. If the current owner of the bus requires the bus to perform additional transactions upon completion of the current transaction, it should keep its REQ# line asserted after assertion of FRAME# for the current transaction. If no other bus masters are requesting the use of the bus or the current bus master has the highest priority, the bus arbiter will continue to grant the bus to the current bus master at the conclusion of the current transaction.

The sample arbitration sequence pictured in figure 6-3 proceeds as follows:

1. Prior to clock edge one, bus master **A** asserts its REQ# to request access to the PCI bus. The arbiter samples its REQ# active at the rising-edge of clock one. At this point, bus master **B** doesn't yet require the bus. During clock cell one, the arbiter asserts GNT# to bus master **A**, granting it

ownership of the bus. During the same clock period, bus master **B** asserts its REQ#, indicating its desire to execute a transaction.

2. Bus master A samples its GNT# asserted on the rising-edge of clock two. In addition, it also samples IRDY# and FRAME# deasserted, indicating that the bus is in the idle state. In response, bus master A initiates the first of its two transactions. It asserts FRAME# and begins to drive the start address onto AD[31:0] and the command onto the Command/Byte Enable bus. If master A did not have another transaction to perform after this one, it would deassert its REQ# line during clock cell two. In this example, it does have another transaction to perform, so it keeps its REQ# line asserted.
3. The PCI bus arbiter samples the requests from bus masters A and **B** asserted at the rising-edge of clock two and begins the arbitration process to determine the next bus master.
4. During clock cell two, the arbiter removes the GNT# from master A. On the rising-edge of clock three, master A determines that it has been preempted, but continues its transaction because its LT timer has not yet expired (the LT timer is covered later in this chapter).
5. During clock cell three, the arbiter asserts bus master **B**'s GNT#. On the rising-edge of clock four, master **B** samples its GNT# asserted, indicating that it may be the next owner of the bus. It must continue to sample its GNT# on each subsequent rising-edge of the clock until it has bus acquisition. This is necessary because the arbiter may remove its grant and grant the bus to another party with a higher priority before the bus goes idle. Master **B** cannot begin to use the bus until the bus returns to the idle state.
6. Master A begins to drive the first data item onto the AD bus (this is a write transaction) during clock cell three, asserts the appropriate Command/Byte Enables (to indicate the data lanes to be used for the transfer) and asserts IRDY# to indicate to the target that the data is present on the bus. At the rising-edge of clock four, IRDY# and TRDY# are sampled asserted and the first data transfer takes place.
7. At the rising-edge of clock five, IRDY# and TRDY# are sampled asserted and the second data transfer takes place.
8. During clock cell five, master A keeps IRDY# asserted and deasserts FRAME#, indicating that the final data phase is in progress. At the rising-edge of clock six, IRDY# and TRDY# are sampled asserted and the third and final data transfer takes place.
9. During clock cell six, bus master A deasserts IRDY#, returning the bus to the idle state.

10. On the rising-edge of clock seven, master **B** samples FRAME# and IRDY# both deasserted and determines that the bus is now idle. It also samples its GNT# still asserted, indicating that it has bus acquisition. In response, it starts its transaction and turns off its REQ# line during clock cell seven (because it only requires the bus to perform this one transaction).
11. When it asserts FRAME# during clock cell seven, master **B** also begins driving the address onto the AD bus and the command onto Command/Byte Enable bus.
12. At the rising-edge of clock eight, the arbiter samples master **B**'s REQ# deasserted and master **A**'s REQ# still asserted. In response, the arbiter deasserts master **B**'s GNT# and asserts master **A**'s GNT# during clock cell eight. Master **A** had kept its REQ# line asserted because it wanted to use the bus for another transaction. Master **A** now samples IRDY# and FRAME# on the rising-edge of each clock until the bus is sensed idle. At that time, it can begin its next transaction.
13. During clock cell eight, master **B** deasserts FRAME#, indicating that its first (and only) data phase is in progress. It also begins to drive the write data onto the AD bus and the appropriate setting onto the Command/Byte Enable bus during clock cell. It asserts IRDY# to indicate to the target that the data is present on the AD bus.
14. At the rising-edge of clock nine, IRDY# and TRDY# are sampled asserted and the data transfer takes place.
15. The initiator, master **B**, then deasserts IRDY# (during clock cell nine) to return the bus to the idle state.
16. Master **A** samples the bus idle and its GNT# asserted at the rising-edge of clock ten and initiates its second transaction during clock cell ten. It also deasserts its REQ# when its asserts FRAME#, indicating to the arbiter that it does not require the bus again upon completion of this transaction.

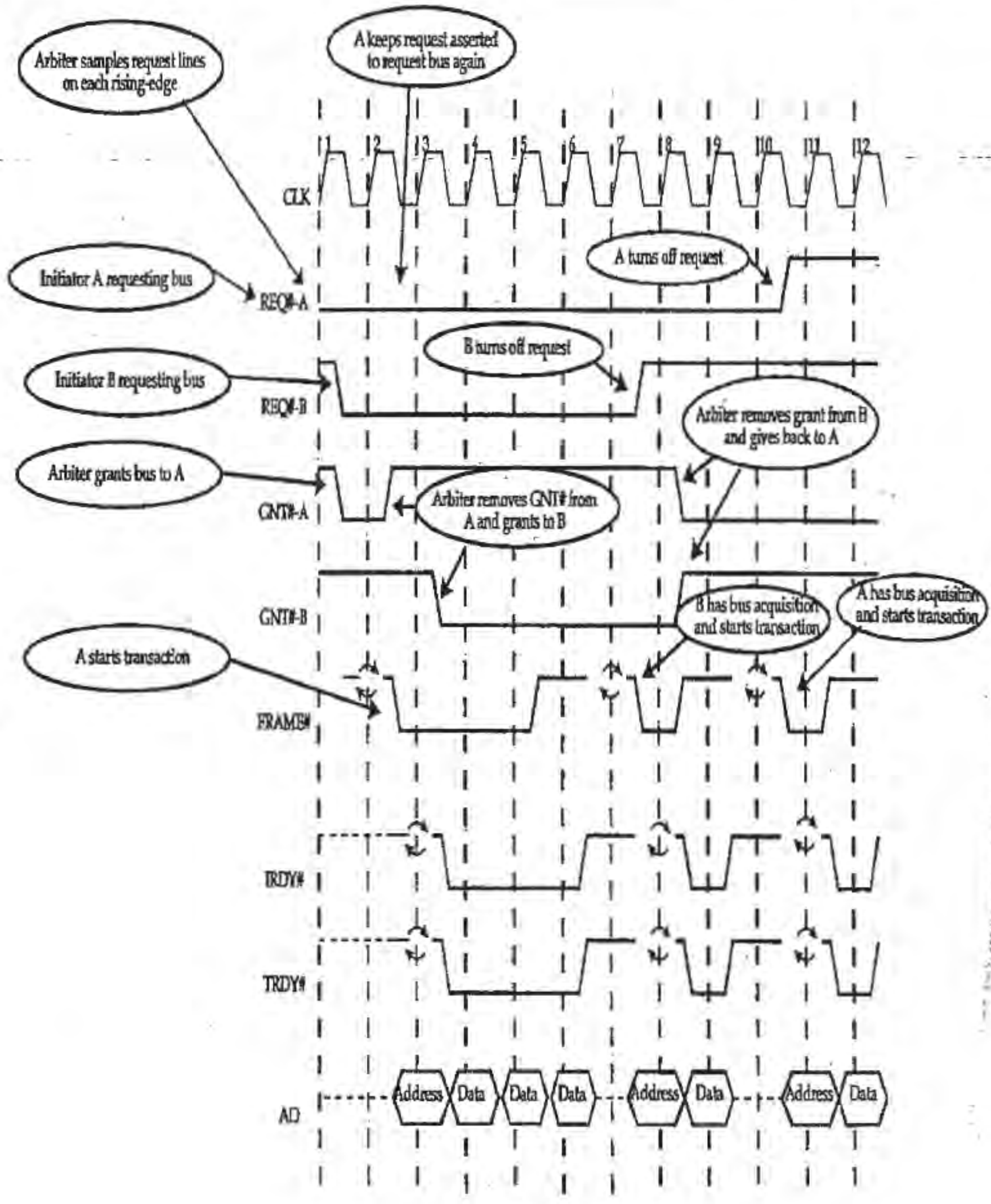
Arbiter samples request lines on each rising-edge
A keeps request asserted to request bus again
CLK
1
2
3
4
5
6
7
8
9
10
11
12
Initiator A requesting bus
REQ#-A
A turns off request
B turns off request
Initiator B requesting bus
REQ#-B
Arbiter removes grant from B and gives back to A
Arbiter grants bus to A
GNT#-A
Arbiter removes GNT# from A and grants to B
GNT#-B
B has bus acquisition and starts transaction
A has bus acquisition and starts transaction
A starts transaction
FRAME#
IRDY#
TRDY#
AD
Address
Data
Data
Data
Address
Data
Address
Data

Figure 6-3. PCI Bus Arbitration Between Two Masters

Bus Access Latency

When a bus master wishes to transfer a block of data between itself and a target PCI device, it must request the use of the bus from the bus arbiter. Bus access latency is defined as the amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction. Figure 6-4 illustrates the different components of the access latency experienced by a PCI bus master. Table 6-2 describes each latency component.

Table 6-2. Access Latency Components

Component	Description
Bus Access Latency	Defined as the amount of time that expires from the moment a bus master requests the use of the PCI bus until it completes the first data transfer of the transaction. In other words, it is the sum of arbitration, bus acquisition and target latency.
Arbitration Latency	Defined as the period of time from the bus master's assertion of REQ# until the bus arbiter asserts the bus master's GNT#. This period is a function of the arbitration algorithm, the master's priority and whether any other masters are requesting access to the bus.
Bus Acquisition Latency	Defined as the period time from the reception of GNT# by the requesting bus master until the current bus master surrenders the bus. The requesting bus master can then initiate its transaction by asserting FRAME#. The duration of this period is a function of how long the current bus master's transaction-in-progress takes to complete. This parameter is the larger of either the current master's LT value (in other words, its timeslice) or the longest latency to first data phase completion in the system.
Target Latency	Defined as the period of time from the start of a transaction until the currently-addressed target is ready to complete the first data transfer of the transaction. This period is a function of the access time for the currently-addressed target device.

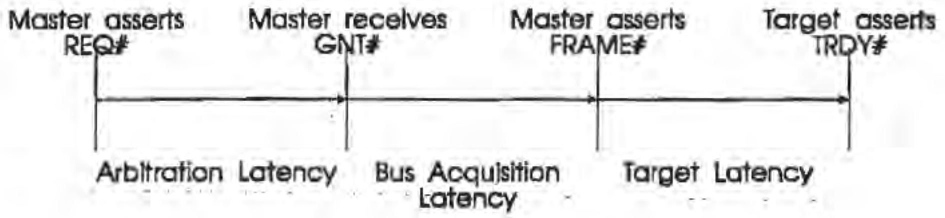

Figure 6-4. Access Latency Components

PCI bus masters should always use burst transfers to transfer blocks of data between themselves and a target PCI device (some poorly-designed masters use a series of single-data phase transactions to transfer a block of data). The transfer may consist of anywhere from one to an unlimited number of bytes. A bus master that has requested and has been granted the use of the bus (its GNT# is asserted by the arbiter) cannot begin a transaction until the current bus master completes its transaction-in-progress. If the current master were permitted to own the bus until its entire transfer were completed, it would be possible for the current bus master to lock out other bus masters from using the bus for extended periods of time. The extensive delay incurred could cause other bus masters (and/or the application programs they serve) to experience poor performance or even to malfunction (buffer overflows or starvation may be experienced).

As an example, a bus master could have a buffer full condition and is requesting the use of the bus in order to off-load its buffer contents to system memory. If it experiences an extended delay (latency) in acquiring the bus to begin the transfer, it may experience a data overrun condition as it receives more data from its associated device (such as a network) to be placed into its buffer.

In order to insure that the designers of bus masters are dealing with a predictable and manageable amount of bus latency, the PCI specification defines two mechanisms:

- Master Latency Timer.
- Target-Initiated Termination.

Master Latency Timer: Prevents Master From Monopolizing Bus

Location and Purpose of Master Latency Timer

The master latency timer, or LT, is implemented as a PCI configuration register in the bus master's configuration space. It is either initialized by the configuration software at startup time, or contains a hardwired value. The value contained in the LT defines the minimum amount of time (in PCI clock periods) that the bus master is permitted to retain ownership of the bus whenever it acquires bus ownership and initiates a transaction.

How LT Works

When the bus master detects bus idle (FRAME# and IRDY# deasserted) and its GNT# asserted, it has bus acquisition and may initiate a transaction. Upon initiation of the transaction, the master's LT is initialized to the value written to the LT by the configuration software at startup time (or its hardwired value). Starting on the next rising-edge of the PCI clock and on every subsequent rising-edge, the master decrements its LT by one.

If the master is in the midst of a burst transaction and the arbiter removes its GNT#, this indicates that the arbiter has detected a request from another master and is granting ownership of the bus for the next transaction to the other master. In other words, the current master has been preempted.

If the current master's LT has not yet been exhausted (decremented all the way down), it has not yet used up its timeslice and may retain ownership of the bus until either:

- it completes its burst transaction or
- its LT expires,

whichever comes first. If it is able to complete its burst before expiration of its LT, the other master that has its GNT# may assume bus ownership when it detects that the current master has returned the bus to the idle state. If the current master is not able to complete its burst transfer before expiration of its LT, it is permitted to complete one more data transfer and must then yield the bus.

If the current master has exhausted its LT, still has its GNT# and has not yet completed it burst transfer, it may retain ownership of the bus and continue to burst data until either:

- it completes its overall burst transfer or
- its GNT# is removed by the arbiter.

In the latter case, the current master is permitted to complete one more data transfer and must then yield the bus.

It should be noted that, when forced to prematurely terminate a data transfer, the bus master must "remember" where it was in the transfer. After a brief period, it may then reassert its REQ# to request bus ownership again so that it may continue where it left off. This topic is covered in the chapter entitled "Premature Transaction Termination."

Is Implementation of LT Register Mandatory?

Yes. It must be implemented as a read/writable register by any master that performs more than two data phases per transaction.

Can LT Value Be Hardwired (read-only)?

Yes, for a master that performs one or two data phases per transaction, but the hardwired value may not exceed 16.

How Does Configuration Software Determine Timeslice To Be Allocated To Master?

The bus master designer implements a read-only register referred to as the minimum grant (Min_Gnt) register. This register is found in the bus master's configuration space. A value of zero indicates that the bus master has no specific requirements regarding the setting assigned to its LT. A non-zero value indicates, in increments of 250ns, how long a timeslice the master requires in order to achieve adequate performance. The value hardwired into this register by the bus master designer assumes a bus speed of 33MHz.

Treatment of Memory Write and Invalidate Command

Any master performing a memory write and invalidate command (see the chapters entitled "The Commands" and "PCI Cache Support") should not terminate its transfer until it reaches a cache line boundary (even if its LT has expired and it has been preempted) unless STOP# is asserted by the target. If it reaches a cache line boundary with its LT expired and its GNT# has been removed by the arbiter, the initiator *must* terminate the transaction. If a memory write and invalidate command is terminated by the target (STOP# asserted by a non-cacheable memory target), the master should complete the line update in memory using the memory write command as soon as it can. Cacheable memory targets must not disconnect a memory write and invalidate command except at cache line boundaries, even if caching is currently disabled. For this reason, the snooper (i.e., the host/PCI bridge) can always assume that the memory write and invalidate command will complete without disconnection if the access is within a memory range designated as cacheable.

Limit on Master's Latency

Is a rule that the initiator may not keep IRDY# deasserted for more than eight PCI clocks during any data phase. If the initiator has no buffer space available to store read data, it must delay requesting the bus until is has room for the data. On a write transaction, the initiator must have the data available before it asks for the bus.

Preventing Target From Monopolizing Bus

General

The problem of a bus master hogging the bus is solved by:

1. The inclusion of the LT associated with each master.
2. The rule that requires the initiator to keep IRDY# deasserted for no longer than eight PCI clocks during any data phase.

It is also possible, however, for a target with a very slow access time to monopolize the bus while a data item is being transferred between itself and the current master. The target currently being addressed does not allow the transfer of a data item to complete until it is ready. This is accomplished by holding off assertion of the target ready signal, TRDY#, until the addressed device is ready to complete the transfer of the data item.

This problem is addressed in the PCI specification by requiring slow targets to terminate a transfer prematurely if it will tie up the bus for long periods. There are three possible cases:

1. If the **time to complete** the **first data phase** will be **greater than 16 PCI CLKs** (from the assertion of FRAME#), the target **must** (the revision 2.0 specification used the word "should" rather than "must") immediately issue a **retry** to the master. This rule applies to all new devices. There are only two exceptions: memory read performed at startup time to copy an expansion ROM image into RAM; and configuration accesses during startup (configuration accesses performed after startup must adhere to the 16 PCI clock limit). A host/PCI bridge that is snooping is permitted to exceed the 16 clock limit, but may never exceed 32 clocks. An example would be a target with an empty buffer that must access a slow device to get the requested data. This forces the master to terminate the transaction with no data transferred, thus freeing up the bus for other masters to use. After two PCI clocks have elapsed, the master that received the retry can reassert its request and, when it receives its GNT#, reinitiate its transaction again. The start address it issues is the address of the data item that was retried.
2. If the target ascertains that it will take it **more than eight PCI clocks to complete the current data phase** (this is referred to as the incremental latency timeout) and it is not the final data phase (FRAME# is still asserted), the target issues a **disconnect** to the master when it is ready to transfer the current data item. The master terminates the transaction when the current data item is transferred and "remembers" the point of disconnection. After two PCI clocks have elapsed, the master that received the disconnect can reassert its request and, when it receives its GNT#, reinitiate its transaction again. The start address it issues is the address of the data item after the one that the disconnect was detected on earlier.
3. If an attempt to communicate with a target results in a **collision on a busy resource** (e.g., a PCI master is attempting a data transfer with an EISA target, but the EISA bridge recognizes that an EISA master currently owns the EISA bus), the target should immediately issue a **retry** to the master. This forces the master to terminate the transaction with no data transferred, thus freeing up the PCI bus for other masters to use. After two PCI clocks have elapsed, the master that received the retry can reassert its request and reinitiate its transaction again. The start address it issues is the address of the data item that was retried.

For further information on termination and re-initiation, refer to the chapter entitled "Premature Transaction Termination."

It should be noted that the incremental latency timeout, or target-initiated termination, is completely independent of the master's LT. The target has no visibility to the master's LT (and visa versa) and therefore cannot tell whether it has timed out or not. This means that slow access targets (greater than eight clocks from the start of one data transfer to the start of the next) always (before or after LT expiration) disconnect from the master after each slow access, thereby fragmenting the overall burst transaction into a series of single data phase transactions. Two examples of devices that might perform disconnects are:

- Targets that are very slow all of the time (virtually all ISA bus devices would fall into this category).
- A target that exhibits very slow access sometimes (perhaps because of a buffer full condition or the need for mechanical movement) and would therefore tie up the PCI bus.

Target Latency on First Data Phase

The following rule was stated earlier: If the time to complete the first data phase will be greater than 16 PCI CLKs, the target must (the revision 2.0 specification used the word "should" rather than "must") immediately issue a retry to the master. This rule applies to all new devices.

A master cannot depend on targets responding to the first data phase within 16 clocks because this rule only affects new devices. Target devices designed prior to the revision 2.1 specification can take longer than 16 clocks to respond.

Options for Achieving Maximum 16 Clock Latency

The target can use any of the following three methods to meet the 16 clock requirement:

1. The simplest case is one where the target can always respond within 16 clocks. No special action is necessary.
2. In the second case, a target may occasionally not be able to meet the 16 clock limit due to a busy resource (e.g., a video frame buffer is being refreshed). In this case, the target simply issues a retry to the master.