PCI System Architecture

Third Edition

MINDSHARE, INC.

TOM SHANLEY
AND
DON ANDERSON

Addison-Wesley Publishing Company
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan
Paris • Seoul • Milan • Mexico City • Taipei

Library of Congress Cataloging-in-Publication Data

ISBN: 0-201-40993-3

Sponsoring Editor: Keith Wollman
Project Manager: Eleanor McCarthy
Production Coordinator: Lora L. Ryan
Cover design: Barbara T. Atkinson
Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9 -MA- 9998979695
First printing, February 1995

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

To Nancy and Sheryl, two very understanding ladies.

Contents

Part II: Revision 2.1 Essentials

CHAPTER 3: Intro to PCI Bus Operation

CHAPTER 4: Intro to Reflected-Wave Switching

CHAPTER 5: The Functional Signal Groups

CHAPTER 6: PCI Bus Arbitration

CHAPTER 7: The Commands

CHAPTER 8: The Read and Write Transfers

CHAPTER 9: Premature Transaction Termination

CHAPTER 10: Error Detection and Handling

CHAPTER 11: Interrupt-Related Issues

CHAPTER 12: Shared Resource Acquisition

CHAPTER 13: The 64-bit PCI Extension

CHAPTER 14: Add-in Cards and Connectors

Part III: Device Configuration In System With a Single PCI Bus

CHAPTER 15: Intro to Configuration Address Space

CHAPTER 16: Configuration Transactions

CHAPTER 17: Configuration Registers

CHAPTER 18: Expansion ROMs

Part IV: PCI-to-PCI Bridge

CHAPTER 19: PCI-to-PCI Bridge

Part V: The PCI BIOS

CHAPTER 20: The PCI BIOS

Part VI: PCI Cache Support

CHAPTER 21: PCI Cache Support

Part VII: 66MHz PCI Implementation

CHAPTER 22: 66MHz PCI Implementation

Part VIII: Overview of VLSI Technology VL82C59x SuperCore PCI Chipset

CHAPTER 23: Overview of VLSI Technology VL82C59x SuperCore PCI Chipset

Appendices

Figures

Tables

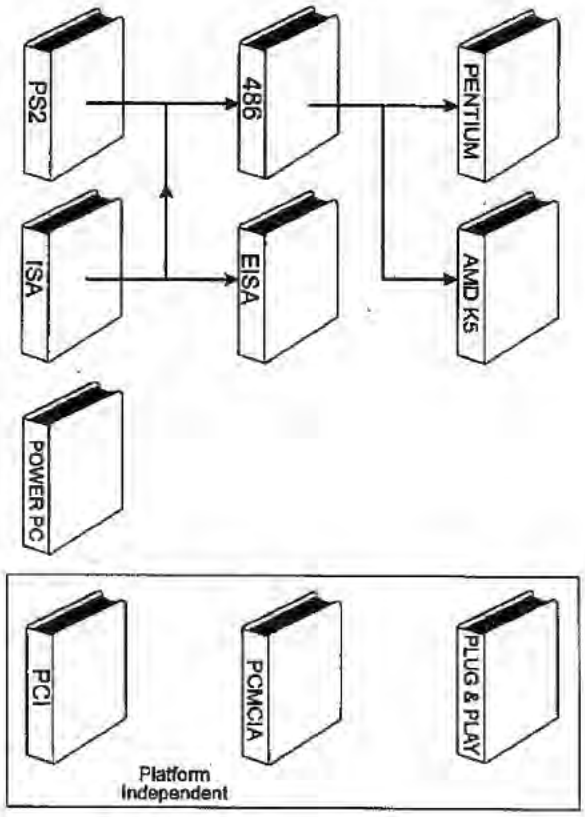
Acknowledgments

To John Swindle for his tireless attention to detail and his marvelous teachin ability. To the editorial staff at Addison-Wesley for their patience. To the folk at Computer Literacy Bookshops for their collective support in the initia launch of this book series. And finally, to the many hundreds of engineers a Intel, IBM, Compaq, Dell, Hewlett-Packard, Motorola, and other clients, wh subject themselves to our teaching on a regular basis.

The MindShare Architecture Series

The MindShare Architecture book series includes: *ISA System Architecture, EISA System Architecture, 80486 System Architecture, PCI System Architecture, Pentium System Architecture, PCMCIA System Architecture, PowerPC System Architecture, Plug-and-Play System Architecture,* and *AMD K5 System Architecture.*

Rather than duplicating common information in each book, the series uses the building-block approach. *ISA System Architecture* is the core book upon which the others build. The figure below illustrates the relationship of the books to each other.



Series Organization

Organization of This Book

The third edition of *PCI System Architecture* has been updated to reflect revision 2.1 of the PCI bus specification. In addition, it has been completely reorganized and expanded to include more detailed discussions of virtually every topic found in the first two editions. The book is divided into eight parts:

- Part I: Intro to the Local Bus Concept. Defines the performance problems inherent in PC architecture before the introduction of the local bus. Having defined the problem, the possible solutions are explored.
- Part II: Revision 2.1 PCI Essentials. This part of the book provides a detailed explanation of the mainstream aspects of PCI bus operation.
- Part III: Device Configuration In a System With a Single PCI Bus. Provides an introduction to the PCI configuration address space, a detailed description of the methods for generating configuration bus transactions, the configuration read and write transactions timing, the configuration registers defined by the specification, and the implementation of expansion ROMs associated with PCI devices.
- Part IV: The PCI-to-PCI Bridge. This part provides a detailed discussion of the PCI-to-PCI Bridge specification, a discussion of peer and hierarchical PCI buses, and the accessing of configuration registers in devices residing on subordinate PCI buses.
- Part V: The PCI BIOS. This part provides a detailed discussion of the PCI BIOS specification.
- Part VI: Support for Cacheable PCI Memory.
- Part VII: 66MHz PCI Implementation.
- Part VIII: Overview of VLSI VL82C59x Supercore Chipset. This part provides an operational overview of the VLSI chip set.

Who this Book is For

This book is intended for use by hardware and software design and support personnel. Due to the clear, concise explanatory methods used to describe each subject, personnel outside of the design field may also find the text useful.

Prerequisite Knowledge

It is highly recommended that the reader have a good knowledge of PC and processor bus architecture prior to reading this book. The MindShare publications entitled *ISA System Architecture* and *80486 System Architecture* provide all of the background necessary for a complete understanding of the subject matter covered in this book. Alternately, the reader may substitute *Pentium System Architecture* or *PowerPC System Architecture* for *80486 System Architecture*.

Object Size Designations

The following designations are used throughout this book when referring to the size of data objects:

- A **byte** is an 8-bit object.
- A **word** is a 16-bit, or two byte, object.
- A **doubleword** is a 32-bit or four byte, object.
- A **quadword** is a 64-bit, or eight byte, object.
- A **paragraph** is a 128-bit, or 16 byte, object.
- A **page** is a 4K-aligned 4KB area of address space.

Documentation Conventions

This section defines the typographical convention used throughout this book.

Hex Notation

All hex numbers are followed by an "h." Examples:

9A4Eh
0100h

Binary Notation

All binary numbers are followed by a "b." Examples:

0001 0101b
01b

Decimal Notation

Numbers without any suffix are decimal. When required for clarity, decimal numbers are followed by a "d." The following examples each represent a decimal number:

16
255
256d
128d

Signal Name Representation

Each signal that assumes the logic low state when asserted is followed by a pound sign (#). As an example, the TRDY# signal is asserted low when the target is ready to complete a data transfer.

Signals that are not followed by a pound sign are asserted when they assume the logic high state. As an example, IDSEL is asserted high to indicate that a PCI device's configuration space is being addressed.

Identification of Bit Fields (logical groups of bits or signals)

All bit fields are designated in little-endian bit ordering as follows:

[X:Y],

where "X" is the most-significant bit and "Y" is the least-significant bit of the field. As an example, the PCI address/data bus consists of AD[31:0], where AD31 is the most-significant and AD0 the least-significant bit of the field.

We Want Your Feedback

MindShare values your comments and suggestions. You can contact us via mail, phone, fax or internet email.

Phone: (214) 231-2216
Fax: (214) 783-4715
E-mail: mindshar@interserv.com

To request information on public or private seminars, email your request to: mindshar@interserv.com or call our bulletin board at (214) 705-9604.

Bulletin Board

Because we are constantly on the road teaching, we can be difficult to get hold of. To help alleviate problems associated with our migratory habits, we have initiated a bulletin board to supply the following services:

- Download of course abstracts.
- Download of tables of contents of each book in the series.
- Facility to inquire about public architecture seminars.
- Message area to log technical questions.
- Message area to log suggestions for book improvements.
- Facility to view book errata and clarifications.

The bulletin board may be reached 24-hours a day, seven days a week.

BBS phone number: (214) 705-9604

Mailing Address

MindShare, Inc.
2202 Buttercup Drive
Richardson, Texas 75082

Part I

Introduction to the Local Bus Concept

Chapter 1

In This Chapter

This chapter defines the performance constraints experienced when devices that perform block data transfers are placed on the expansion bus (e.g., the ISA, EISA and Micro Channel buses). It also uses the performance requirements of teleconferencing to highlight the bandwidth requirements of systems requiring fast block transfers between multiple subsystems in order to achieve superior system performance.

The Next Chapter

The next chapter introduces the concept of the local bus. The VESA VL bus and the PCI bus implementations of the local bus are introduced as solutions to the throughput problem.

Block-Oriented Devices

In today's operating environments, it is imperative that large block data transfers be accomplished expeditiously. This is especially true in relation to the following types of subsystems:

- Graphics video adapter.
- Full-motion video adapter.
- SCSI host bus adapter.
- FDDI network adapter.

Graphics Interface Performance Requirements

The Windows, OS/2 and Unix X-Windows user interfaces require extremely fast updates of the graphics image in order to move, resize and update multiple windows without imposing discernible delays on the end-user. Since the

screen image is stored in video RAM, this means that the processor must be able to update and/or move large blocks of data within video memory very fast. The same is true for the updating of full-motion video in video ram.

SCSI Performance Requirements

The SCSI interface is used to move large blocks of data between target I/O devices and system memory. Mass storage devices such as hard disk drives, CD-ROM drives and tape backup subsystems typically reside on the SCSI bus. The time required to read or write files on hard drives or tape, or to read files from CD-ROM can impose delays on the end-user. Anything that can be done to speed up these block data transfers has a significant effect on overall system performance.

Network Adapter Performance Requirements

When a network adapter is used to transfer entire files of information to or from a server (a print or file server), the rate at which the information can be transferred between system memory and the network adapter detracts from or contributes to overall system performance.

X-Bus Device Performance Constraints

The devices just described are just some examples of subsystems that benefit significantly from a fast transfer rate. Unfortunately, the majority of subsystems reside on the PC's expansion bus. Depending on the machine's design, this may be the ISA, EISA or Micro Channel expansion bus. As described later in this chapter, all three of these expansion bus architectures suffer from an inadequate data transfer rate.

In many cases, subsystems such as the graphics video adapter have been integrated onto the system board. This would seem to imply that they do not reside on the expansion bus, but this is not the case. Most of the integrated subsystems reside on a buffered version of the expansion bus known as the X-bus (eXtension to the expansion bus; also referred to as the utility bus). This being the case, these subsystems are bound by the mediocre transfer rates achievable when communicating with devices residing on the expansion bus. Figure 1-1 illustrates the relationship of the X-bus to the expansion bus and the system board microprocessor.

When performing memory reads, the microprocessor can communicate with its internal (level one, or L1) cache at its full native speed if the requested information is found in the cache. If the cache is implemented as a write-back cache, memory writes to currently-cached locations may also be completed at full speed. When an L1 cache miss occurs on a memory read or the cache must write information into memory, the processor must use its local bus to communicate with memory. The memory access request is first submitted to the external, level 2 (L2) cache for fulfillment. In the event of an L2 cache miss, the L2 cache performs an access to system DRAM memory. The linkage between the L2 cache and system DRAM memory is typically optimized to allow information transfers to complete as quickly as possible.

When a memory read or write addresses memory other than system DRAM memory or when the processor is performing an I/O read or write, the expansion bus bridge must pass the bus cycle through to the expansion bus. The completion of the bus cycle is bound by the maximum expansion bus speed and the access time of the expansion bus device being accessed. If a large amount of data is to be transferred to or from the target expansion device, performance is bound by the speed of the bus, the access time of the target, and the expansion bus data bus width.

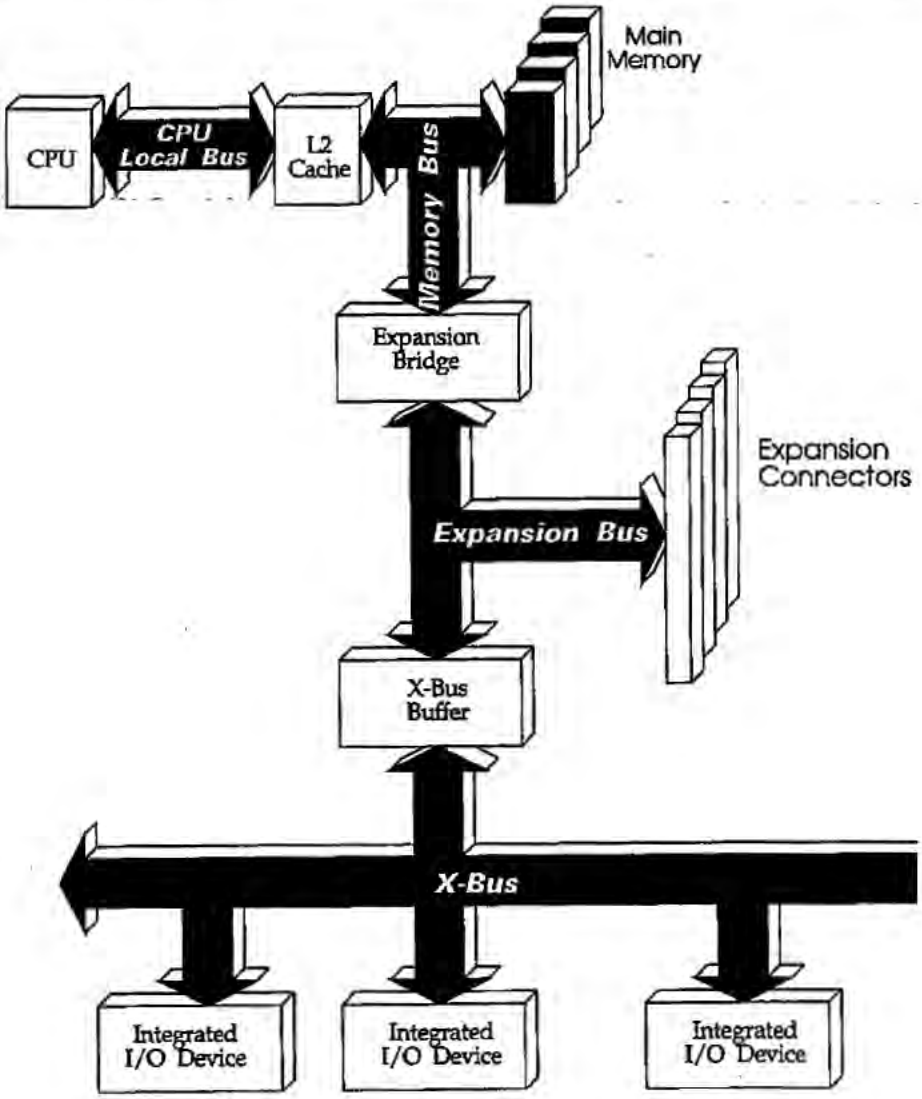


Figure 1-1. The X-Bus

Expansion Bus Transfer Rate Limitations

ISA Expansion Bus

All transfers performed over the ISA bus are synchronized to an 8MHz (more typically, 8.33MHz) bus clock signal (BCLK). It takes a minimum of two cycles of the bus clock (if the target device is a zero wait state device) to perform a data transfer. This equates to 4.165 million transfers per second. Since the data path on the ISA bus is only 16-bits wide, a maximum of two bytes may be transferred during each transaction. This equates to a theoretical maximum transfer rate of 8.33 MBytes per second.

For more information on the ISA expansion bus, refer to the Addison-Wesley publication entitled *ISA System Architecture*.

EISA Expansion Bus

Like the ISA bus, all transfers performed over the EISA bus are synchronized to an 8MHz (more typically, 8.33MHz) bus clock signal (BCLK). It takes a minimum of one cycle of the bus clock (if the target device supports EISA burst mode transfers) to perform a data transfer. This equates to 8.33 million transfers per second. Since the data path on the EISA bus is 32-bits wide, a maximum of four bytes may be transferred during each transaction. This equates to a theoretical maximum transfer rate of 33 Mbytes per second.

For more information on the EISA expansion bus, refer to the Addison-Wesley publication entitled *EISA System Architecture*.

Micro Channel Architecture Expansion Bus

At the current time, the maximum achievable transfer rate on the Micro Channel (as implemented in the PS/2 product line) is 40Mbytes per second (using the 32-bit Streaming Data Procedure). This is based on a 10MHz bus speed with one data transfer taking place during each cycle of the 10MHz clock (10 million transfers per second * four bytes per transfer). Faster transfer rates of 80 and 160Mbytes per second are possible when the 64-bit and enhanced 64-bit Streaming Data Procedures are implemented.

Teleconferencing Performance Requirements

Figure 1-2 illustrates three PCs linked via a telecommunications network. Each of the three units has the capability to simultaneously merge multiple graphics and video sources onto the screen in real-time. Figure 1-3 illustrates the contents of each screen.

The large portion of the screen (devoted to a graphics image) is utilized to display the document under discussion. In order to successfully emulate an actual face-to-face conferencing situation, the system must be capable of updating this image fast enough to simulate flipping through the pages of a document at the rate of ten pages (or frames) per second. With an image resolution of 1280 x 1024 pixels and color resolution of 16 million colors (three bytes per pixel), the amount of video memory required to store one image is 3.93216Mbytes. To alter the graphics display at the rate of ten frames per second would require a video memory update rate of 39.3216Mbytes per second.

The video preview portion of the screen is used to display a real-time video image of a video source local to the unit. This image has a resolution of 320 x 240 pixels and a color resolution of 256 colors (one byte per pixel). In order to provide full-motion video, the image must be updated at the rate of thirty frames per second. The amount of video memory required to store one image would be 76.8Kbytes. To alter the graphics display at the rate of thirty frames per second would require a video memory update rate of 2.3Mbytes per second.

Each of the two video remote screen areas is used to display a full-motion video image from one of the other two participants. These images each have a resolution of 640 x 480 pixels and a color resolution of 256 colors (one byte per pixel). In order to provide full-motion video, each image must be updated at the rate of thirty frames per second. The amount of video memory required to store one image would be 307.2Kbytes. To alter each of the video remote windows at the rate of thirty frames per second would require a video memory update rate of 9.2Mbytes per second.

Each of the three video images would be transmitted in compressed video image format at the rate of 200Kbytes per second per video stream.

In summary, each host system must supply sufficient bus bandwidth to support the combined transfer rates required to update the images presented in the graphics, preview, remote one and remote two windows, as well as the

three 200Kbyte per second compressed video streams. The bus structure must then support the simultaneous transfer rates listed in table 1-1. ISA (8.33Mbytes per second) and the current version of EISA (33Mbytes per second) will not support the combined bandwidth requirement of 60.516Mbytes per second. The Micro Channel (40Mbytes per second) does not currently support the required rate, but the 64-bit Streaming Data Procedures (not supported on the PS/2 product line) are able to achieve transfer rates of 80 to 160Mbytes per second. As described later in this document, the PCI bus currently supports a transfer rate of 132Mbytes per second. If the 64-bit PCI extension is implemented, a transfer rate of 264Mbytes per second can be achieved. Table 1-2 lists the transfer rates for the video and other subsystems.

Table 1-1. Teleconferencing Transfer Rate Requirements

Screen Element	Transfer Rate (Mbytes/second)
Graphics Window	39.216
Preview Window	2.3
Video Remote One	9.2
Video Remote Two	9.2
Preview Compressed Video Stream	.2
Video Remote One Compressed Video Stream	.2
Video Remote Two Compressed Video Stream	.2
Total transfer rate required to support teleconferencing	60.516

Table 1-2. Required Subsystem Transfer Rates

Subsystem	(Mbytes per second)
Graphics	30 to 40
Full-Motion Video	2 to 9 per window
LAN	15 for FDDI (Fiber Distributed Data Interface)
	3 for Token Ring
	2 for Ethernet
Hard Disk	5 to 20 using SCSI
CD-ROM	2 using SCSI
Audio	1 for CD quality output

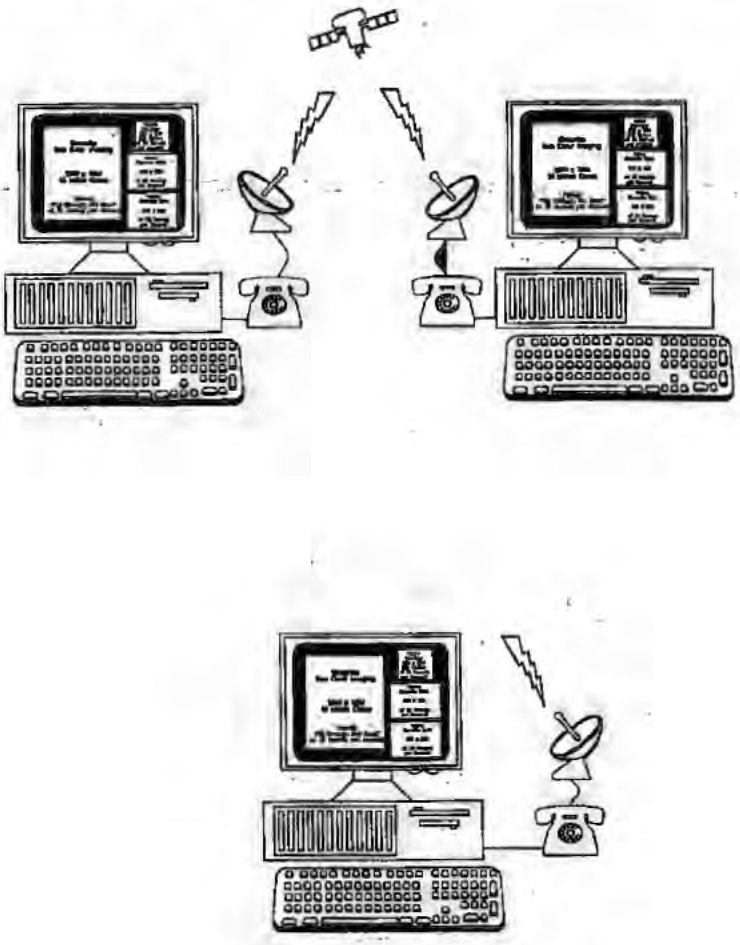


Figure 1-2. The Teleconference

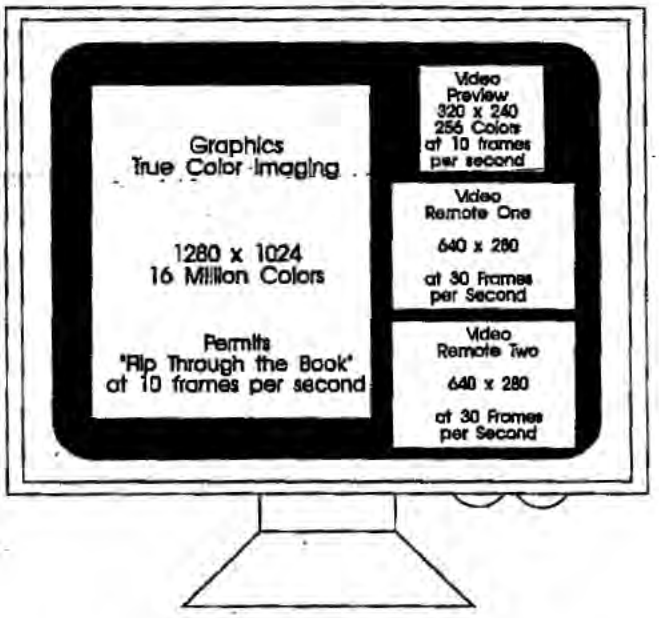


Figure 1-3. The Teleconference Screen Layout

Chapter 2

The Previous Chapter

The previous chapter discussed the performance constraints placed upon subsystems installed on the expansion bus or integrated onto the system board's X-bus.

This Chapter

This chapter introduces the concept of the local bus and provides an overview of the two major local bus standards:

- the VESA VL bus
- the PCI bus

The Next Chapter

The next chapter provides an introduction to the PCI transfer mechanism.

Graphics Accelerators: Before Local Bus

An interim attempt to improve the performance of video graphics adapters implemented as expansion bus devices involved the enhancement of the adapter's intelligence. Earlier adapters processed very low-level commands issued by the microprocessor. The processor and therefore the programmer had to be intimately involved in every aspect of screen management. Later adapters are frequently based on processors like the Intel i860XR/XP or the Texas Instruments TMS34010/34020 and can handle high-level commands to offload screen-intensive operations from the microprocessor. As an example, a BITBLT command can be issued to the adapter, causing it to quickly move a window graphic from one area of video memory to another without any further intervention on the microprocessor's part. The video memory is on the expansion adapter card and can therefore be accessed directly by the adapter's local processor at high speed.

Local Bus Concept

To maximize throughput when performing updates to video graphics memory, many PC vendors have moved the video graphics adapter from the slow expansion bus to the processor's local bus. Figure 2-1 illustrates the processor's local bus. The video adapter is redesigned to connect directly to the processor's local bus and the adapter design is optimized to minimize or eliminate the number of wait states inserted into each bus cycle when the processor accesses video memory and the adapter's I/O registers. In addition, the video graphics adapter typically also incorporates a local processor and can handle high-level commands (as discussed earlier).

Direct-Connect Approach

There are three basic methods for connecting a device to the microprocessor's local bus. The first scenario is pictured in figure 2-1 and is very straightforward: the device is connected directly to the processor's bus structure. This could be any processor type (such as the 486). As an example, when the 486 performs zero wait-state bus cycles at its maximum rated speed of 33MHz (the actual bus speed is processor implementation-dependent), read burst transfers can be performed at the rate of 132Mbytes per second (if the processor is communicating with video memory that supports burst mode and is cacheable). When performing memory writes to update the video frame buffer in memory, the programmer may specify no more than four bytes to be written to memory per bus cycle. If the video memory supports zero wait-state writes, this would permit a data transfer rate of 66Mbytes/second. The direct-connect approach imposes a number of important design constraints:

- Since the device is connected directly to the processor's local bus, it must be redesigned in order to be used with next generation processors (if the bus structure or protocol are altered).
- Due to the extra loading placed on the local bus, no more than one local bus device may be added.
- Because the local bus is running at a high frequency, the design of the local bus device's bus interface is difficult.
- Although the system may work when it's shipped, it may exhibit aberrant behavior when an Intel Overdrive Processor is installed in the upgrade socket (thereby placing another load on the local bus).
- It does not permit the processor to perform transfers with one device while the local bus device is involved in a transfer with another device.

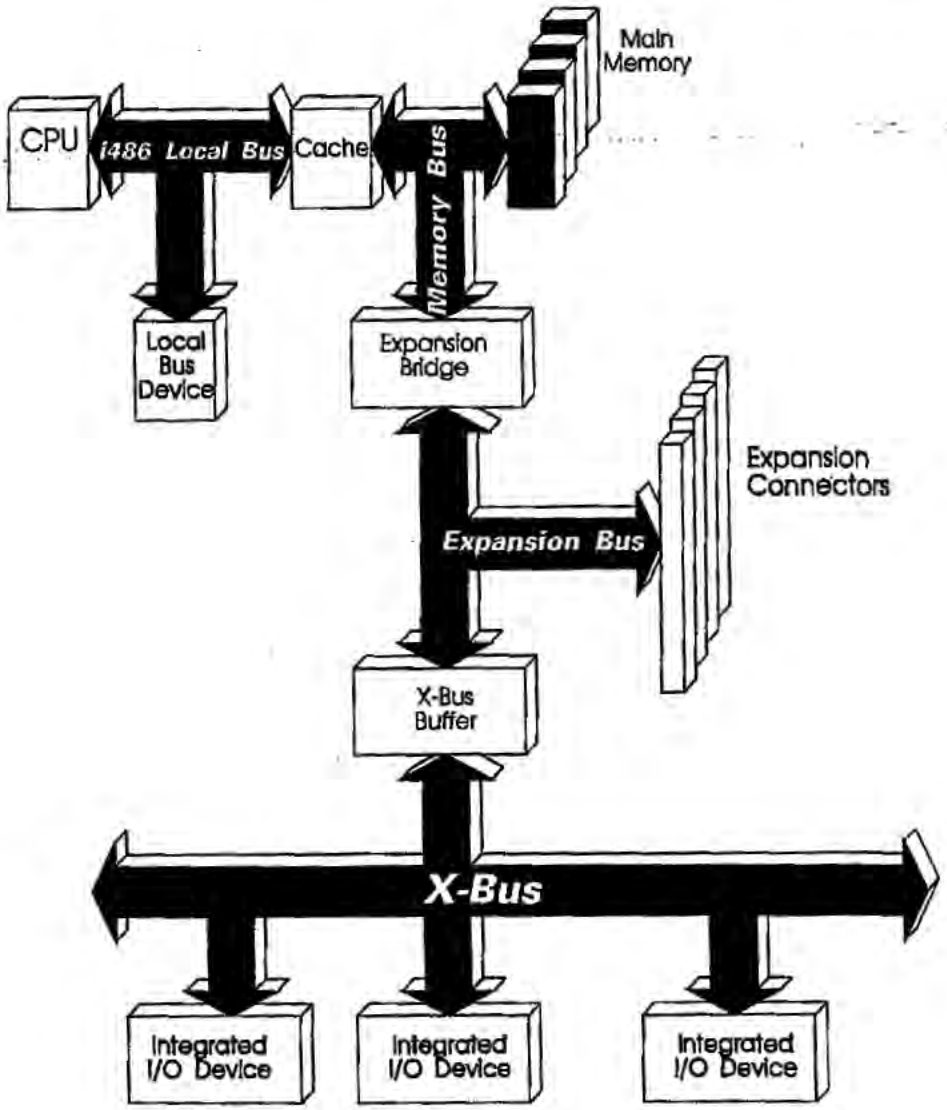


Figure 2-1. The Direct-Connect Local Bus Approach

Buffered Approach

The second approach that can be utilized in connecting a local bus device to the processor's local bus is the buffered approach. Figure 2-2 illustrates this scenario. The buffer/driver redrives all of the local bus signals, thereby permitting fanout to more than one local bus device. Since the buffered local bus is electrically-isolated from the microprocessor's local bus, it only presents one load to the microprocessor's local bus. Typically, a maximum of three local bus devices can be placed on the buffered local bus. This is the only real advantage of this approach over the direct-connect approach.

A major disadvantage of the buffered approach is that the processor's local bus and the buffered local bus are essentially one bus. Any transaction initiated by the processor appears on the local and buffered local buses. Likewise, any bus transaction initiated by a bus master that resides on the buffered local bus appears on both the buffered local bus and the processor's local bus. In other words, either the processor or a local bus master may use the bus, but not both simultaneously. If a local bus master is using the bus and the processor requires the bus to perform a transaction, the processor is stalled until the bus master surrenders ownership of the bus. The reverse situation is also true.

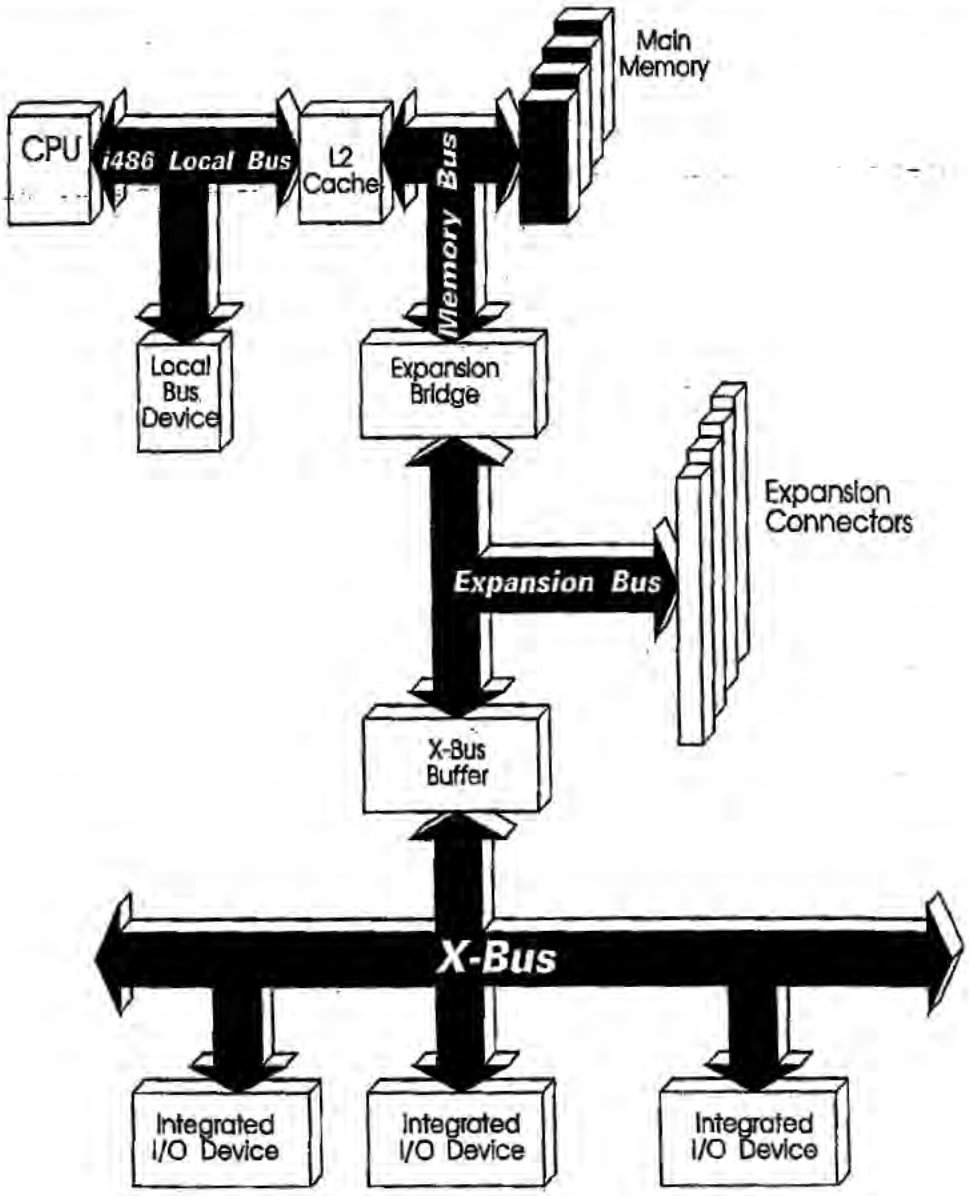


Figure 2-2. The Buffered Local Bus Approach

Workstation Approach

Figure 2-3 illustrates an approach used in many workstation architectures to achieve high performance. The processor's L2 cache controller is combined with a bridge that provides the interface between the processor, main memory and the high-speed I/O bus (in this case, the PCI bus). The devices that reside on the I/O bus may consist only of target devices or a mixture of targets and intelligent peripheral adapters with bus master capability. Via the specially-designed bridge, either the processor (through its L2 cache) or a bus master on the I/O bus (or the expansion bus) can access main memory. Optimally, the processor can continue to fetch information from its L1 or L2 cache while the cache controller provides a bus master on the I/O bus with access to main memory. Bus masters on the I/O bus can also communicate directly with target devices on the I/O bus while the processor is accessing its L1 or L2 cache or while the L2 cache controller is accessing main memory for the processor.

Another very distinct advantage of this approach is that it renders the I/O bus device interface independent of the processor bus. Processor upgrades can be easily implemented without impacting the design of the I/O bus and its associated devices. Only the cache/bridge would require a redesign (to match the new host processor interface).

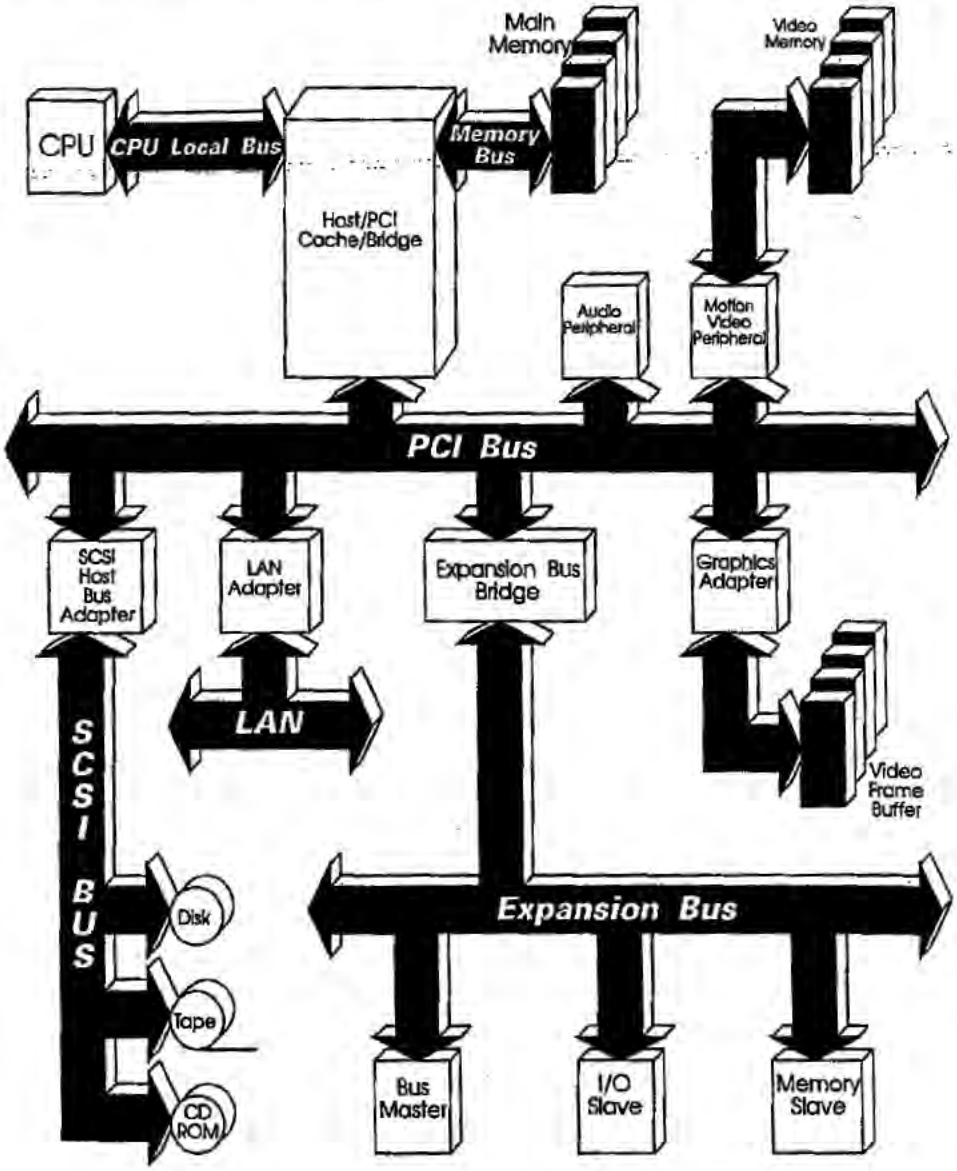


Figure 2-3. The Workstation Approach

A variation on this theme would find the processor's L2 cache implemented as a lookaside cache located on the processor's local bus. In this configuration, main memory may be located either on a dedicated memory bus (as shown in figure 2-3) or it may reside on the processor's local bus along with the lookaside L2 cache. If the main memory is located on the processor's local bus, it should be noted that it can only be accessed by the performance of a memory read or write transaction on the processor's local bus. This is true even if a bus master located on the I/O bus is accessing main memory. This could diminish the processor's performance by diminishing the local bus availability. The reverse would also be true: bus masters other than the host processor cannot access main memory while the processor is utilizing its local bus.

VESA VL Bus Solution

Until several years ago, there existed no standard that defined the interconnection schema used to integrate local bus devices into the PC environment. The Video Electronics Standards Association (VESA), an association of companies involved in the design and manufacturing of video graphics adapters, commissioned the development of a local bus standard. The preliminary specification was completed and refers to the local bus as the VL bus (VESA Local bus). The initial version of the VESA specification, version 1.0, defines two methods of interfacing to the microprocessor's local bus: the direct-connect and the buffered approaches described earlier. The direct-connect approach is referred to as the VL Type "A" bus, while the buffered version is referred to as the VL Type "B" bus. In both cases, the bus is modeled on the 486 bus. Some characteristics of each implementation are listed in table 2-1. A brief description of each of the listed characteristics follows the table.

Table 2-1. VL Bus Characteristics

Characteristics	Type "A"	Type "B"
logic cost	$0.	Cost of buffering.
Performance	At a bus speed of 33MHz, 132Mbytes/second (peak) on burst reads, and 66Mbytes/second on write transfers.	Same as type "A", but the delay imposed by the buffer almost certainly causes wait states to be inserted in each transfer.
Longevity	Tied to 386/486 bus structure.	Tied to 386/486 bus structure.
Teleconferencing Support	One local bus device.	Three local bus devices.
Electrical Integrity	Not defined.	Not defined.
Modularity	None.	Three Micro Channel connectors.
Auto-Configuration	Supports Auto Configuration (see "Auto-Configuration" section later in this chapter).	Supports Auto-Configuration (see "Auto-Configuration" section later in this chapter).

Logic Cost

No additional system board logic is necessary to implement a VL Type "A" local bus device. The device is connected directly to the microprocessor's local bus. In a Type "B" design, the cost of the buffering logic must be taken into account.

Performance

Using the type "A" and "B" approaches, a peak data transfer rate of 132Mbytes per second may be achieved (at a processor bus speed of 33MHz). It should be noted that the longest burst read performed by the 486 processor occurs during a cache line fill operation. Sixteen bytes (four doublewords) are transferred to the processor during the cache line fill. The first doubleword takes two processor clocks, while the subsequent three doublewords may be transferred back to the microprocessor at the rate of one per processor clock cycle (if the access time of the target device supports this speed).

The 486 processor is only capable of performing burst writes under the following circumstances:

- When it attempts to write two to four bytes (in one bus cycle) to an 8-bit device (BS8# is sampled asserted). An 8-bit device that supports burst mode operation can achieve a transfer rate of 33Mbytes/second (one byte transferred during each processor clock cycle), but it should be noted that

this rate can only be sustained for the transfer of up to three successive bytes.

- When it attempts to write two to four bytes (in one bus cycle) to a 16-bit device (BS16# is sampled active). A 16-bit device that supports burst mode operation can achieve a transfer rate of 66Mbytes/second, but it should be noted that this rate can only be sustained for the transfer of one 16-bit object.

When performing 32-bit non-burst write transfers, the 486 microprocessor can achieve a maximum transfer rate of 66Mbytes/second (two processor clock cycles per 32-bit transfer).

It should be noted that the VL bus specification defines bus speeds up to a maximum frequency of 66MHz. All performance estimates quoted in this publication are based on a maximum bus speed of 33MHz because this is the achievable norm at the current time.

Longevity

Both the type "A" and "B" approaches are short term solutions because they are designed around the and 486 processor bus structure. The interface logic must be redesigned for next generation processors with more advanced bus structures. Bridge logic would be necessary to translate between the new processor bus and the VL bus.

Teleconferencing Support

The type "A" approach does not offer a teleconferencing solution because it provides support for only one local bus device. At a bare minimum, teleconferencing requires high-speed support for at least two peripheral subsystems: the graphics and full-motion video adapters. The type "B" solution provides minimal teleconferencing support by supporting up to three local bus peripherals.

Electrical Integrity

The VESA VL 1.0 bus specification provides no electrical design guidelines to ensure the integrity of local bus design. System board designers must design the PCI system board layout from scratch. While this isn't a problem at low

bus speeds, buses running at today's accelerated rates present a formidable design challenge.

Add-in Connectors

Modularity refers to the ability to add new local bus peripherals by installing an option card into a local bus connector. Type "A" solutions are direct-connect and do not provide a connector. Type "B" solutions can support up to three connectors. The VL specification defines a Micro Channel-style connector as the expansion vehicle.

Auto-Configuration

The VESA VL 1.0 specification states that VL local bus devices must support automatic system configuration. However, the specification does not define the standard automatic configuration support that must be provided in each VL bus-compliant local bus device. The specification also states that VL bus-compliant local bus devices must be transparent to device drivers. In other words, they must respond to the same command set and supply the same status as their non-local bus cousins.

The fact that the VL bus specification does not define the location or format of the local bus devices' configuration registers opens the door for a "tower of Babel" scenario regarding the software interface to these devices.

Revision 2.0 VL Specification

The rev 2.0 specification adds support for VESA VL bus masters.

PCI Bus Solution

Intel defined the PCI bus to ensure that the marketplace would not become crowded with various permutations of local bus architectures implemented in a short-sighted fashion. The first release of the specification, version 1.0, became available on 6/22/92. Revision 2.0 became available in April of 1993. The current revision of the specification, 2.1, became available in Q1 of 1995. Intel made the decision not to back the VESA VL standard because the emerging standard did not take a sufficiently long-term approach towards the problems presented at that time and those to be faced in the coming five

years. In addition, the VL bus has very limited support for burst transfers thereby limiting the achievable throughput.

PCI stands for Peripheral Component Interconnect. The PCI bus can be populated with adapters requiring fast accesses to each other and/or system memory and that can be accessed by the host processor at speeds approaching that of the processor's full native bus speed. It is very important to note that all read and write transfers over the PCI bus are burst transfers. The length of the burst is negotiated between the initiator and target devices and may be of any length. ***This is in sharp contrast to the burst capability inherent in the VL bus design.*** Table 2-2 identifies some of PCI's major design goals. The chapters that follow provide a detailed description of the PCI bus and related subjects.

The PCI specification allows system design centered around two of the three approaches discussed earlier: the buffered and workstation approaches. Due to its performance and flexibility advantages, the workstation approach is preferred. Figure 2-4 illustrates the basic relationship of the PCI, expansion processor and memory buses.

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.

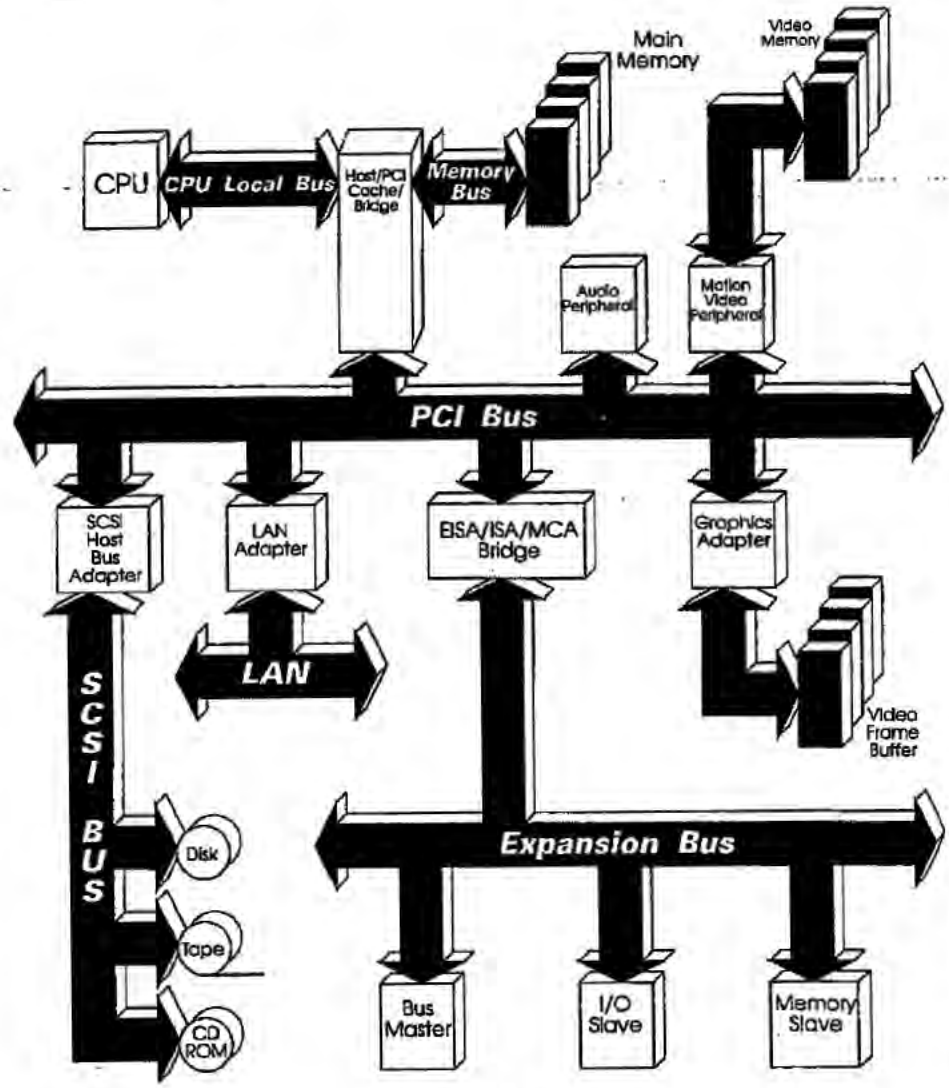


Figure 2-4. The PCI Bus

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.

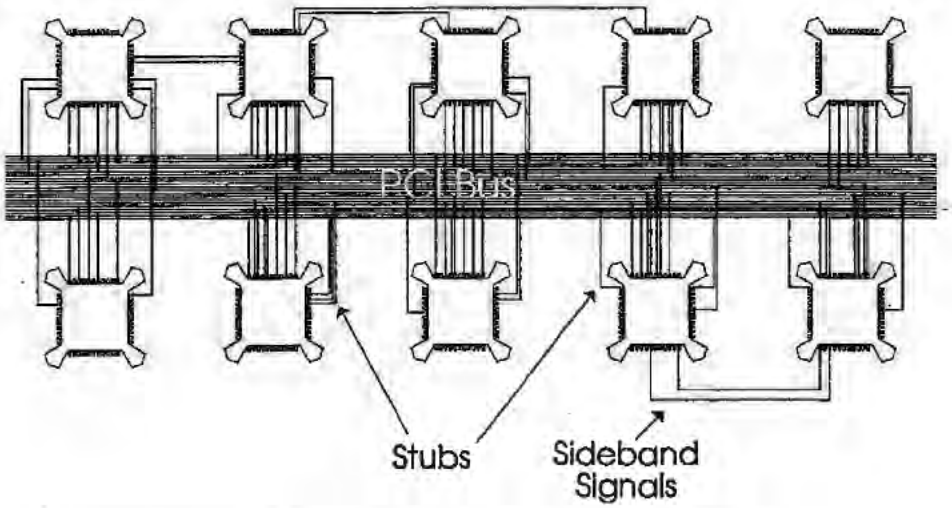


Figure 2-5. PCI Devices Attached to the PCI Bus

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.

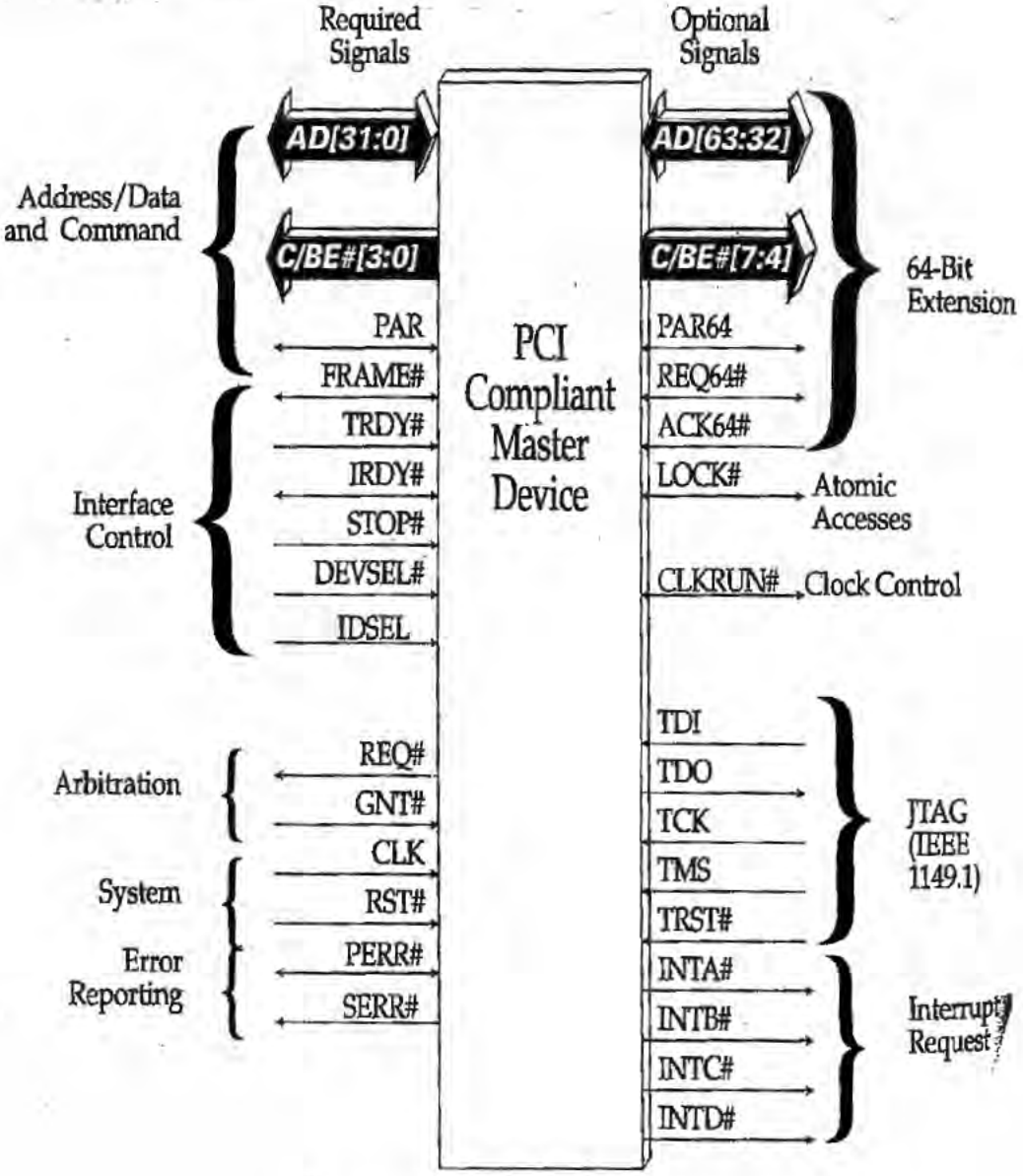


Figure 5-1. PCI-Compliant Master Device Signals

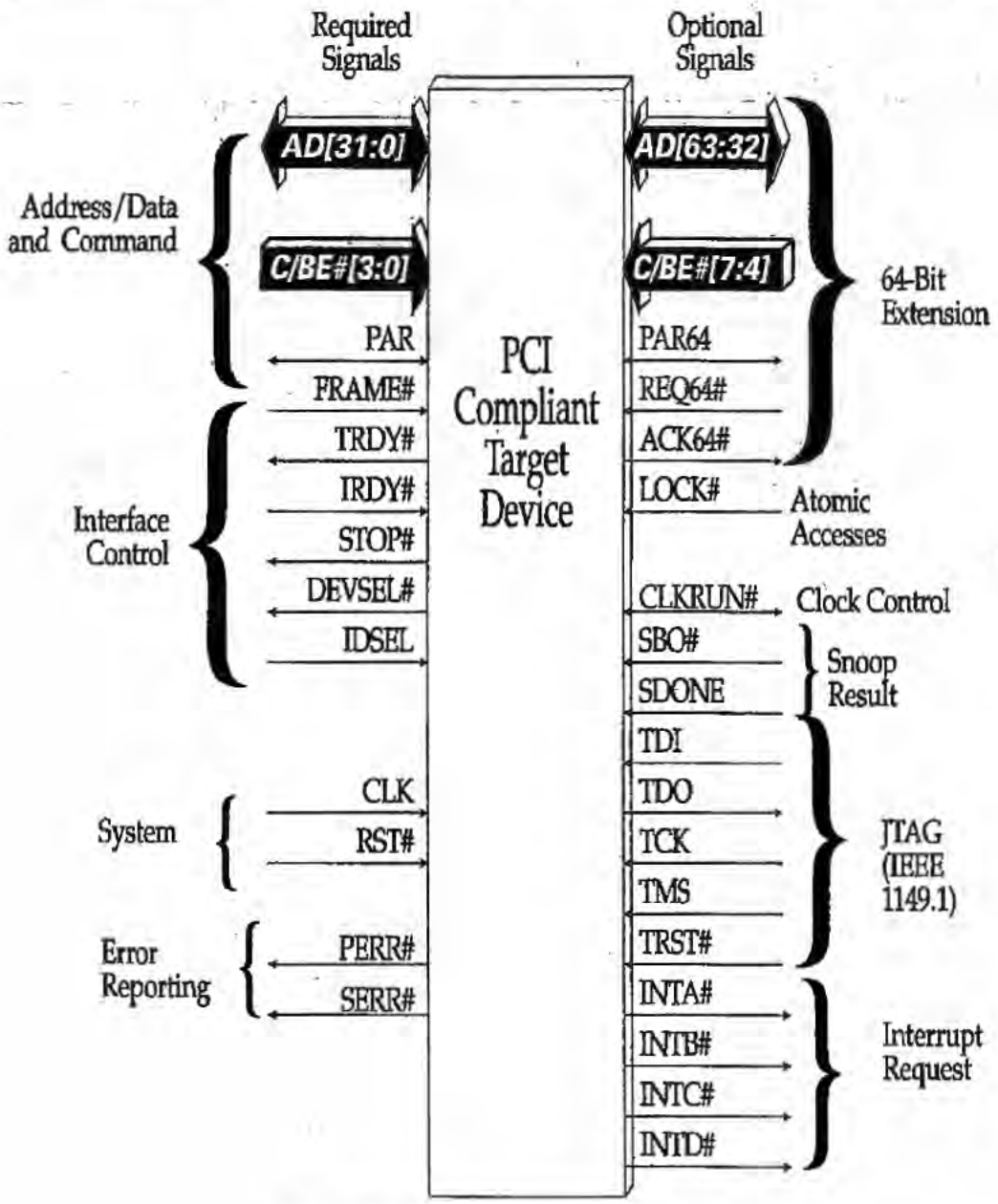


Figure 5-2. PCI-Compliant Target Device Signals

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 com

pleted 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).

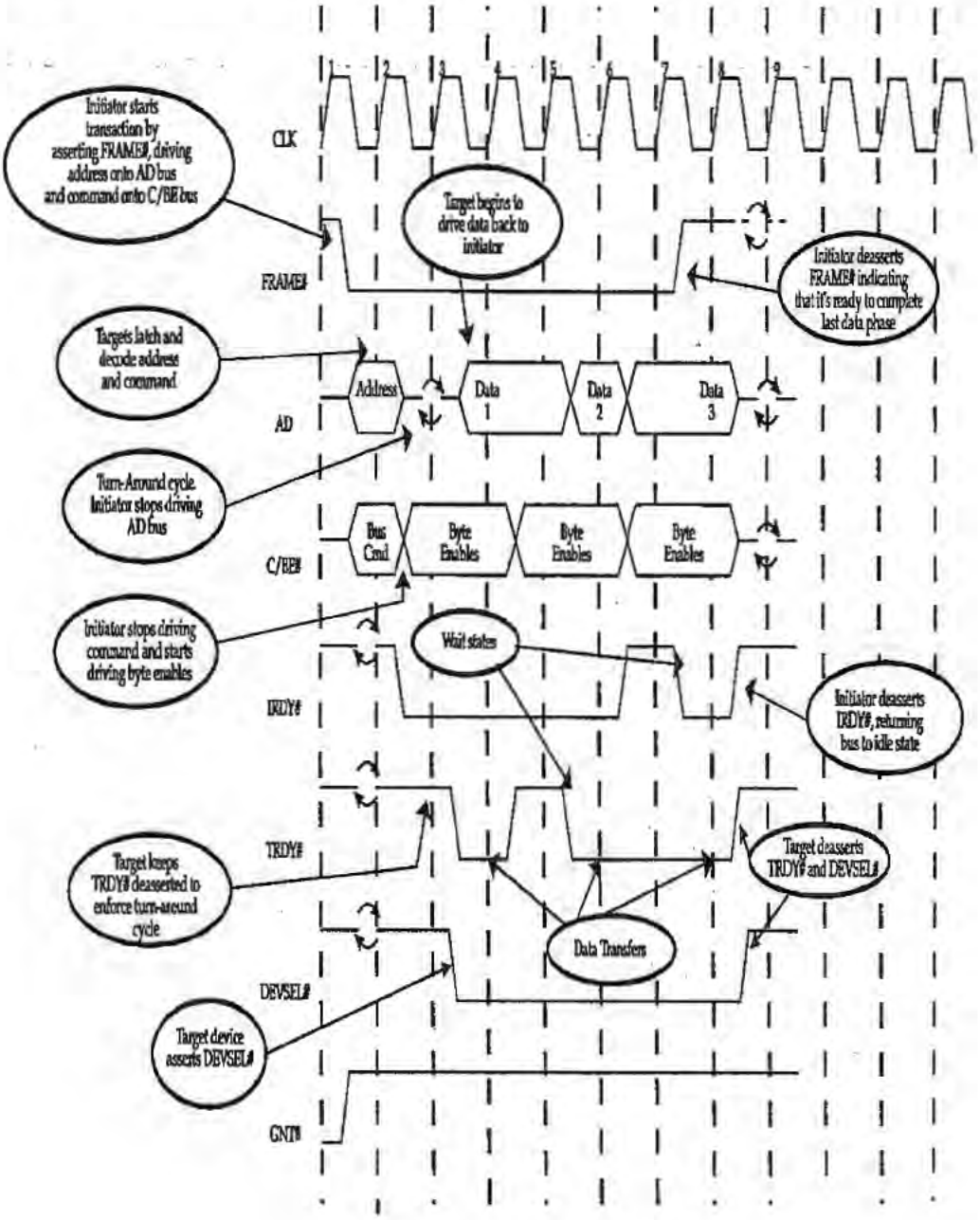


Figure 5-3. Typical PCI Timing Diagram

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.

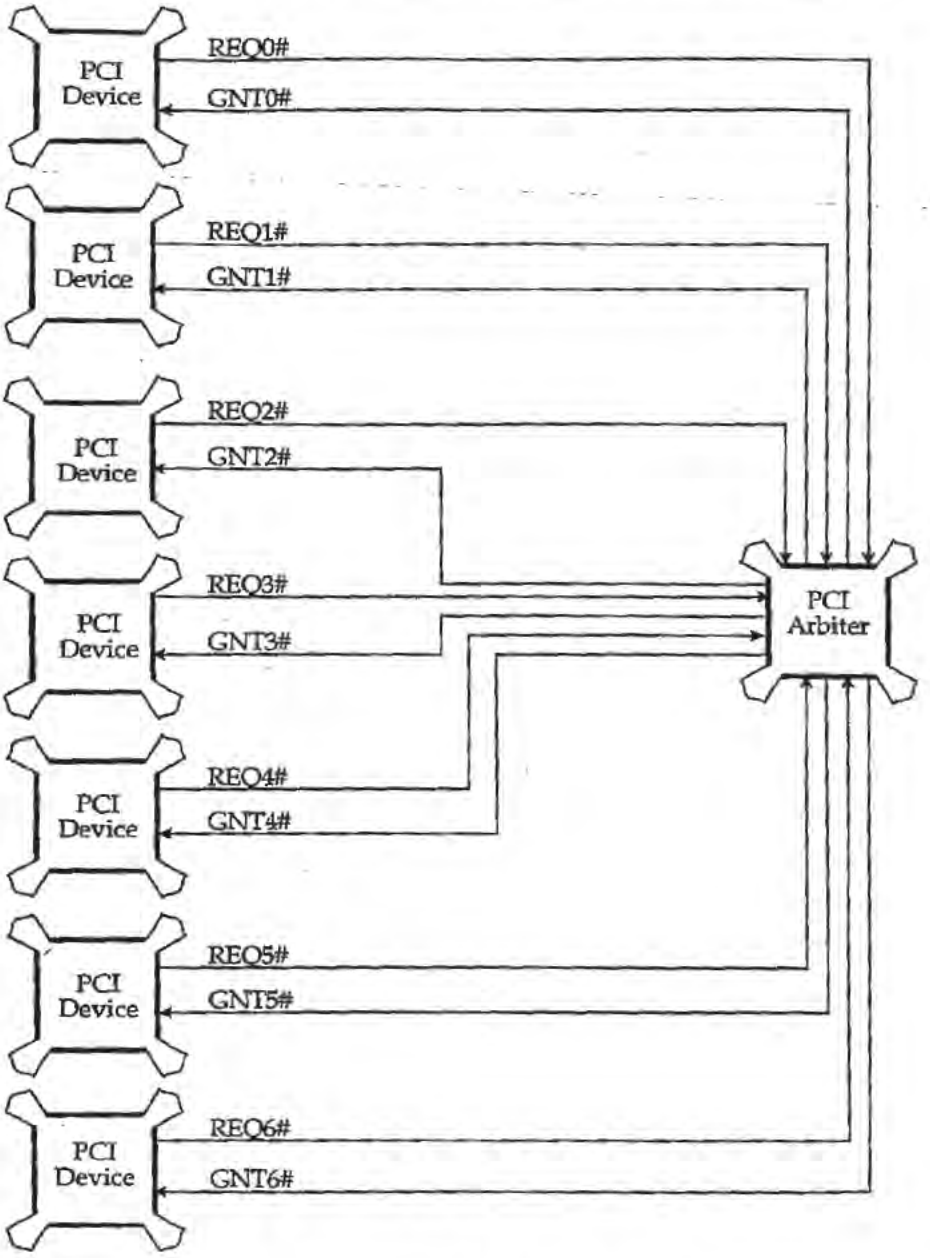


Figure 6-1. The PCI Bus Arbiter

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.

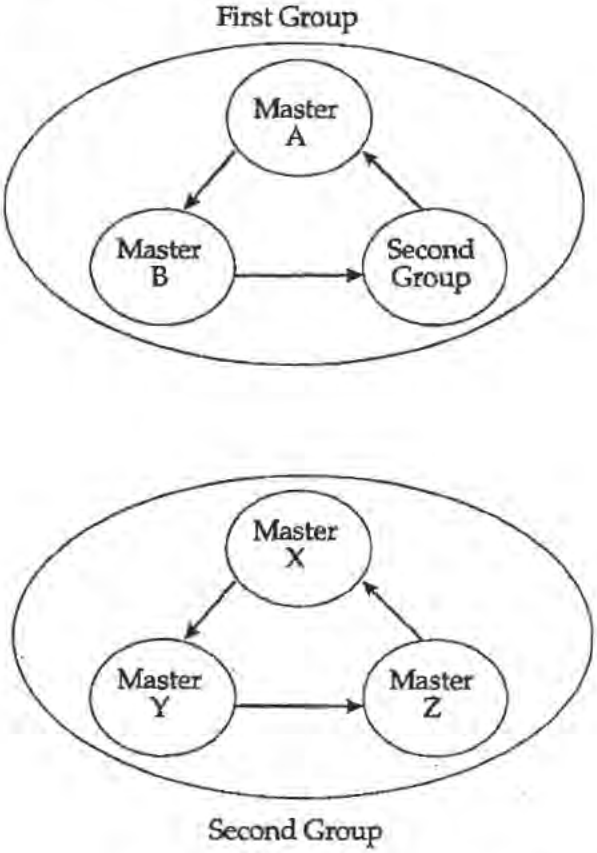


Figure 6-2. Example Arbitration Scheme

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.

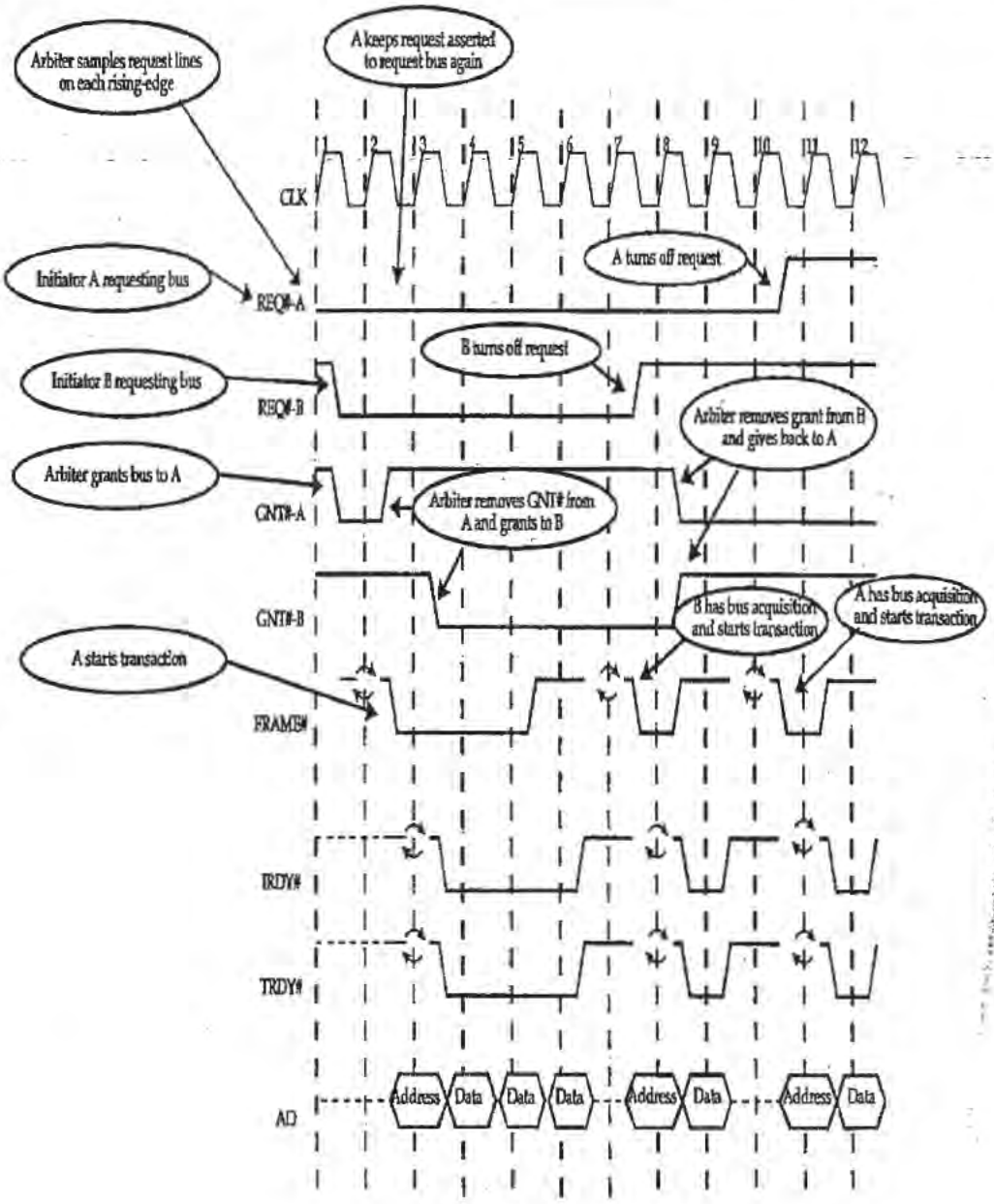


Figure 6-3. PCI Bus Arbitration Between Two Masters

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.

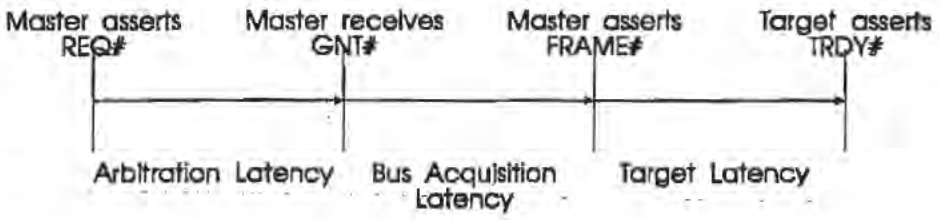


Figure 6-4. Access Latency Components

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.

More than likely, the busy condition will have been cleared by the time the master retries the transaction. It is possible, however, that the master may have to make several attempts before succeeding. A target is only permitted to use this option is there is a high probability that it will be able to complete the transfer the first time that the master retries it. Otherwise, it must use option three.

3. In the third case (option three), the target has to access a slow medium to fetch the requested data and it will take longer than 16 clocks. In this case, the target latches the address, command and the first set of byte enables and then issues a retry to the initiator. The initiator is thereby forced to end the transaction with no data transferred and is required to retry the transaction again later using precisely the same address, command and byte enables. The target, meanwhile, proceeds to fetch the requested data and set it up in a buffer for the master to read later when it retries the transaction. When the target sees the master retry the transaction, it attempts to match the second request with the initial request by comparing the start address, command and initial byte enables to those latched earlier. If they match, the requested data is transferred to the master. If they aren't an exact match, the target interprets this as a new request (for data other than that in its buffer) and issues a retry to the master again. To summarize, if the master doesn't duplicate the transaction exactly each time it retries the transaction, it will never have its read request fulfilled. The target is not required to service retries from its buffered data that aren't exact matches. Option three is referred to as a delayed transaction. It can also be used for a write transaction (e.g., where the bus master is not permitted to proceed with other activities until it accomplishes the write). In this case, the target latches the address, command, byte enables and the first data item and issues the retry. It then proceeds to write the data item to the slow destination. Each time that the master retries the write transaction it will receive a retry until the target device has acknowledge receipt of the data. When the target is ready to permit the transfer and the master next attempts the access, the target compares the address, command, byte enables and the write data to determine if this is the same master that initially requested the write transfer.

Different Master Attempts Access To Device With Previously-Latched Request

If a different master attempts to access the target and the target can only deal with one latched request at a time, it must issue a retry to the master without latching its transaction information.

Special Cycle Monitoring While Processing Request

If the target is designed to monitor for special cycles, it must be able to process a special cycle during the same period of time that is processing a previously latched read or write request.

Delayed Request and Delayed Completion

A delayed transaction consists of two parts: the request phase and the completion phase. The request phase occurs when the target latches the request and issues retry to the master. This is referred to as the delayed request transaction. Once the transaction has been latched, the target (typically a bridge to a slow expansion bus) begins the transaction on the target bus. When the transfer completes on the target bus, this is referred to as the delayed completion transaction. This is the start of the completion phase. A delayed transaction must complete on the target bus before it is permitted to complete on the initiating bus. The master is required to periodically re-attempt the transfer until the target finally asserts TRDY# and allows the data to be transferred. This ends the completion phase of the delayed transaction.

Handling Multiple Data Phases

When the master is successful in completing the first data phase, it may proceed with more data phases. The target may issue a disconnect on any data phase after the first. The master is not required to resume the transaction later. Both the master and the target consider the original request fulfilled.

Master or Target Abort Handling

A delayed transaction is also considered completed if it receives a master abort or a target abort rather than a retry on a re-attempt of the retried transaction. The target compares to ensure that the master is the one that originated the request before it issues the master or target abort to it. This means that the transaction on the target bus ended in a master abort because

no target responded or in a target abort because of a broken target. In both of these cases, the master is not required to repeat the transaction.

Commands That Can Use Delayed Transactions

A delayed transaction normally consist of a single data phase and is used for the following commands:

- Interrupt Acknowledge.
- I/O read.
- I/O write.
- Memory read.
- Configuration read.
- Configuration write.

The delayed transaction could also be used with the memory write commands, but it's results in better performance to post the write an permit the master to complete the write quickly.

Delayed Read Prefetch

A delayed read can result in the reading more data than indicated in the master's initial data phase if the target knows that prefetching data doesn't alter the contents of memory locations (as it would in memory-mapped I/O ports) The target can prefetch more data than initially requested under the following circumstances:

- The master has used the memory read line or memory read multiple command, thereby indicating that it knows the target is prefetchable memory.
- The master used a memory read command, but the bridge that accepted the delayed transaction request recognizes that the address falls within a range defined as prefetchable.

In all other cases, the target (i.e., the bridge) cannot perform anything other than the single data phase indicated by the originating master.

Request Queuing and Ordering Rules

A target device (typically a bridge) can be designed to latch and process multiple delayed requests. The device must, however, ensure that the

transactions are performed in the proper order. Table 6-3 defines the rules that the device must observe in order to ensure that posted memory writes and delayed transactions are performed in the proper order. The table was extracted from the specification. The following abbreviations are used in the table:

- **PMW = posted memory write.** The master is permitted to end a memory write immediately if the device posts it.
- **DRR = delayed read request.** A delayed read request occurs when the target latches the address, command and byte enables and issues a retry to the master. It is then the responsibility of the target to perform the read on the target bus to fetch the requested data.
- **DWR = delayed write request.** A delayed write request occurs when the target latches the address, command, byte enables and write data and issues a retry to the master. It is then the responsibility of the target to perform the write on the target bus.
- **DRC = delayed read completion.** A delayed read completion occurs when the device that latched a read request completes reading the requested data on the target bus and has the data ready to deliver to the master that originated the request. The device is now waiting for the originating master to retry its read so that it may deliver the data to the master.
- **DWC = delayed write completion.** A delayed write completion occurs when the device that latched a write request completes writing the data on the target for the master that originated the write. The device is now waiting for the originating master to retry its write so that it may confirm the delivery of the write data.

The table is formatted as follows:

- The first column represents a delayed transaction request (one of five types) that has just been latched.
- The second column indicates whether the transaction just latched can pass a previously-posted memory write.
- The third column indicates whether the transaction just latched can pass a previously-latched delayed read request.
- The fourth column indicates whether the transaction just latched can pass a previously-latched delayed write request.
- The fifth column indicates whether the transaction just latched can pass a previously-latched delayed read completion.

- The sixth column indicates whether the transaction just latched can pass a previously-latched delayed write completion.

The rule list immediately following the table was extracted from the specification. The superscripts in each box corresponds to the rule list.

As an example, the table indicates that a posted memory write can pass (be performed) a delayed read or write request or a delayed write completion, but it is not permitted to pass another posted memory write or a delayed read completion.

Table 6-3. Ordering Rules

		Delayed Request		Delayed Completion	
Transaction just latched	**PMW**	**DRR**	**DWR**	**DRC**	**DWC**
PMW	No[1]	Yes[2]	Yes[2]	No[4]	Yes[8]
DRR	No[3]	No[1]	No[1]	No[4]	Yes[8]
DWR	No[3]	No[1]	No[1]	No[4]	Yes[8]
DRC	No[6]	Yes[5]	Yes[5]	No[1]	No[1]
DWC	Yes[7]	Yes[5]	Yes[5]	No[1]	No[1]

Rule list:

1. Transactions of the same type cannot pass each other.
2. A posted memory write can pass a delayed request.
3. A delayed request cannot complete before a posted memory write.
4. A posted memory write or a delayed request cannot pass a delayed read completion.
5. A delayed completion can pass a delayed request.
6. A delayed read completion cannot pass a posted memory write.
7. A delayed write completion can pass a posted memory write.
8. A posted memory write or a delayed read request can pass a delayed write completion.

The primary rule is that all device accesses must complete in order from the programmer's perspective. In the following list, the author has attempted to explain each table entry.

1. A newly-latched PMW cannot pass (be completed before) a previously-PMW because all writes have to complete in the order in which they have been latched.
2. A newly-latched PMW can pass a previously-latched DRR. This is permitted because the master has already completed the write while the other master has not yet completed its read. From the programmer's standpoint, this means the write completed before the read.
3. A newly-latched PMW can pass a previously-latched DWR. This is permitted because the master has already completed the posted-write while the other master has not yet completed its delayed write. From the programmer's standpoint, this means the posted-write completed before the delayed write.
4. A newly-latched PMW cannot pass a previously-latched DRC. From the programmer's perspective, the write has already completed but the read has not. One master originated the read before the write was performed by the other master, so the programmer expects to get back the read data as it looked before the write occurred.
5. A newly-latched PMW can pass a previously-latched DWC. The device has completed the write to the target and is waiting for the delayed master to reattempt the write so that it can let the master complete the write. From the programmer's perspective, the posted-write has already completed while the delayed write hasn't.
6. A newly-latched DRR cannot pass a previously-PMW. If this were permitted, the read might fetch stale data (because the posted write might be to one of the locations to be read).
7. A newly-latched DRR cannot pass a previously-latched DRR. The reads must complete in the order the programmer generated them.
8. A newly-latched DRR cannot pass a previously-latched DWR. The write was originated before the read and must therefore occur before the read (in case they target the same locations.
9. A newly-latched DRR cannot pass a previously-latched DRC. The reads must complete in the order the programmer generated them.
10. A newly-latched DRR can pass a previously-latched DWC. The target has already been written to and updated, so it contains fresh information. The device may therefore initiate the read from the target to fetch the data requested by the originator.
11. A newly-latched DWR cannot pass a previously-PMW. From the programmer's perspective, the posted write occurred before the delayed write (which has not yet completed). The device must perform the posted write before the newly-accepted delayed write so that the data is delivered to the target(s) in the correct order.

12. A newly-latched DWR cannot pass a previously-latched DRR. It is the programmer's intention that the read occur before the write.
13. A newly-latched DWR cannot pass a previously-latched DWR. It is the programmer's intention that the two writes occur in the order received.
14. A newly-latched DWR cannot pass a previously-latched DRC. The programmer initiated the read before the write, so the read must be permitted to complete (on the originating bus) before the write occurs.
15. A newly-latched DWR can pass a previously-latched DWC. The data for the first write (the DWC) has already been delivered to the target, so the data from the second write (the DWR) can now be delivered. The target(s) will receive the data in the order intended by the programmer.
16. A newly-latched DRC cannot pass a previously-PMW. If the write and read are accessing the same locations, the read would return stale data. From the programmer's perspective, the write has already completed and the target data updated. If reading from the same location(s), the programmer therefore expects to receive the newly-written data.
17. A newly-latched DRC can pass a previously-latched DRR. The DRC is associated with a DRR that was received prior to the DRR that is still outstanding. The data from the DRC can therefore be delivered to the requesting master immediately.
18. A newly-latched DRC can pass a previously-latched DWR. The data associated with the DRC was requested prior to the reception of the DWR by the device. The read data can therefore be delivered to the requesting master immediately (before the write is performed on the target bus).
19. A newly-latched DRC cannot pass a previously-latched DRC. Read requests must be performed in the order that they were received.
20. A newly-latched DRC cannot pass a previously-latched DWC. The data associated with the DRC was requested prior to the reception of the DWR that caused the DWC. The read data can therefore be delivered to the requesting master immediately (before the write is performed on the target bus).
21. A newly-latched DWC can pass a previously-PMW. Writes must complete in the order they are received and the write associated with the DWC was received prior to the write associated with the PMW.
22. A newly-latched DWC can pass a previously-latched DRR. The write originated before the read, so the master that originated the write can be told about its completion immediately.
23. A newly-latched DWC can pass a previously-latched DWR. The write associated with the DWC originated before the write that originated the DWR. The master that originated the DWC can therefore be told about the write completion immediately.

24. A newly-latched DWC cannot pass a previously-latched DRC. The read associated with the DRC originated before the write associated with the DWC. The master that originated the read must therefore be given the read data before the master that originated the write is told of its completion.
25. A newly-latched DWC cannot pass a previously-latched DWC. The write associated with the previously-completed DWC originated before the write associated with the just completed DWC. The completions must therefore be reported to the originating masters in that order.

Locking, Delayed Transactions and Posted Writes

The following rules must be followed when a device permits delayed transactions and also supports locking:

1. A target that accepts a locked access (i.e., it latches the request) must behave as a locked target.
2. The target cannot accept any posted writes after accepting a delayed lock request moving in the same direction (except as noted by rule five).
3. While locked, the target may continue to accept delayed requests.
4. Posting of write data in the opposite direction of the locked access must be disabled once lock has been established on the destination bus.
5. Posting of write data from the locking master is allowed.
6. Once lock has been established (between the originating master and the actual target), the device stays locked until LOCK# and FRAME# are sampled deasserted (on the same rising-edge of the clock) on the originating bus.

Fast Back-to-Back Transactions

Assertion of its grant by the PCI bus arbiter gives a PCI bus master access to the bus for a single transaction. If a bus master desires another access, it should continue to assert its REQ# after it has asserted FRAME# for the first transaction. If the arbiter continues to assert its GNT# at the end of the first transaction, the master may then immediately initiate a second transaction. However, a bus master attempting to perform two, back-to-back transactions usually must insert an idle cycle between the two transactions. This is illustrated in figure 6-5. When it doesn't have to insert the idle cycle between the two bus transactions, this is referred to as fast back-to-back transactions. This can only occur if there is a guarantee that there will not be contention (on

any signal lines) between the masters and/or targets involved in the two transactions. There are two scenarios where this is the case.

1. In the first case, the master guarantees that there will be no contention.
2. In the second case, the master and the community of PCI targets collectively provide the guarantee.

The sections that follow describe these two scenarios.

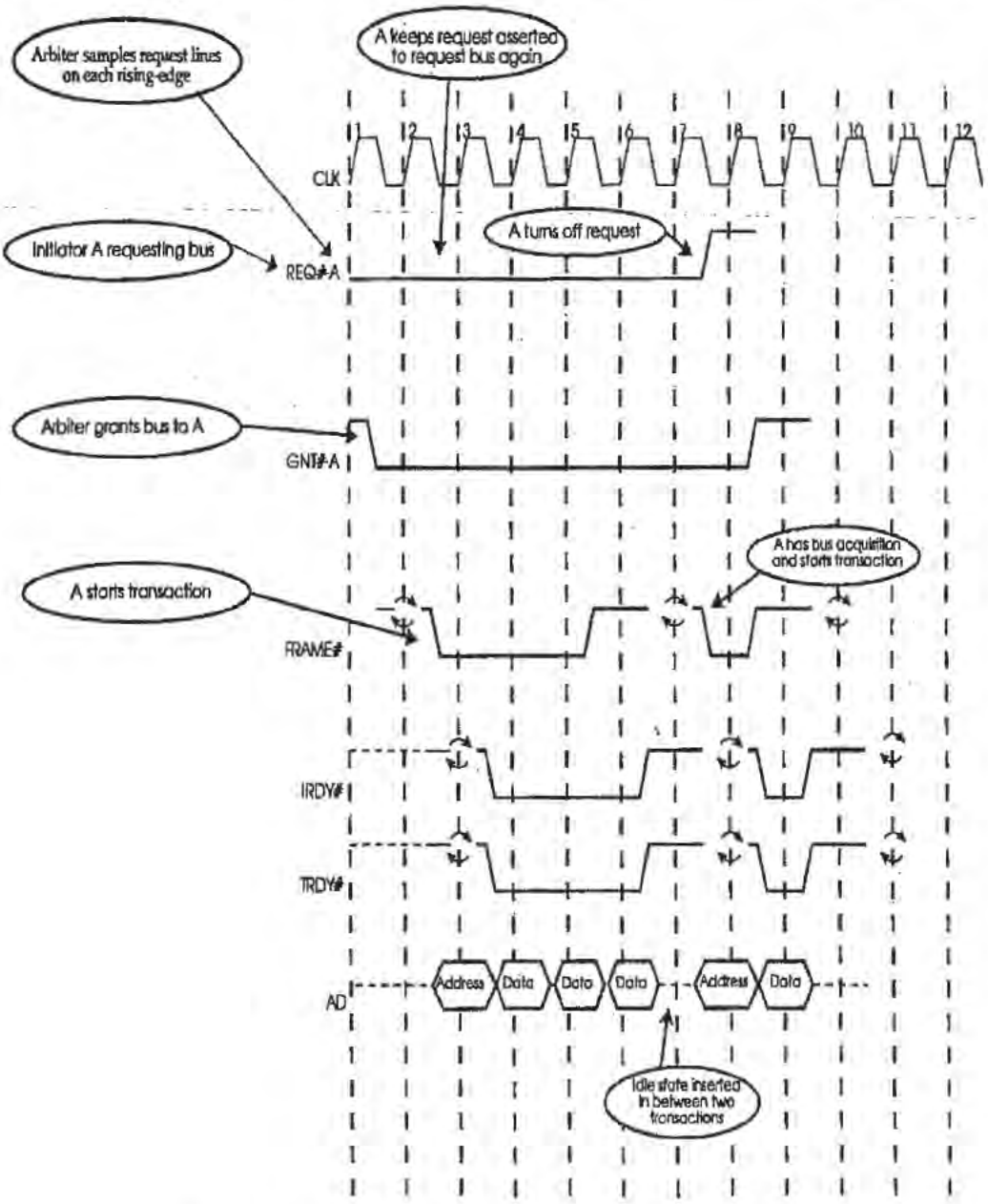


Figure 6-5. Back-to-Back Transactions With an Idle State In-Between

Decision to Implement Fast Back-to-Back Capability

The subsequent two sections describe the rules that permit deletion of the idle state between two transactions. Since they represent a fairly constraining set of rules, the designer of a bus master should make an informed decision as to whether or not it's worth the additional logic it would take to implement it.

Assume that the nature of a particular bus master is such that it typically performs long burst transfers whenever it acquires bus ownership. In this case, including the extra logic to support fast back-to-back transactions would not make a great deal of sense. Percentage-wise, you're only saving one clock tick of latency in between each pair of long transfers.

Assume that the nature of another master is such that it typically performs lots of small data bursts. In this case, inclusion of the extra logic may result in a measurable increase in performance. Since each of the small transactions typically only consists of a few clock ticks and the master performs lots of these small transactions in rapid succession, the savings of one clock tick in between each transaction pair can amount to the removal of a fair percentage of overhead normally spent in bus idle time.

Scenario One: Master Guarantees Lack of Contention

In this scenario (defined in revision 1.0 of the specification and still true in revision 2.x), the master must ensure that, when it performs two back-to-back transactions with no idle state in between the two, there is no contention on any of the signals driven by the bus master or on those driven by the target. An idle cycle is required whenever AD[31:0], C/BE#[3:0], FRAME# and IRDY# are driven by different masters from one clock cycle to the next. The idle cycle allows one cycle for the master currently driving these signals to surrender control (cease driving) before the next bus master begins to drive the bus. This prevents bus contention.

How Collision Avoided On Signals Driven By Master

The master must ensure that the same set of output drivers are driving the master-related signals at the end of the first transaction and the start of the second. This means that the master must ensure that it is driving the bus at the end of the first transaction and at the start of the second.

To meet this criteria, the first transaction must be a write transaction and the second transaction can be either a read or a write but must be initiated by the same master. Refer to figure 6-6. When the master acquires bus ownership and starts the first transaction (clock edge one), it asserts FRAME# and continues to assert its REQ# line to request the bus again after the completion of the current transaction. When the address phase is completed (clock edge two), the master drives the first set of data bytes onto the AD bus and sets the byte enables to indicate which data paths contain valid data bytes. At the conclusion of the first (clock edge three) and any subsequent data phases, the bus master is driving the AD bus and the byte enables. Furthermore, the bus master is asserting IRDY# during the final data phase. On the rising-edge of the PCI clock where the final data item is transferred (clock edge three), FRAME# has already been deasserted and IRDY# asserted (along with TRDY# and DEVSEL#). If, on this same clock edge (clock edge three) the master samples its GNT# still asserted by the arbiter, this indicates that it has retained bus ownership for the next transaction.

In the clock cell immediately following this clock edge (clock edge three), the master can immediately reassert FRAME# and drive a new start address and command onto the bus. There isn't a collision on the FRAME# signal because the same output driver that was driving FRAME# deasserted at the end of the first transaction begins to assert FRAME# at the start of the second transaction. There isn't a collision on the AD bus or the C/BE bus because the same master's drivers that were driving the final data item and byte enables at the end of the first transaction are driving the start address and command at the start of the second transaction.

At the end of the address phase of the second transaction (clock edge four), the same master that was deasserting IRDY# at the end of the first transaction begins to reassert it (so there is no collision between two different IRDY# drivers).

How Collision Avoided On Signals Driven By Target

The signals asserted by the target of the first transaction at the completion of the final data phase (clock edge three) are TRDY# and DEVSEL# (and, possibly, STOP#). Two clocks after the end of the data phase, the target may also drive PERR#. Since it is a rule in this scenario that the same target must be addressed in the second transaction, the same target again drives these signals. Even if the target has a fast address decoder and begins to assert DEVSEL# (and TRDY# if it is a write) during clock cell four in the second

transaction, the fact that it is the same target ensures that there is not a collision on TRDY# and DEVSEL# (and possibly STOP# and PERR#) between output drivers associated with two different targets.

How Targets Recognize New Transaction Has Begun

It is a rule that all PCI targets must recognize either of the following conditions as the start of a new transaction:

- Bus idle (FRAME# and IRDY# deasserted) on a rising-edge of the PCI clock followed on the next rising-edge by address phase in progress (FRAME# asserted and IRDY# deasserted).
- Final data phase in progress (FRAME# deasserted and IRDY# asserted) on a rising-edge of the PCI clock, followed on the next rising-edge by address phase in progress (FRAME# asserted and IRDY# deasserted).

Implementation of support for this type of fast back-to-back capability is optional for an initiator, but all targets must be able to decode them.

Fast Back-to-Back and Master Abort

When a master experiences a master abort on a transaction during a fast back-to-back series, it may continue performing fast transactions (as long as it still has its GNT#). No target responded to the aborted transaction, thereby ensuring that there will not be a collision on the target-related signals. If the transaction that ended with a master abort was a special cycle, the target(s) that received the message were already given sufficient time (by the master) to process the message and should be prepared to recognize another transaction. The author would like to note that this portion of the 2.1 specification states that the target(s) of the special cycle were given five clocks after the last data transfer to process the message. This conflicts with the specification description of the special cycle which cites four clocks are required after the last data transfer.

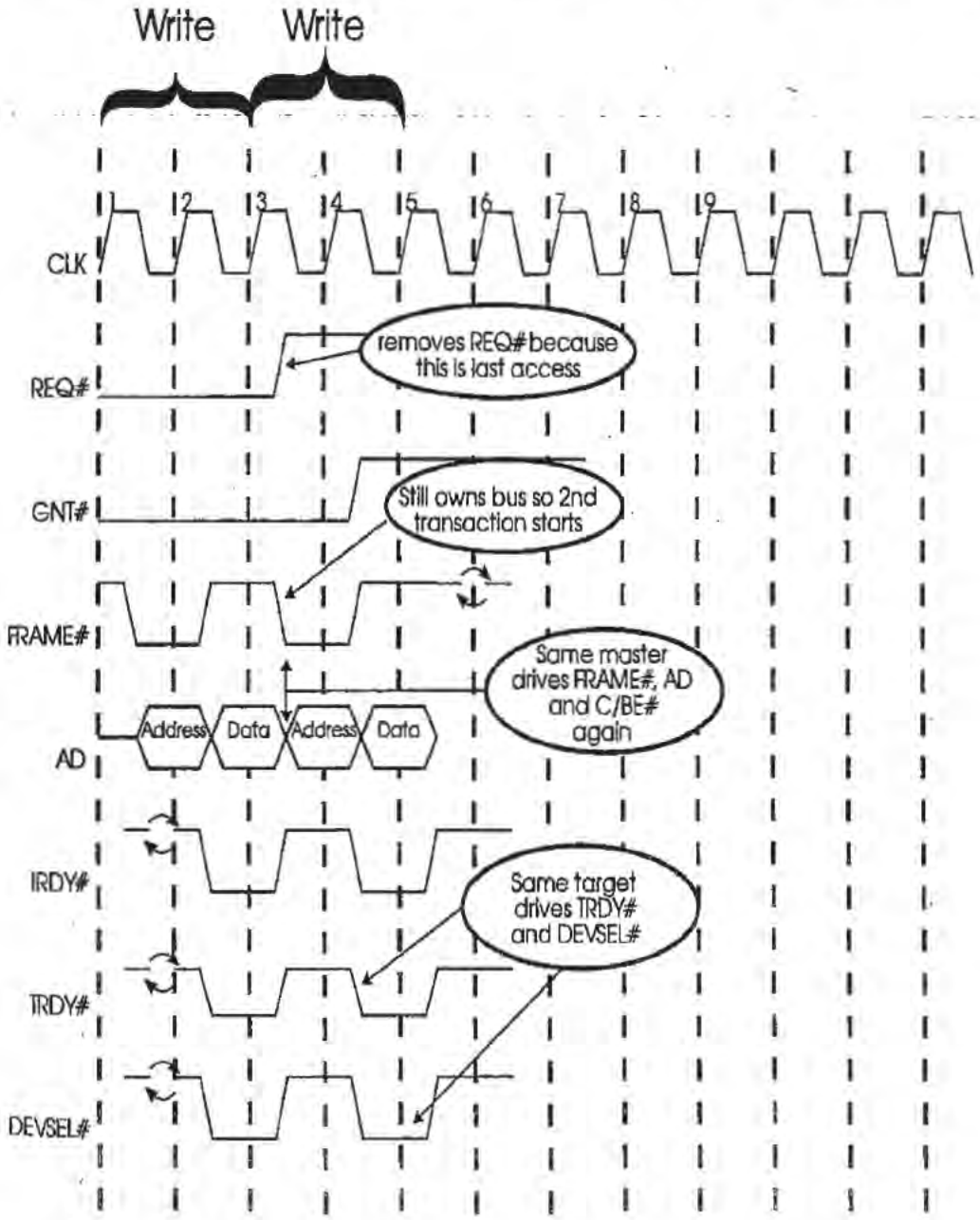


Figure 6-6. Arbitration For Fast Back-To-Back Accesses

Scenario Two: Targets Guarantee Lack of Contention

In the second scenario (defined in revision 2.0 of the specification and still true in revision 2.1), the entire community of PCI targets that reside on the PCI bus and the bus master collectively guarantee lack of contention during fast back-to-back transactions. A constraint incurred when using the master-guaranteed method (defined in revision 1.0 of the specification) is that the master can only perform fast back-to-back transactions if both transactions access the same target and the first transaction is a write.

The reason that scenario one states that the target of the first and second transactions must be the same target is to prevent the possibility of a collision on the target-related signals: TRDY#, DEVSEL# and STOP# (and, possibly, PERR#). This possibility can be avoided if:

1. All targets have medium or slow address decoders and
2. All targets are capable of discerning that a new transaction has begun without a transition through the bus idle state and are capable of latching the address and command associated with the second transaction.

If the full suite of targets on a PCI bus meet these requirements, then any bus master that is fast back-to-back capable can perform fast back-to-back transactions with different targets in the first and second transactions. The first transaction must still be a write, however, and the second transaction must be performed by the same master (to prevent collisions on master-related signals).

The previous statement implies that there is a method to determine if all targets support this feature. During system configuration (at power-up), software polls each device's configuration status register and checks the state of its FAST BACK-TO-BACK CAPABLE bit. The designer of a device hardwires this read-only bit to zero if the device doesn't support this feature, while hardwiring it to a one indicates that it does. If all devices indicate support for this capability, then the configuration software can set each bus master's FAST BACK-TO-BACK ENABLE bit in its configuration command register (this bit, and therefore this capability is optional for a bus master). When this bit is set, a master is enabled to perform fast back-to-back transactions with different targets in the first and second transactions.

A target supports this capability if it meets the following criteria:

- Normally a target recognizes a bus idle condition by sampling FRAME# and IRDY# deasserted. It then expects and recognizes the start of the next transaction by sampling FRAME# asserted and IRDY# deasserted. At that point, it latches the address and command and begins address decode. To support the feature under discussion, it must recognize the completion of the final data phase of one transaction by sampling FRAME# deasserted and IRDY# and TRDY# asserted. This would then be immediately followed by the start of the next transaction, as indicated by sampling FRAME# asserted and IRDY# deasserted on the next rising-edge of the PCI clock.
- The target must ensure that there isn't contention on TRDY#, DEVSEL# and STOP# (and, possibly, PERR#). If the target has a medium or slow address decoder, this provides the guarantee. If the target has a fast address decoder, it must delay assertion of these three signals by one clock to prevent contention. Note that this does not affect the DEVSEL# timing field in the device's configuration status register. The setting in this field is used by the bus's subtractive decoder to adjust when it asserts DEVSEL# to claim transactions unclaimed by PCI devices. During the second transaction of a fast back-to-back transaction pair, the subtractive decoder must delay its assertion of DEVSEL# if it normally claims during the medium or slow time slot (otherwise, a collision may occur on DEVSEL#, TRDY#, and STOP# (and, possibly, PERR#).
- There are two circumstances when a target with a fast address decoder doesn't have to insert this one clock delay:

1. The current transaction was preceded by a bus idle state (FRAME# and IRDY# deasserted).
2. The currently-addressed target was also addressed in the previous transaction. This ensures a lack of contention on TRDY#, STOP# and DEVSEL# (because it was driving these signals during the previous transaction).

State of REQ# and GNT# During RST#

While RST# is asserted, all masters must tri-state their REQ# output drivers and must ignore their GNT# inputs.

Pullups On REQ# From Add-In Connectors

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.

Broken Master

The arbiter may assume that a master is broken if the arbiter has issued GNT# to the master, the bus has been idle for 16 clocks, and the master has not asserted FRAME# to start its transaction. The arbiter is permitted to ignore all further requests from the broken master and may optionally report the failure to the operating system (in a device-specific fashion).

Chapter 7

The Previous Chapter

The previous chapter provided a description of PCI bus arbitration.

In This Chapter

This chapter defines the types of commands, or transaction types, that a bus master may initiate when it has acquired ownership of the PCI bus.

The Next Chapter

The next chapter provides a detailed analysis of the PCI transfer, utilizing timing diagrams and a description of each step involved in the transfer.

Introduction

When a bus master acquires ownership of the PCI bus, it may initiate one of the types of transactions listed in table 7-1. During the address phase of a transaction, the Command/Byte Enable bus, C/BE#[3:0], is used to indicate the command, or transaction, type. Table 7-1 provides the setting that the initiator places on the Command/Byte Enable lines during the address phase of the transaction to indicate the type of transaction in progress. The following sections provide a description of each of the command types.

Table 7-1. PCI Command Types

C/BE3#	C/BE2#	C/BE1#	C/BE0#	Command Type
0	0	0	0	Interrupt Acknowledge
0	0	0	1	Special Cycle
0	0	1	0	I/O Read
0	0	1	1	I/O Write
0	1	0	0	Reserved
0	1	0	1	Reserved
0	1	1	0	Memory Read
0	1	1	1	Memory Write
1	0	0	0	Reserved
1	0	0	1	Reserved
1	0	1	0	Configuration Read
1	0	1	1	Configuration Write
1	1	0	0	Memory Read Multiple
1	1	0	1	Dual-Address Cycle
1	1	1	0	Memory Read Line
1	1	1	1	Memory Write and Invalidate

Interrupt Acknowledge Command

Introduction

In response to an interrupt request, an Intel x86 processor issues two interrupt acknowledge transactions to read the interrupt vector from the interrupt controller. The interrupt vector tells the processor which interrupt service routine to execute.

Background

In an Intel x86-based system, the host processor is usually the device that services interrupt requests received from subsystems that require servicing. In a PC-compatible system, the subsystem requiring service issues a request by asserting one of the system interrupt request signals, IRQ0 through IRQ15. When the IRQ is detected by the interrupt controller, it asserts INTR to the host processor. Assuming that the host processor is enabled to recognize interrupt requests (the interrupt flag bit in the EFLAGS register is set to one), the processor responds by requesting the interrupt vector from the interrupt

controller. This is accomplished by the processor stepping through the following sequence:

1. **Processor generates an interrupt acknowledge bus cycle**. No address is output by the processor because the address of the target device, the interrupt controller, is implicit in the bus cycle type. The purpose of this bus cycle is to **command the interrupt controller to prioritize its currently-pending requests** and select the request to be processed. The processor doesn't expect any data to be returned by the interrupt controller during this bus cycle.
2. **Processor generates a second interrupt acknowledge bus cycle to request the interrupt vector** from the interrupt controller. BE0# is asserted by the processor, indicating that an 8-bit vector is expected to be returned on the lower data path, D[7:0]. To state this more plainly, the processor requests that the interrupt controller return the index into the interrupt table in memory. This tells the processor which table entry to read. The table entry contains the start address of the device-specific interrupt service routine in memory. In response to the second interrupt acknowledge bus cycle, the interrupt controller must drive the interrupt table index, or vector, associated with the highest-priority request currently pending back to the processor over the lower data path, D[7:0], and assert ready to the processor to indicates the presence of the vector. In response, the processor reads the vector from the bus and uses it to determine the start address of the interrupt service routine that it must execute.

Host/PCI Bridge Handling of Interrupt Acknowledge Sequence

When the host/PCI bridge detects the start of an interrupt acknowledge sequence on the host side, it can handle it one of two ways:

1. It filters out (does not pass to the PCI bus) the first interrupt acknowledge bus cycle. Ready is asserted to the processor to terminate the first interrupt acknowledge bus cycle. When the processor initiates the second interrupt acknowledge bus cycle, the bridge acquires the PCI bus and initiates a PCI interrupt acknowledge transaction. This transaction is illustrated in figure 7-1 and is described in the next section. When the PCI target that contains the interrupt controller detects the interrupt acknowledge transaction, it asserts DEVSEL# to claim the transaction. It then internally generates two, back-to-back interrupt acknowledge pulses to the

8259A interrupt controller, thereby emulating the double interrupt acknowledge generated by an Intel x86 processor. In response, the interrupt controller drives the interrupt vector onto the lower data path and asserts TRDY# to indicate the presence of the vector to the initiator (the host/PCI bridge). When the host/PCI bridge samples TRDY# and IRDY# asserted, it reads the vector from the lower data path and terminates the PCI interrupt acknowledge transaction. During this period, the bridge was inserting wait states into the host processor's second interrupt acknowledge bus cycle. It then drives the 8-bit interrupt vector onto the processor's lower data path and asserts ready to the processor. When the processor samples ready asserted, it reads the vector from the bus and uses it to index into the memory-based interrupt table to get the start address of the interrupt service routine to execute.

2. Instead of filtering out the first of the processor's interrupt acknowledge bus cycles, the bridge could pass it onto the PCI bus. Rather than waiting for the completion of the PCI transaction, however, the bridge would immediately assert ready to the processor, permitting it to end the first interrupt acknowledge bus cycle and begin the second. This would permit the interrupt controller to claim the transaction earlier and therefore return the vector sooner. When the interrupt controller returns the vector, it is passed directly back to the processor and ready is asserted, permitting the processor to read the vector and terminate the second bus cycle.

PCI Interrupt Acknowledge Transaction

Figure 7-1 illustrates the PCI interrupt acknowledge transaction. The bridge does not drive an address onto the AD bus during the address phase, but must drive stable data onto the AD bus along with correct parity on the PAR line. The C/BE bus contains the interrupt acknowledge command during the address phase. During the data phase, the target holds off the assertion of TRDY# and DEVSEL# to enforce the turnaround cycle. This is necessary to permit the bridge sufficient time to turn off its AD bus output drivers before the target (the interrupt controller) begins to drive the requested interrupt vector back to the bridge on the AD bus. The target then drives the vector onto the data path(s) indicated by the byte enable settings on the C/BE bus (just BE0# asserted in an ix86 environment) and asserts TRDY# to indicate the presence of the requested vector. The byte enables are a duplicate of the byte enables asserted by the host processor during its second interrupt acknowledge bus cycle. When the bridge samples IRDY# and TRDY# asserted, it reads the vector from the AD bus and terminates the PCI interrupt acknowledge transaction. It then passes the vector back to the host processor and asserts

ready to indicate its presence. When the host processor samples ready asserted, it reads the vector from its data bus and terminates the second interrupt acknowledge bus cycle.

In a PowerPC, PRePcompliant platform, the programmer performs a one to four byte memory read from memory location BFFFFFF0h. When the host/PCI bridge detects this read, it acquires ownership of the PCI bus and initiates the PCI interrupt acknowledge transaction. When the interrupt controller supplies the requested vector to the host/PCI bridge, the bridge in turn supplies it to the processor and asserts TA# to indicate its presence. The processor reads the vector and places into the GPR indicated by the load instruction being executed. The programmer then uses the vector as an index into the interrupt service routine jump table.

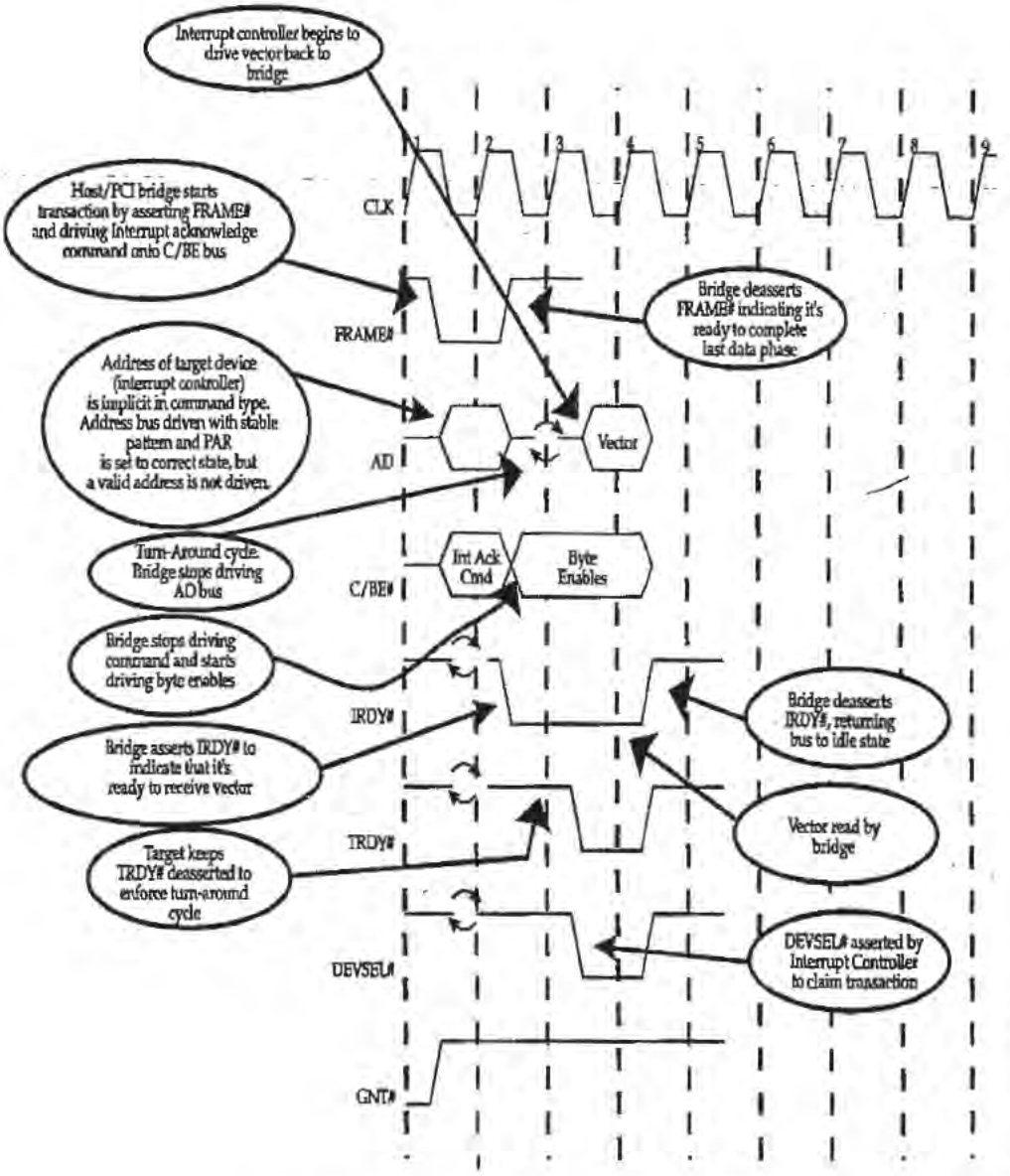


Figure 7-1: The PCI Interrupt Acknowledge Transaction

Special Cycle Command

General

The special cycle command is issued by an initiator to broadcast a message to one or more targets residing on a target PCI bus. Each target on the PCI bus must examine the message to determine whether the message applies to it (a target may be designed not to recognize any messages or to recognize only specific messages). Via its configuration command register, a target's ability to monitor special cycle messages can be enabled or disabled. As an example of message passing using the special cycle, Intel x86 processors use the special cycle to indicate when they are going into a halt or shutdown condition.

During the address phase, a valid address is not driven onto the AD bus. The AD bus and PAR must be driven with a stable pattern, however, so that the parity of the AD bus and the command can be checked for correctness. The initiator uses the C/BE bus to indicate that this is a special cycle transaction.

During the data phase, the initiator broadcasts the message type on AD[15:0] and an optional, message-dependent data field may be presented on AD[31:16]. The message and associated data are only valid during the clock when IRDY# is asserted. The data contained in, and the timing of subsequent data phases is message dependent (the subject of multiple data phase special cycles is discussed under the section entitled "Special Cycle Transaction"). If necessary, the initiator may insert wait states into the transaction by deasserting IRDY#, but targets cannot insert wait states. In addition, no target should assert DEVSEL# when it recognizes a message. Since multiple targets can recognize the message type, there would be contention on the DEVSEL# line if they all tried to claim the transaction by asserting DEVSEL#. The targets must watch IRDY# to determine the presence of the message being sent by the initiator. It should be noted that the message type (and any associated data on AD[31:16]) is only valid during the first data phase. Since all special cycles are intended to pass messages only to PCI targets, a subtractive decode bridge should not pass the transaction onto an expansion bus (such as ISA, EISA or the Micro Channel™) when it doesn't see any target claim the transaction by asserting DEVSEL#.

Since no target responds to the special cycle (DEVSEL# is not asserted), another means must be used to end the transaction. The initiator must perform a master-abort to end the transaction. The master-abort process is explained in

the chapter entitled "Premature Transaction Termination." It must be noted that when the initiator terminates the transaction with a master-abort (because DEVSEL# was not asserted by a target), it must not set the MASTER-ABORT DETECTED bit in its configuration status register. That bit should only be set in a transaction where a DEVSEL# is expected but not received.

Table 7-2 provides the message types currently defined in the specification. The first two message codes, 0000h and 0001h, are defined as SHUTDOWN and HALT. Message code 0002h is reserved for use by Intel x86 processors to broadcast x86-specific messages. During the data phase, AD[15:0] would contain 0002h, while AD[31:16] would contain the x86-specific message. The x86-specific message codes are defined by Intel in product-specific documentation. Message codes 0003h through FFFFh are reserved for future use. Allocation of new message codes is handled through the SIG and requests for allocation of new message codes should be submitted to the SIG in writing.

During system design, each PCI device that is capable of recognizing or broadcasting message codes must be hardwired with the message codes it recognizes or broadcasts. Upon recognition of any of its assigned message codes, a PCI target should take the application-specific action defined by the message code received.

Table 7-2. Message Types defined In the Specification

Message Code (on AD[15:0])	Message Type
0000h	**Shutdown.** Processor is going into a shutdown condition due to a severe, unrecoverable software problem.
0001h	**Halt.** The processor has fetched and is executing a Halt instruction. In response, the processor issues the halt message using the special bus cycle to indicate to all external devices that it is going to cease fetching and executing instructions.
0002h	**x86-specific message.** AD[31:16] contains the Intel device-specific message.
0003h-FFFFh	**Reserved.**

The special cycle command takes a minimum of six clocks to complete (more if the initiator inserts wait states by delaying the assertion of IRDY#). One additional clock is required for the turn-around cycle before the next transaction is initiated on the bus. Therefore, a total of seven clock cycles are required from the start of the special cycle to the start of the next cycle.

Special Cycle Generation

Host/PCI bridges are not required to provide a mechanism that permits special cycles to be generated under software control. If the bridge does provide this capability, however, a detailed description of a mechanism can be found in the chapters entitled "Configuration Transactions" and "PCI-to-PCI Bridge."

Special Cycle Transaction

Single-Data Phase Special Cycle Transaction

Figure 7-2 illustrates the special cycle transaction timing. During the address phase, the initiator drives a stable pattern onto the AD bus and PAR. This is only for parity checking purposes. No actual address is driven. In addition, the initiator drives the special cycle command onto the C/BE bus during the address phase.

At the end of the address phase, the data phase begins. The initiator drives the message code onto AD[15:0] and any optional, message-related data onto AD[31:16]. It also asserts the appropriate byte enable lines (i.e., C/BE#[1:0] or [3:0]). The message is only guaranteed to be present on the AD bus for one clock when the initiator asserts IRDY#. The initiator can insert wait states into the data phase by delaying the assertion of IRDY#. When the message is driven onto the AD bus, the initiator asserts IRDY# to indicate its presence. The targets that are designed to recognize special cycles latch the message information from the AD bus when IRDY# is sampled asserted.

Since a target is not expected to claim a special transaction, DEVSEL# is sampled deasserted (by the initiator) at the end of clocks three through six. Since the transaction isn't claimed on any of these clocks, the initiator executes a master-abort to return the bus back to the idle state. If the master inserted one or more wait states before presenting the message and asserting IRDY#, the master must extend the master abort timeout period by at least the number of wait states inserted (before performing the master abort to return the bus to the idle state). The specification states that this time period is required to give the target(s) sufficient time to "process" the message. This period of time is necessary to ensure that the target(s) have completed all internal actions related to reception of the message and are prepared to handle another transac-

tion. When it occurs, the master abort is accomplished by deasserting FRAME# and then IRDY#.

Multiple Data Phase Special Cycle Transaction

It is permissible for an initiator to deliver multiple packets of message information during the special cycle. No messages are currently defined that provide this capability, however. The target(s) latch the first message packet on the rising-edge of the clock when IRDY# is first sampled asserted. The message type encoded on AD[15:0] may imply the number of additional message packets to be delivered or the data field encoded on AD[31:16] may state the number of packets. The second data phase start during the clock cell immediately following the first assertion of IRDY#. Although the specification doesn't clearly state so, the author interprets the specification as indirectly stating that the initiator can deassert IRDY# during the second (and any subsequent) data phase until it has placed the next message packet on the AD bus. Each additional data phase completes when IRDY# is sampled asserted. When the final data transfer completes, the initiator must keep IRDY# asserted for at least four additional clocks before performing a master abort to return the bus to the idle state. This time period is required to give the target(s) sufficient time to "process" the message. The specification does not explain what form this "processing" might take.

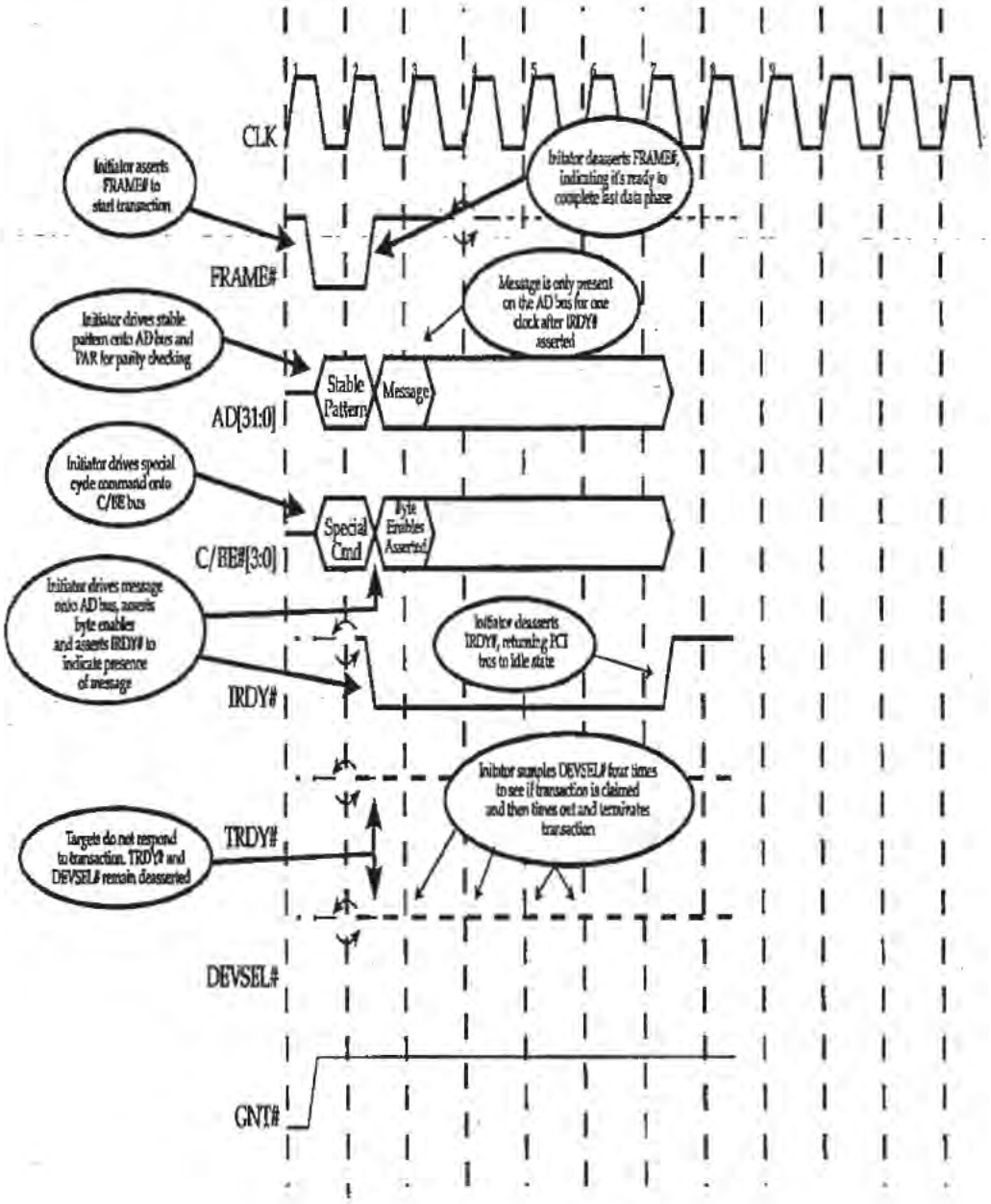


Figure 7-2. The Special Cycle Transaction

I/O Read and Write Commands

The I/O read and write commands are used to transfer data between the initiator and the currently-addressed I/O target. The target must decode the entire 32-bit address. For a detailed description of I/O addressing and I/O read and write transactions, refer to the chapter entitled "The Read and Write Transfers."

Accessing Memory

The PCI specification defines five commands utilized to access memory:

- Memory read command.
- Memory read line command.
- Memory read multiple command.
- Memory write command.
- Memory write and invalidate command.

The specification says that the cache line size configuration register (described in the chapter entitled "Configuration Registers") must be implemented by bus masters that utilize the memory write and invalidate command (described later in this chapter). It also strongly recommends that this register be implemented for bus masters that utilize the memory read, memory read line and memory read multiple commands.

If the cache line size configuration register is implemented, the initiator should follow the usage guidelines outlined in table 7-3 when performing memory reads. If an initiator accesses memory and does not implement the cache line size configuration register, it should follow the guidelines outlined in table 7-4 when performing memory reads. In essence, the rules are the same, but the bus master assumes a cache line size of 16 or 32 bytes.

The specification strongly recommends that the bulk read/write commands be used when transferring large blocks of data to or from memory. These commands are memory write and invalidate, memory read line and memory read multiple.

Table 7-3. Read Policy When Cache Line Size Register Implemented

Read Command Type	To Be Used When
Memory Read	Bursting less than a cache line.
Memory Read Line	Bursting a cache line.
Memory Read Multiple	Bursting more than one cache line.

Table 7-4. Read Policy When Cache Line Size Register Not Implemented

Read Command Type	To Be Used When
Memory Read	Bursting less than a cache line (assuming a cache line size of 16 or 32 bytes).
Memory Read Line	Bursting a cache line (assuming a cache line size of 16 or 32 bytes).
Memory Read Multiple	Bursting more than one cache line (assuming a cache line size of 16 or 32 bytes).

Reading Memory

The following three commands are available to be used when reading data from memory.

Memory Read Command

The memory read command should be used when transferring less than a cache line.

Memory Read Line Command

When a master uses the memory read line command, it is indicating that it will read a complete cache line from the target memory device. This permits the memory target to prefetch the entire line from its memory rather than accessing memory on a data phase by data phase basis. The intent is to yield better performance when performing bulk reads from memory. A memory target that doesn't implement this command will treat it as a memory read and access its memory on a data phase by data phase basis.

Memory Read Multiple Command

When a master uses the memory read multiple command, it is indicating that it will read more than one complete cache line from the target memory device. This permits the memory target to prefetch data from its memory a line at a time rather than accessing memory on a data phase by data phase basis. The

intent is to yield better performance when performing bulk reads from memory. A memory target that doesn't implement this command will treat it as a memory read and access its memory on a data phase by data phase basis.

When this command is used, the target memory device should fetch the requested cache line from memory. When the requested line has been fetched from memory, the memory controller should start fetching the next line from memory in anticipation of a request from the initiator. The memory controller should continue to prefetch lines from memory as long as the initiator keeps FRAME# asserted. It should be noted that the memory target is responsible for ensuring the validity of data prefetched from memory during an anticipatory read.

Writing Memory

The initiator may use the memory write or the memory write and invalidate command to update data in memory.

Memory Write Command

This command is used to transfer one or more data objects to memory. When the target asserts TRDY#, it has assumed responsibility for maintaining the coherency of the data. This can be done by ensuring that any software-transparent posting buffer is flushed prior to synchronization events such as interrupts, or the updating of an I/O status register or memory flag being passed through the device that contains the posted-write buffer (i.e., a bridge).

Memory Write and Invalidate Command

Problem

Assume that another PCI master is performing a memory write and the processor's write back cache(s) is snooping the transaction. It experiences a snoop hit on a modified line. This means that the initiator is about to update a stale line in memory. Assuming that the cache is not capable of data snarfing (latching the data from the AD bus) to keep the cache line updated, it could invalidate the cache line. This, however, would be a mistake. The fact that the line is marked modified indicates that some or all of the information in the line is more current than the corresponding line in memory. The memory write being performed by the current initiator is updating some item in the memory line. Trashing the line from the cache would quite probably trash some data that is more current than that in the memory line.

If the cache permits the initiator to complete the memory write and then flushes the cache line to memory, the data just written by the initiator is overwritten by the stale data in the cache line. The correct action would be to force the initiator that is attempting the write to get off the bus (abort the transaction). The cache then acquires the bus and performs a memory write to transfer, or flush, the modified cache line to memory. In the cache directory, the cache line is then invalidated because the initiator will subsequently update the memory line immediately after the cache line is flushed to memory. The cache then removes the back off, permitting the initiator to reinitiate the memory write. The memory line now contains the most current data. The cache snoops this transaction as well, but it now results in a cache miss (because the cache line was invalidated after it was deposited in memory). The cache does not interfere in the memory write this time.

Description of Memory Write and Invalidate Command

The memory write and invalidate command is identical to the memory write command except that it guarantees the transfer of a complete cache line (or multiple cache lines) during the current transaction. This implies that the cache line size configuration register must be implemented in the initiator so that it can make the termination that an entire cache line will be written.

If, when snooping, the write-back cache detects a memory write and invalidate initiated and experiences a snoop hit on a modified line, the cache can just invalidate the line and doesn't need to back off the initiator in order to perform the flush to memory. This is possible because the initiator has indicated that it is updating the entire memory line and all of the data in the modified cache line is therefore stale and can be invalidated. This increases performance by eliminating the requirement for the back off and line flush.

It is a requirement that the initiator must assert all of the byte enable signals during each data phase of the memory write and invalidate transaction. It also required that linear addressing be used. For information on the byte enables and on linear addressing, refer to the chapter entitled "The Read and Write Transfer."

More Information On Memory Transfers

For a detailed description of read and write transactions, refer to the chapter entitled "The Read and Write Transfers."

Configuration Read and Write Commands

Each PCI device may implement up to 64 doublewords of configuration registers that are used during system initialization to configure the PCI device for proper operation in the system. To access a PCI agent's configuration registers, a configuration read or write command must be initiated and the agent must sense its IDSEL input asserted during the address phase. IDSEL acts as a chip-select, AD[10:8] select the function within the device and the contents of AD[7:2] (during the address phase) are used to select one of the target's 64 doublewords of configuration space.

The x86 processor family implements two address spaces: memory and I/O. PCI requires the implementation of a third address space: configuration space. The mechanism used to generate configuration transactions is described in the chapter entitled "Configuration Transactions."

Dual-Address Cycle

The initiator uses the dual-address cycle command to indicate that it is using 64-bit addressing. This subject is covered in the chapter entitled "The 64-Bit PCI Extension."

Reserved Bus Commands

Targets must not respond (assert DEVSEL#) to reserved bus commands. This means that use of a reserved bus command will result in the initiator experiencing a master abort.

Chapter 8

The Previous Chapter

The previous chapter introduced the types of commands, or transactions, that an initiator may perform once it has acquired ownership of the PCI bus.

In This Chapter

This chapter provides a detailed description of the basic PCI data transfer, using timing diagrams to illustrate the exact sequence and timing of events during the transfer.

The Next Chapter

The next chapter describes the circumstances under which the initiator or target may need to abort a transaction and the mechanisms provided to accomplish the abort.

Some Basic Rules

The ready signal from the device sourcing the data must be asserted when it is driving valid data onto the data bus. The PCI agent receiving the data can keep its ready line deasserted its ready signal until it is ready to receive the data. Once a device's ready signal is asserted, it must remain so until the end of the current data phase.

An agent may not alter its control line settings once it has indicated that it is ready to complete the current data phase. Once the initiator has asserted IRDY#, it may not change the state of IRDY# or FRAME# regardless of the state of TRDY#. Once a target has asserted TRDY# or STOP#, it may not change TRDY#, STOP# or DEVSEL# until the current data phase completes.

Parity

Parity generation, checking, error reporting and timing is not discussed in this chapter. This subject is covered in detail in the chapter entitled "Error Detection and Handling."

Read Transaction

Description

During the following description of the read transaction, refer to figure 8-1.

Each clock cycle is numbered for easy reference and begins and ends on the rising-edge. It is assumed that the bus master has already arbitrated for and been granted access to the bus. The bus master then must wait for the bus to become idle. This is accomplished by sampling the state of FRAME# and IRDY# on the rising-edge of each clock (along with GNT#). When both are sampled deasserted (clock edge one), the bus is idle and a transaction may be initiated by the bus master.

At the start of clock one, the initiator asserts FRAME#, indicating that the transaction has begun and that a valid start address and command are on the bus. FRAME# must remain asserted until the initiator is ready to complete the last data phase. At the same time that the initiator asserts FRAME#, it drives the start address onto the AD bus and the transaction type onto the Command/Byte Enable lines, C/BE[3:0]#. The address and transaction type are driven onto the bus for the duration of clock one.

A turn-around cycle (i.e., a dead cycle) is required on all signals that may be driven by more than one PCI bus agent. This period is required to avoid a collision when one agent is in the process of turning off its output drivers and another agent begins driving the same signal(s). During clock one, IRDY#, TRDY# and DEVSEL# are not driven (in preparation for takeover by the new initiator and target). They are kept in the deasserted state by keeper resistors on the system board (required system board resource).

At the start of clock two, the initiator ceases driving the AD bus. This will allow the target to take control of the AD bus to drive the first requested data item (between one and four bytes) back to the initiator. During a read, clock two is defined as the turn-around cycle because ownership of the AD bus is

changing from the initiator to the addressed target. **It is the responsibility of the addressed target to keep TRDY# deasserted to enforce this period.**

Also at the start of clock two, the initiator ceases to drive the command onto the Command/Byte Enable lines and uses them to indicate the bytes to be transferred in the currently-addressed doubleword (as well as the data paths to be used during the data transfer). Typically, the initiator will assert all of the byte enables during a read.

The initiator also asserts IRDY# to indicate that it is ready to receive the first data item from the target . Upon asserting IRDY#, the initiator does not deassert FRAME#, thereby indicating that this is not the final data phase of the example transaction. If this were the final data phase, the initiator would assert IRDY# and deassert FRAME# simultaneously to indicate that it is ready to complete the final data phase.

It should be noted that the initiator does not have to assert IRDY# immediately upon entering a data phase. It may require some time before it's ready to receive the first data item (e.g., it has a buffer full condition). However, the initiator may not keep IRDY# deasserted for more than eight PCI clocks during any data phase. This rule has been added in version 2.1 of the specification.

During clock cell three, the target:

- asserts DEVSEL# to indicate that it has recognized its address and will participate in the transaction.
- begins to drive the first data item (between one and four bytes, as requested by the setting of the C/BE lines) onto the AD bus and asserts TRDY# to indicate the presence of the requested data.

When the initiator and the currently-addressed target sample TRDY# and IRDY# both asserted at the rising-edge of clock four, the first data item is read from the bus by the initiator, completing the first data phase. The first data phase consisted of clock cell two and the wait state (turnaround cycle) inserted by the target (clock cell three). At the start of the second data phase (clock edge four), the initiator sets the byte enables to indicate the bytes to be transferred within the next doubleword.

It is a rule that the initiator must immediately output the byte enables for a data phase upon entry to the data phase. If for some reason the initiator

doesn't know what the byte enable setting will be for the next data phase, it should keep IRDY# deasserted and not let the current data phase end until it knows what they will be.

In this example, the initiator keeps IRDY# asserted upon entry into the second data phase, but does not deassert FRAME#. This indicates that the initiator is ready to read the second data item, but this is not the final data phase.

In a multiple-data phase transaction, it is the responsibility of the target to latch the start address into an address counter and to manage the address from data phase to data phase. As an example, upon completion of one data phase, the target would increment the latched address by four to point the next doubleword. It then examines the initiator's byte enable settings to determine the bytes to be transferred within the currently-addressed doubleword. This subject is covered in more detail later in this chapter.

In this example, the target is going to need some time to fetch the second data item requested, so it deasserts TRDY# to insert a wait state (clock cell five) into the second data phase. In order to keep the data paths from floating, the target must continue to drive a stable data pattern, usually consisting of the last data item, onto the AD bus until it has acquired and is presenting the second requested data item. This is illustrated in clock four. It is necessary to keep the AD bus from floating in order to prevent all of the CMOS input buffers connected to the AD bus from oscillating and drawing excessive current. Mentioned earlier in the book, this is one of the measures taken to achieve the green nature of the PCI bus.

At the rising-edge of clock five, the initiator samples TRDY# deasserted and, recognizing that the target is requesting more time for the transfer of the second data item, it inserts a wait state into the second data phase (clock cell five).

During the wait state, the target begins to drive the second data item onto the AD bus and asserts TRDY# to indicate its presence. When the initiator samples both IRDY# and TRDY# asserted at the rising-edge of clock six, it reads the second data item from the bus. This completes the second data phase. The second data phase consisted of clock cells four and five.

At the start of the third data phase, the initiator sets the byte enables to indicate the bytes to be transferred in the next doubleword. It also deasserts

IRDY#, indicating that it requires more than one clock cell before it will be ready to receive the data.

During clock cell six, the target keeps TRDY# asserted, indicating that it is driving the third requested data item onto the AD bus. In this example, however, the initiator requires more time before it will be able to read the data item (probably because it has a temporary buffer full condition). This causes a wait state to be inserted into data phase three. The target must continue to drive the third data item onto the AD bus during the wait state (clock cell seven).

During clock cell seven, the initiator asserts IRDY#, indicating its willingness to accept the third data item on the next rising clock edge. It also deasserts FRAME#, indicating that this is the final data phase. Sampling both IRDY# and TRDY# asserted at the rising-edge of clock eight, the initiator reads the third data item from the bus. The third data phase consisted of clocks six and seven. Sampling FRAME# deasserted instructs the target that this is the final data item.

The overall burst transfer consisting of three data phases has been completed. The initiator deasserts IRDY#, returning the bus to the idle state (on the rising-edge of clock nine), and the target deasserts TRDY# and DEVSEL#.

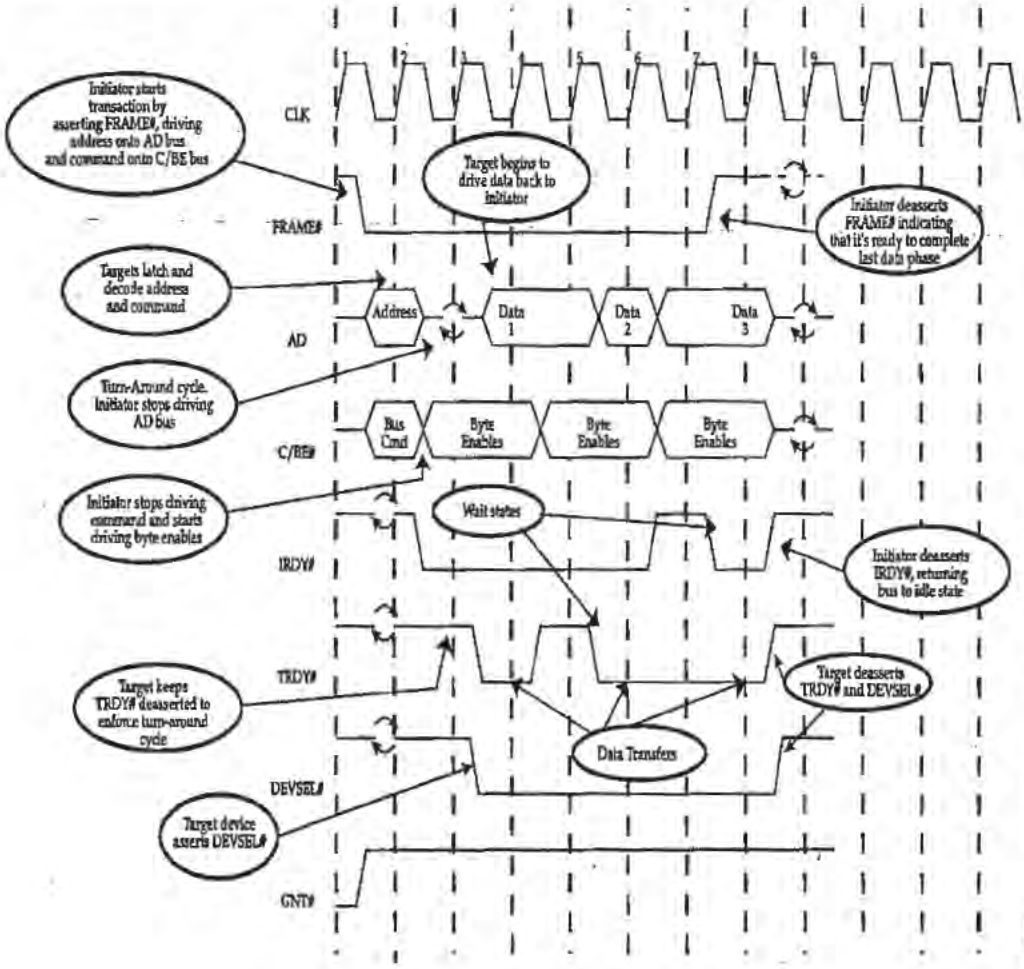


Figure 8-1. The Read Transaction

Treatment of Byte Enables During Read or Write

Byte Enable Settings May Vary from Data Phase to Data Phase

PCI permits burst transactions where the byte enables change from one data phase to the next. Furthermore, the initiator may use any byte enable setting consisting of contiguous or non-contiguous byte enables. During a read trans-

action, the initiator will typically assert all of the byte enables during each data phase, but it may use any combination.

It should be noted that all targets may not be capable of handling non-contiguous byte enables. An example would be an PCI/ISA bridge. In this case, the target could take one of the following actions:

- assert SERR#.
- break the transaction into two 16-bit transfers.

Data Phase with No Byte Enables Asserted

As stated in the previous paragraph, any combination of byte enables is valid in any data phase. This includes a data phase with no byte enables asserted (a null data phase). This can occur for a number of reasons. Some examples would be:

- During a burst transfer, the **programmer may wish to "skip" a double-word.** This would be accomplished by keeping all byte enables deasserted during that data phase.
- At the initiation of a **64-bit transfer**, the initiator does not yet know whether the target device is a 64 or a 32-bit device. In certain cases, if a 32-bit device responds, this can result in the first data phase being null. This case is described in the chapter entitled "The 64-bit PCI Extension."
- There are cases where the **last data phase** of a block transfer may not have any of the byte enables asserted. Assume that an expansion bus master (EISA or Micro Channel™) has initiated a series of accesses with a PCI target. The bridge between the expansion and PCI buses will frequently packetize this series of bus master accesses into a PCI burst transfer. When the expansion bus master has completed its last data transfer, the bridge signals this to the target by deasserting FRAME#. This informs the target that the last data transfer is in progress. Since the bus master has already transferred all of the data, however, the bridge will not assert any of the byte enables during this last data phase.

When none of the byte enables are asserted, the target must react as follows:

- **On a read:** the target must ensure that no data or status is destroyed or altered as a result of this data transfer. The target must supply a stable pattern on all data paths and must generate the proper parity (for the AD and C/BE buses) on the PAR bit.

- On a write: the target must not store any data and the initiator must supply a stable pattern on all data paths and ensure that PAR is valid for the AD and C/BE buses.

Target with Limited Byte Enable Support

I/O and memory targets may support restricted byte enable settings and may respond with target abort for any other pattern. All devices must support any byte enable combination during configuration transactions.

Rule for Sampling of Byte Enables

If the target requires sampling of the byte enables (in order to precisely determine which bytes are to be transferred within the currently-addressed doubleword) during each data transfer, it must wait for the byte enables to be valid during each data phase before completing the transfer. An example of a device that requires sampling of byte enables would be a memory-mapped I/O device. It should not accept a write to or a read from 8-bit ports within the currently-addressed doubleword until it has verified (via the byte enables) that the initiator is in fact addressing those ports.

If a target does not require examination of the byte enables on a read, the target must supply all four bytes. An example of a device that would not have to wait to sample the byte enables would be a typical memory target. Memory typically yields the same data from a location no matter how many times the location is read from. In other words, performing a speculative read from the memory does not alter the data stored in the location. This type of memory target can be designed to supply all four bytes in every data phase of a read burst. The initiator only take the bytes it's addressing and ignores the others.

Ignore Byte Enables During Line Read

If the initiator is reading a line of data from memory, the memory target must return all four bytes regardless of the byte enable settings. This action is guaranteed in one of the following manners:

- If the cacheability of the target memory is determined by the initiator, the initiator must ensure that all byte enables are asserted so that the target will return all four bytes.

- If the target memory determines that the access is cacheable, it should ignore the byte enable settings during each data phase (except for parity generation) and return all four bytes.

Prefetching

If a target does not support caching but does support prefetching (indicated by hardwiring the PREFETCHABLE attribute bit in its base address configuration register to a one), it must return all four bytes (on a read) regardless of the byte enable settings. A target only supports this feature if there are no side effects from the read (for example, data destroyed or status change in a memory-mapped I/O register).

Performance During Read Transactions

As described earlier, a turn-around cycle must be included in the first data transfer of a read transaction. This being the case, a single data phase read from a target consists of at least three cycles of the PCI clock (one clock cell for the address phase and two clock cells for the data phase). At a clock rate of 33MHz, a read transaction consisting of a single data transfer would take 90ns to complete. An idle cycle (at 33MHz, 30ns in duration) must be included between transactions, resulting in 120ns per transaction. Using back-to-back single data phase read transfers, the data throughput would be 8.33 million transfers per second. If each transfer involved four bytes, the resultant transfer rate would be 33.33Mbytes per second.

In actual practice, though, most read transactions involve the transfer of multiple objects between the initiator and the currently-addressed target. The read transaction involving multiple data phases only requires the turn-around cycle during the first data phase. The second through the last data phases can each be accomplished in a single clock cycle (if both the initiator and the currently-addressed target are capable of zero wait state transfers). The achievable transfer rate during the second through the last data phases is thus one transfer every 30ns (at a PCI bus speed of 33MHz), or 33 million transfers per second. If each data phase involves the transfer of four bytes, the resultant data transfer rate is 132Mbytes per second. Figure 8-2 illustrates a read transaction consisting of three data phases, two of which complete with zero wait states.

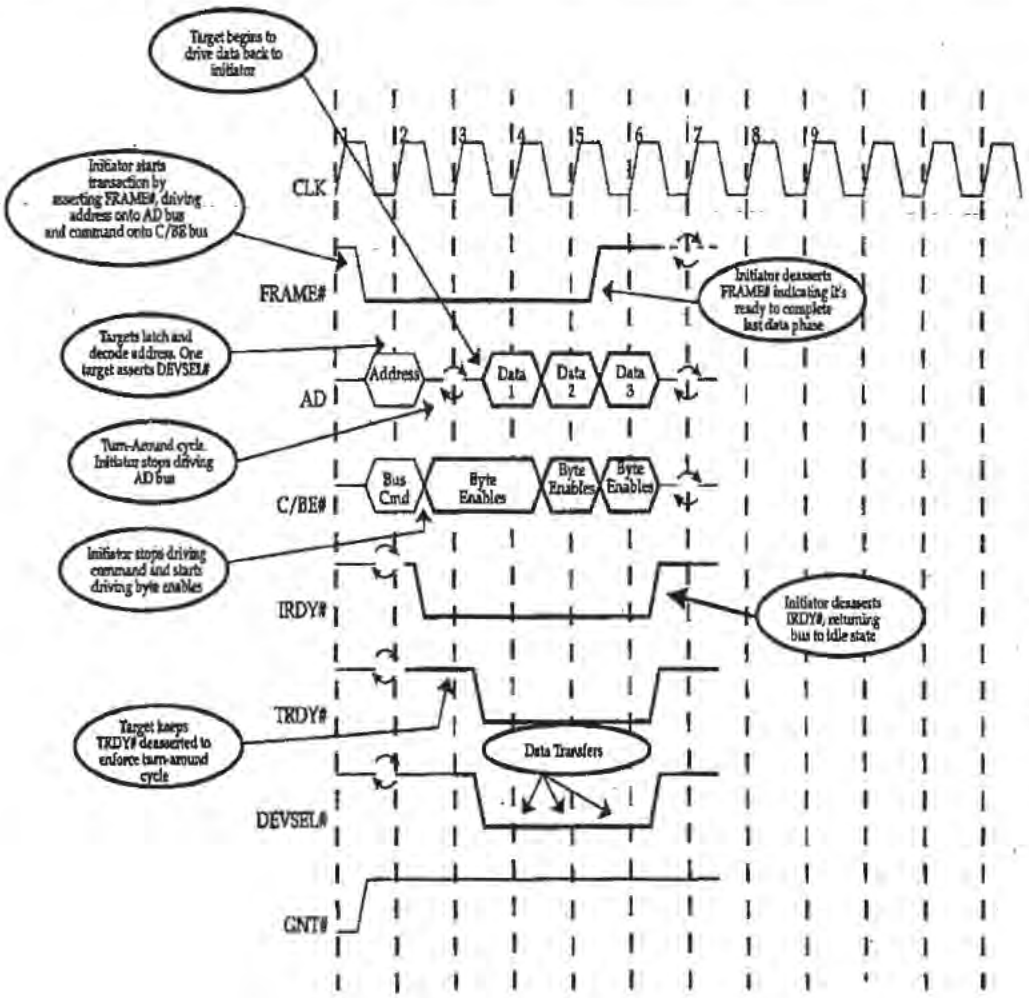


Figure 8-2. Optimized Read Transaction (no wait states)

Write Transaction

Description

During the following description of the write transaction, refer to figure 8-3.

It is assumed that the bus master has already arbitrated for and been granted access to the bus. The bus master then must wait for the bus to become idle. This is accomplished by sampling the state of FRAME# and IRDY# on the rising-edge of each clock. When both are sampled deasserted (on the rising-edge of clock one), the bus is idle and a transaction may be initiated by the bus master whose grant signal is currently asserted by the bus arbiter.

At the start of clock cell one, the initiator asserts FRAME# to indicate that the transaction has begun and that a valid start address and command are present on the bus. FRAME# remains asserted until the initiator is ready to complete the last data phase. At the same time that the initiator asserts FRAME#, it drives the start address onto the AD bus and the transaction type onto the Command/Byte Enable bus. The address and transaction type are driven onto the bus for the duration of clock one.

A turn-around cycle is required on all signals that may be driven by more than one PCI bus agent. This period is required to avoid the collision that would occur if a device turned on its output drivers at the same time that another device's output drivers are disconnecting from the signal(s). During clock cell one, IRDY#, TRDY# and DEVSEL# are not driven (in preparation for takeover by the new initiator and target).

At the start of clock cell two, the initiator changes the information that it is presenting to the target over the AD bus. During a write transaction, the initiator is driving the AD bus during both the address and data phases. Since it doesn't have to hand off control of the AD bus to the target, as it does during a read, a turn-around cycle is unnecessary. The initiator may begin to drive the first data item onto the AD bus at the start of clock cell two. In addition, during clock cell two the initiator uses the Command/Byte Enable lines to indicate the bytes to be transferred to the currently-addressed doubleword and the data paths to be used during the first data phase.

At the start of clock cell two, the initiator drives the write data onto the AD bus and asserts the respective byte enables to indicate the data paths that

carry valid data. It also asserts IRDY# to indicate the presence of the data on the bus. The initiator doesn't deassert FRAME# when it asserts IRDY# (because this is not the final data phase).

It should be noted that the initiator does not have to assert IRDY# immediately upon entering a data phase. It may require some time before it's ready to source the first data item (e.g., it has a buffer empty condition). However, the initiator may not keep IRDY# deasserted for more than eight PCI clocks during any data phase. **This rule has been added in version 2.1 of the specification.**

During clock cell two, the target decodes the address and command and asserts DEVSEL# to claim the transaction. In addition, it asserts TRDY#, indicating its readiness to accept the first data item.

At the rising-edge of clock three, the initiator and the currently-addressed target sample both TRDY# and IRDY# asserted, indicating that they are both ready to complete the first data phase. This is a zero wait state transfer. The target accepts the first data item from the bus on the rising-edge of clock three (and samples the byte enables in order to determine which bytes are being written), completing the first data phase.

During clock cell three, the initiator drives the second data item onto the AD bus and sets the byte enables to indicate the bytes to be transferred and the data paths to be used during the second data phase. It also keeps IRDY# asserted and does not deassert FRAME#, thereby indicating that it is ready to complete the second data phase and that this is not the final data phase. Assertion of IRDY# indicates that the write data is present on the bus.

At the rising-edge of clock four, the initiator and the currently-addressed target sample both TRDY# and IRDY# asserted, indicating that they are both ready to complete the second data phase. This is a zero wait state data phase. The target accepts the second data item from the bus on the rising-edge of clock four (and samples the byte enables), completing the second data phase.

The initiator requires more time before beginning to drive the next data item onto the AD bus (it has a buffer empty condition). It inserts a wait state into the third data phase by deasserting IRDY# at the start of clock cell four. This allows the initiator to delay presentation of the new data by one clock, but it must set the byte enables to the proper setting for the third data phase at the start of clock cell four.

In this example, the target also requires more time before it will be ready to accept the third data item. To indicate the requirement for more time, the target deasserts TRDY# during clock cell four. When the initiator and target sample IRDY# and TRDY# deasserted at the rising-edge of clock five, they insert a wait state (clock cell five) into the third data phase.

During clock cell four, although the initiator does yet have the third data item available to drive, it must drive a stable pattern onto the data paths rather than let the AD bus float (remember the rule about PCI being green). The specification doesn't dictate the pattern to be driven during this period. It is usually accomplished by continuing to drive the previous data item. The target will not accept the data being presented to it for two reasons:

- By deasserting TRDY#, it has indicated that it isn't ready to accept data.
- By deasserting IRDY#, the initiator has indicated that it is not yet presenting the next data item to the target.

During clock cell five, the initiator asserts IRDY# and drives the final data item onto the AD bus. It also deasserts FRAME# to indicate that this is the final data phase. The target keeps TRDY# deasserted, indicating that it is not yet ready to accept the third data item.

At the rising-edge of clock six, the initiator samples IRDY# asserted, indicating that it is presenting the data, but TRDY# is still deasserted (because the target is not yet ready to accept the data item). The target also samples FRAME# deasserted, indicating that the final data phase is in progress. The only thing impeding the completion of the final data phase now is the target (by keeping TRDY# deasserted until it is ready to accept the final data item).

In response to sampling TRDY# deasserted on clock edge six, the target and initiator insert a second wait state (clock cell six) into the third data phase. During the second wait state, the initiator continues to drive the third data item onto the AD bus and maintains the setting on the byte enables. The target keeps TRDY# deasserted, indicating that is not ready yet.

At the rising-edge of clock seven, the target and initiator sample IRDY# asserted, indicating that the initiator is still presenting the data, but TRDY# is still deasserted. In response, the target and initiator insert a third wait state (clock cell seven) into the third data phase. During the third wait state, the initiator continues to drive the third data item onto the AD bus and maintains

the setting on the byte enables. The target asserts TRDY#, indicating that it is ready to complete the final data phase.

At the rising-edge of clock eight, the target and initiator sample both IRDY# and TRDY# asserted, indicating that both the initiator and the target are ready to end the third and final data phase. In response, the third data phase is completed on the rising-edge of clock eight. The target accepts the third data item from the AD bus. The third data phase consisted of four clock periods (the first clock cell of the data phase, clock cell four, plus three wait states).

During clock cell eight, the initiator ceases to drive the data onto the AD bus, stops driving the C/BE bus, and deasserts IRDY# (returning the bus to the idle state). The target deasserts TRDY# and DEVSEL#.

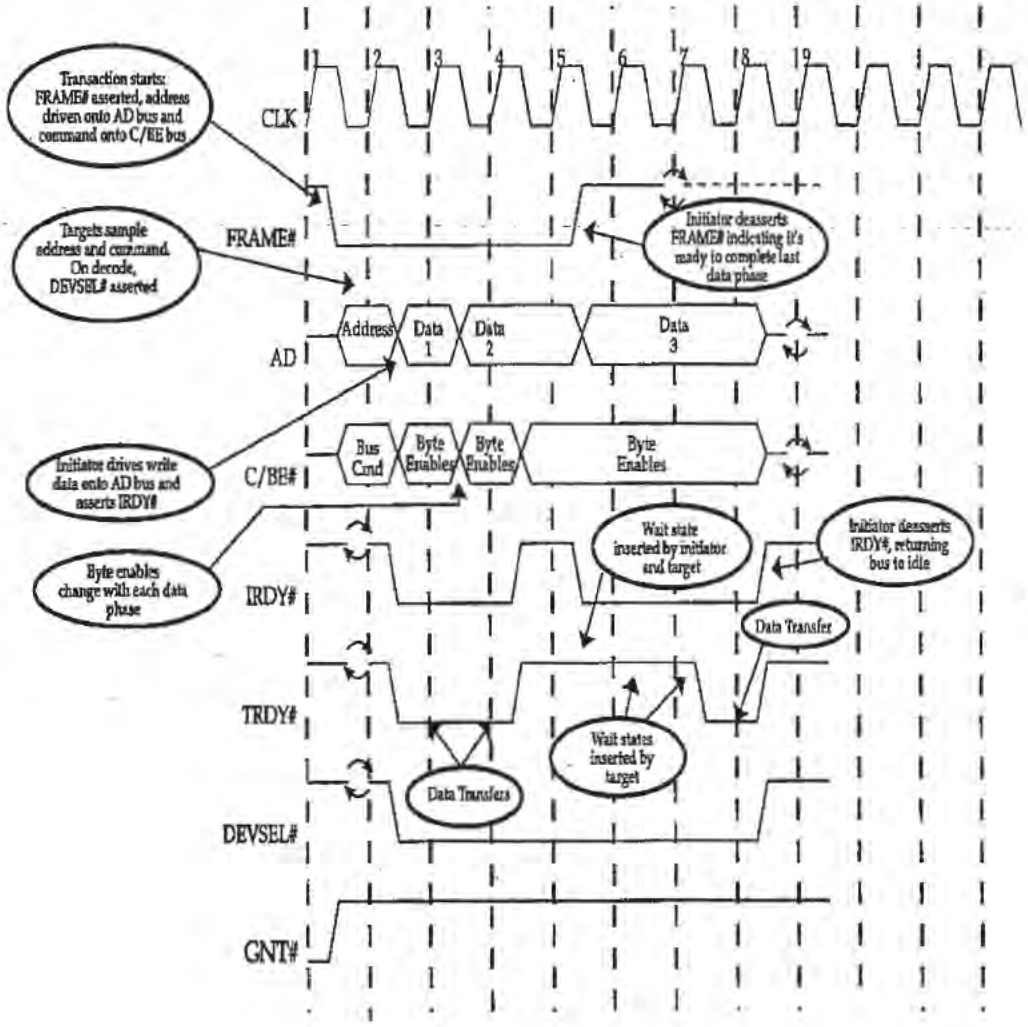


Figure 8-3. The PCI Write Transaction

Performance During Write Transactions

Transactions wherein an initiator performs a single data phase write to a target consist of at least two cycles of the PCI clock (the address phase and a one clock data phase). An idle cycle (at 33MHz, 30ns in duration) must be included between transactions. At a clock rate of 33MHz, then, a single data phase write transaction takes 90ns to complete. Using back-to-back single data phase write transfers, the data throughput would be 11.11 million transfers per second. If each transfer involved four bytes, the resulting transfer rate would be 44.44Mbytes per second.

The second through the last data transfer of a write transaction involving multiple data phases can each be accomplished in a single clock cycle (if both the initiator and the currently-addressed target are capable of zero wait state data phases). The achievable transaction rate during the second through the last data phases is thus one transaction every 30ns (at a PCI bus speed of 33MHz), or 33 million transfers per second. If each transfer involves the transfer of four bytes, the data transfer rate is 132Mbytes per second. Figure 8-4 illustrates a write transaction consisting of three zero wait state data phases.

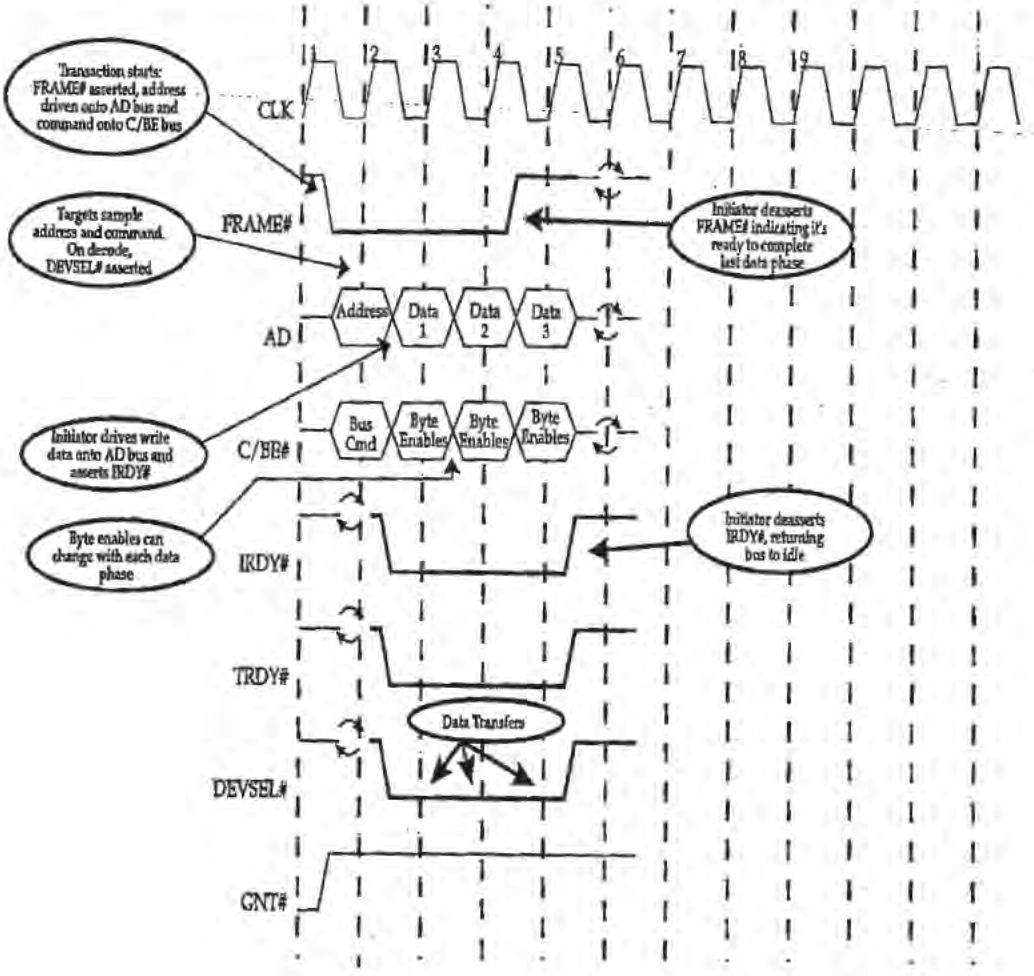


Figure 8-4. Optimized Write Transaction (no wait states)

Posted-Write Buffer

General

A bridge (PCI-to-PCI bridge or host/PCI) may incorporate a posted-write buffer that allows a bus master to complete a memory write quickly. The transaction and the write data are latched within the bridge's posted-write buffer and the master is permitted to complete the transaction. When a bridge implements a posted-write buffer, a potential problem exists. Another bus master (or the same one) may initiate a memory read from the target of the posted write before the data is actually written to the memory target. If this were permitted, the master performing the read would not receive the freshest copy of the information. In order to prevent this from occurring, the bridge designer must first flush all posted writes to their destination memory targets before permitting a read to occur on the bus. A device driver can ensure that all memory data has been written to its device by performing a read from the device. This will force the flushing of all posted write buffers in bridges that reside between the processor executing the read and the target device before the read is permitted to complete.

It is also a requirement that the bridge must perform all posted writes in the same order in which they were originally posted.

A bridge is only permitted to post writes to regular memory targets. Software must be assured real-time communication with I/O and memory-mapped I/O devices, as well as with configuration registers.

Combining

A bridge may combine posted memory writes to successive doublewords into a single burst memory write transaction using linear addressing. This feature is recommended to improve performance. The doublewords must be written in the same order in which they were posted. This means that writes posted to doublewords 0, 1 and 2 (they were posted in that order) can be combined into a linear burst write, while writes posted to doublewords 2, 1, 0 cannot. Instead, these three writes would have to be performed as three separate single data phase memory write transactions. Writes posted to doublewords 0, 1, and 3 (in that order) can be combined into a linear burst write with no byte enables asserted in the third data phase. The specification recommends that bridges that permit combining include a control bit to allow this feature to be disabled.

Byte Merging

A bridge may combine writes to a single doubleword to be merged within one entry in the posted-write buffer. This feature is recommended to improve performance and is only permitted in memory address range that are prefetchable (for more information on prefetchable memory, refer to the base address register section in the chapter entitled "Configuration Registers" and to the chapter entitled "PCI-to-PCI Bridge." As an example, assume that a bus master performs two memory writes: the first writes to locations 00000100h and 00000101h and the second writes to locations 00000102h and 00000103h. These four locations reside within the same doubleword. The bridge could absorb the first two-byte write into a doubleword buffer entry and then absorb the second two byte write into the same doubleword buffer entry. When the bridge performs the memory write, it can complete it in a single data phase. It is a violation of the specification, however, for a bridge to combine separate byte writes to the same location into a single write on the destination bus. As an example, assume that a bus master performs four separate memory writes to the same doubleword: the first writes to location zero in the doubleword, the second to location zero again, the third to location one and the fourth to location two. When the bridge performs the posted writes, it has to perform a single data phase transaction to write the first byte to location zero. It then performs a second single data phase memory write to locations zero (the second byte written to it by the bus master), one and two.

Collapsing

Multiple writes to the same location(s) cannot be performed as a single write on the other side of the bridge. Two sequential writes to the same doubleword where at least one of the byte enables was asserted in both transactions must be performed as two separate transactions on the other bus. Collapsing of writes is forbidden for any type of write transactions.

The specification states that a bridge may allow collapsing within a specific range when a device driver indicates that this will not cause operational problems. How the device driver would indicate this to a bridge is outside the scope of the specification.

Cache Line Merging

The bridge may perform cache line merging within an area of memory that the bridge knows is cacheable or when it uses combining and/or byte merging

to create a burst write of a cache line. It captures (i.e., it posts) individual memory writes performed by bus masters on one PCI bus to build a cache line to be written on the other bus using a memory write and invalidate transaction or a linear memory write transaction. The author would like to note that the specification doesn't specifically state that a memory write and invalidate command would be used.

Addressing Sequence During Memory Burst

Linear and Cacheline Wrap Addressing

The start address issued during any form of memory transaction is a doubleword-aligned address presented on AD[31:2] during the address phase. The memory target latches this address into an address counter and uses it for the first data phase. Upon completion of the first data phase and assuming that it's not a single data phase transaction, the memory target must update its address counter to point to the next doubleword to be transferred.

On a memory access, a memory target must check the state of address bits one and zero (AD[1:0]) to determine the policy to use when updating its address counter at the conclusion of each data phase. Table 8-1 defines the addressing sequences defined in the revision 2.1 specification and encoded in the first two address bits. Only two addressing sequences are currently defined:

- **Linear, or sequential, address mode.** All memory devices that support multiple data phase transfers must implement support of linear, or sequential, addressing. The memory write and invalidate command must use linear addressing. At the completion of each data phase, the memory target increments its address counter by four to point to the next sequential doubleword for the next data phase.
- **Cacheline wrap mode.** Support for cacheline wrap mode is optional and is only used for memory reads. At the start of each data phase of the burst read, the memory target increments the doubleword address in its address counter. When the end of the cache line is encountered and assuming that the transfer did not start at the first doubleword of the cache line, the target wraps to start address of the cacheline and continues incrementing the address in each data phase until the entire cache line has been transferred. If the burst continues past the point where the entire cache line has been transferred, the target starts the transfer of the next cache line at the same address that the transfer of the previous line started at.

Implementation of the cacheline wrap mode is optional for memory and meaningless for I/O and configuration targets. The addressing sequence used during a cache line fill is established at the start of the transfer based on the start memory address and the length of the transfer. This implies that the memory target must know that a cache line fill is in progress (wrap mode indicated) and the size of a cache line (established at startup when the platform-specific configuration program writes the system cache line size to the memory target's cache line-size configuration register).

The 486 processor's internal cache has a line size of sixteen bytes (four doublewords) and has a 32-bit data bus. It must therefore perform four 32-bit transfers to fill a cache line. The first doubleword address output by the processor is the one that resulted in an internal cache miss. This could be any of the four doublewords within the line. For a detailed description of the 486 cache line fill addressing sequence, refer to the Addison-Wesley publication entitled *80486 System Architecture*. For that used by the Pentium processor, refer to the Addison-Wesley publication entitled *Pentium Processor System Architecture*. For that used by the PowerPC 60x processors, refer to the Addison-Wesley publication entitled *PowerPC System Architecture*.

As an example, assume that the cache line size is 16 bytes and the start doubleword address issued by the master is 00000104h. This doubleword resides within the 16-byte aligned cache line that occupies memory locations 00000100h through 0000010Fh. The sequence of the doubleword transfers would be 00000104h, 00000108h, 0000010Ch and 00000100h. If the burst continues past this point, the next series of doublewords transferred would be 00000114h, 00000118h, 0000011Ch and 00000110h.

If the target does not implement the cache line size register, the target must issue a disconnect on the first data phase or a retry on the second one (it can't handle wrap mode because it doesn't know the line size).

If the master wants to use a different sequence after the first line has been read, it must end the transaction and begin a new one indicating linear addressing.

Table 8-1. Memory Burst Address Sequence

AD1	AD0	Addressing Sequence
0	0	**Linear**, or sequential, addressing sequence during the burst.
0	1	**Cacheline wrap** mode.
1	0	**Reserved.** When detected, the memory target should signal a target disconnect after the first data phase or a retry on the second data phase.
1	1	**Reserved.** When detected, the memory target should signal a target disconnect after the first data phase or a retry on the second data phase.

Target Response to Reserved Setting on AD[1:0]

Assuming that the initiator has started a multi-data phase memory transaction and that it has placed a reserved pattern on AD[1:0] in the address phase (10b or 11b pattern), the revision 2.x-compliant memory target must either issue a disconnect on the transfer of the first data item, or a retry during the second data phase. This is necessary because the initiator is indicating an addressing sequence the target is unfamiliar with (because it is reserved in the revision 2.1 specification).

Do Not Merge Processor I/O Writes into Single Burst

To ensure that I/O devices function correctly, bridges must never combine sequential I/O accesses into a single (merging byte accesses performed by the processor into a single-doubleword transfer) or a multi-data phase transaction. Each individual I/O transaction generated by the host processor must be performed on the PCI bus as it appears on the host bus. This rule includes both regular and memory-mapped I/O accesses.

PCI I/O Addressing

General

The start I/O address placed on the AD bus during the address phase has the following format:

- AD[31:2] identify the target doubleword of I/O space.
- AD[1:0] identify the least-significant byte within the target doubleword that the initiator wishes to perform a transfer with (00b = byte 0, 01b = byte 1, etc.).

At the end of the address phase, all I/O targets latch the start address and the I/O read or write command and begin the address decode. An I/O target claims the transaction based on the byte-specific start address that it latched. If that 8-bit I/O port is implemented in the target, the target asserts DEVSEL# and claims the transaction. If the target "owns" the entire target doubleword, only AD[31:2] must be decoded to identify the target doubleword and assert DEVSEL#.

The byte enables asserted during the data phase identify the least-significant byte within the doubleword (the same one indicated by the setting of AD[1:0]) as well as any additional bytes (within the addressed doubleword) that the initiator wishes to transfer. It is illegal (and makes no sense) for the initiator to assert any byte enables of lesser significance than the one indicated by the AD[1:0] setting. If the initiator does assert any of illegal byte enable pattern, the target must terminate the transaction with a target abort. Table 8-2 contains some examples of I/O addressing.

Table 8-2. Examples of I/O Addressing

AD[31:0]	C/BE3#	C/BE2#	C/BE1#	C/BE0#	Description
00001000h	1	1	1	0	just location 1000h
000095A2h	0	0	1	1	95A2 and 95A3h
00001510h	0	0	0	0	1510h-1513h
1267AE21h	0	0	0	1	1267AE21h-1267AE23h

Situation Resulting in Target-Abort

If an I/O target claims a transaction (asserts DEVSEL#) based on the byte-specific start address issue during the address phase, then subsequently examines the byte enables (issued during the data phase) and determines that it cannot fulfill the initiator's request, the target must respond by indicating a target-abort (STOP# asserted, TRDY# and DEVSEL# deasserted) to the initiator. The target-abort is covered in the chapter entitled "Premature Transaction Termination." A typical example wherein the target must abort the transaction could result from the following x86 instruction:

```
IN      AX, 60 ;read two bytes from I/O starting at address 60h
```

When executed by an 486 processor, doubleword address 00000060h is driven onto the host bus during the resultant I/O read transaction and the processor asserts BE0# and BE1#, but not BE2# and BE3#. This indicates to the host/PCI bridge that the processor is addressing locations 00000060h and 00000061h within I/O doubleword starting at port 00000060h. Assuming that the host/PCI bridge doesn't incorporate either of these I/O port addresses, it arbitrates for and receives ownership of the PCI bus and initiates an I/O read transaction.

During the address phase, the host/PCI bridge drives the address of the least-significant I/O port to be read by the processor, 00000060h, onto the AD bus. The bridge determines this is the least-significant port to be read by examining the processor's byte enable setting and testing for the least-significant byte enable asserted by the processor. In this case, it is BE0#, corresponding to the first location in the currently-addressed doubleword, 00000060h.

In a PC-compatible machine, this is the address of the keyboard data port. Assuming that the keyboard controller resides on the PCI bus (e.g., embedded within or closely-associated with the PCI/ISA bridge), the keyboard controller would assert DEVSEL# to claim the transaction. Subsequently, when the processor's byte enables are presented during the data phase and are sampled by the target, BE0# and BE1# are asserted. This identifies I/O addresses 60h and 61h as the target locations.

Since port 61h has nothing to do with the keyboard interface (it is system control port B, a general I/O status port on the system board), the keyboard interface cannot service the entire request. It must therefore issue a target-abort to the initiator (STOP# asserted, TRDY# and DEVSEL# deasserted) and terminate the transaction with no data transferred. As a result, the initiator sets its TARGET-ABORT DETECTED status bit and the target sets its SIGNALED TARGET-ABORT status bit (in their respective PCI configuration status registers). The initiator reports this error back to the software in a device-specific fashion (e.g., by generating an interrupt request).

An ISA expansion bus bridge doesn't have specific knowledge regarding all of the I/O ports that exists on the ISA bus. It therefore claims I/O transactions that remain unclaimed by PCI I/O devices. Since it doesn't "know" what I/O ports exists behind it, it can not judge whether to target abort the transaction based on the byte enable settings.

I/O Address Management

As in any PCI read/write transaction, it is the responsibility of the I/O target to latch the start address delivered by the initiator. It then assumes responsibility for managing the address for each subsequent data phase that follows the first data phase. Unlike memory address management, in PCI there is no explicit or implicit I/O address sequencing from one data phase to the next. The initiator and the target must both understand and utilize the same I/O address management. Two examples would be:

- Both the initiator and the target understand that the doubleword address (on AD[31:2]) delivered by the initiator is to be incremented by four at the completion of each data phase. In other words, the read or write transaction proceeds sequentially through the target's I/O address space a doubleword at a time.
- Both the initiator and the target understand that the target doesn't increment the doubleword address for each subsequent data phase. This is how a designer would implement a FIFO port.

At the time of this writing, the author is unaware of any currently-existing processor that is capable of performing burst I/O write transactions. It's easy to assume that the Intel x86 INS (input string) and OUTS (output string) instructions cause the processor to generate a burst I/O read or write series, but this isn't so. When an INS instruction is executed by the x86 processor, it results in a series of back-to-back I/O read and memory write bus cycles. The OUTS instruction results in a string of back-to-back memory read and I/O write bus cycles.

When I/O Target Doesn't Support Multi-Data Phase Transactions

Many PCI I/O targets are not designed to handle multi-data phase transactions. A target can determine that the initiator intends to perform a second data phase upon completion of the first by checking the state of FRAME# when IRDY# is sampled asserted in the first data phase. If IRDY# has been asserted by the initiator and it still has FRAME# asserted, this indicates that this is not the final data phase in the transaction.

If an I/O target doesn't support multi-data phase transactions and the initiator indicates that a second data phase is forthcoming, the target must respond in one of two ways:

- When it's ready to transfer the first data item, **terminate the first data phase with a disconnect** (STOP#, TRDY# and DEVSEL# asserted). The first data item is transferred successfully, but the initiator is forced to terminate the transaction at that point. It must then re-arbitrate for bus ownership and re-address the target using a byte-specific start address within the next I/O doubleword.
- **Terminate the second data phase with a retry** (STOP# and DEVSEL# asserted, TRDY# deasserted). The first data phase completes normally. The initiator is then forced to terminate the transaction during the second data phase without transferring any additional data. The initiator then re-arbitrates for bus ownership and re-addresses the target using a byte-specific start address within the same I/O doubleword.

Address/Data Stepping

Advantages: Diminished Current Drain and Crosstalk

Turning on a large number of signal drivers simultaneously (e.g., driving a 32-bit address onto the AD bus) can result in:

- a large spike of current drain.
- a significant amount of crosstalk within the driver chip and on adjacent external signal lines.

The designer could choose to alleviate both of these problems by turning on the drivers associated with non-adjacent signal drivers in groups over a number of steps, or clock periods.

As an example, assume that the system board designer lays out the 32 AD lines as adjacent signal traces in bit sequential order. By simultaneously driving all 32 lines, crosstalk would be generated on the traces (and within the driver chip). Now assume that there are four 8-bit groups of signal drivers connected as follows:

- driver group one is connected to AD lines 0, 4, 8, 12, 16, 20, 24, 28.
- driver group two is connected to AD lines 1, 5, 9, 13, 17, 21, 25, 29.

- driver group three is connected to AD lines 2, 6, 10, 14, 18, 22, 26, 30.
- driver group four is connected to AD lines 3, 7, 11, 15, 19, 23, 27, 31.

The initiator could turn on the first driver group in clock cell one of a transaction, followed by group two in clock cell two, group three in clock cell three, and group four in clock cell four. Using this sequence, non-adjacent signal lines are being switched during each clock cell, reducing the interaction and crosstalk.

Why Targets Don't Latch Address During Stepping Process

Since the entire address is not present on the bus until clock cell four, the initiator must delay assertion of the FRAME# signal until clock cell four when the final group driver is switched on. Because the assertion of FRAME# qualifies the address as being valid, no targets latch and use the address until FRAME# is sampled asserted.

Data Stepping

The data presented by the initiator during each data phase of a write transaction is qualified by the assertion of the IRDY# signal by the initiator. The data presented by the target during each data phase of a read transaction is qualified by the assertion of the TRDY# signal by the target. In other words, data can be stepped onto the bus, as well as address.

How Device Indicates Ability to Use Stepping

A device indicates its ability to perform stepping via the WAIT CYCLE CONTROL bit in its configuration command register. There are three possible cases:

- If the device is not capable of stepping, the bit is hardwired to zero.
- If the device always using stepping, the bit is hardwired to one.
- If the device's ability to use stepping can be enabled and disabled via software, the bit is implemented as a read/writable bit. If the bit is read/writable, reset sets it to one.

Designer May Step Address, Data, PAR (and PAR64) and IDSEL

The address may be stepped onto the AD bus (including the 64-bit extension) because it is qualified by FRAME#. PAR (and PAR64) may also be stepped because they are guaranteed qualified one clock after the end of the address phase and each data phase. IDSEL can be stepped because it is qualified by the FRAME# signal (refer to the section entitled "Resistively-Coupled IDSEL Is Slow" in the chapter entitled "Configuration Transactions"). Data can be stepped onto the AD bus during each data phase because it is qualified by the assertion of IRDY# (on a write) or TRDY# (on a read).

Table 8-3 defines the relationship of the AD bus, PAR, PAR64, IDSEL and DEVSEL# and the conditions that qualify them as valid.

Table 8-3. Qualification Requirements

Signal(s)	Qualifier
AD bus during address phase	Qualified when FRAME# signal sampled asserted at the end of the address phase.
AD bus during data phase on read	Qualified when TRDY# signal sampled asserted on a rising clock edge during the data phase.
AD bus during data phase on write	Qualified when IRDY# signal sampled asserted on a rising clock edge during the data phase.
PAR and PAR64	Implicitly qualified on rising clock edge after address phase, or by IRDY# and TRDY# (data phase).
IDSEL	Qualified when FRAME# sampled asserted at the end of the address phase and a type zero configuration command is present on the C/BE bus (with AD[1:0] =00b).

Continuous and Discrete Stepping

The initiator (or the target) may use one of two methods to step a valid address or data onto the AD bus, or a valid level onto the PAR and PAR64 signal lines, or IDSEL:

- If the device driving the AD bus and the parity pins or IDSEL, either initiator or target, **uses very weak output drivers**, it may take several clocks for it to drive a valid level onto these bus signals (i.e., the propagation delay may be lengthy because it may take several reflections, with the resultant voltage-doubling effect, before the address (or data) is in the correct state on the bus). This is known as **continuous stepping**. See note in the next section.
- The device driving the AD bus and the parity pins or IDSEL, either initiator or target, may have **strong output drivers** and may drive a subset of them on each of several clock edges until all of them have been driven. This is known as **discrete stepping**.

Disadvantages of Stepping

There are two disadvantages associated with stepping:

- Due to the prolonged period it takes to set up the address or data on the bus, there is a performance penalty associated in any address or data phase where stepping is used.
- In the midst of stepping the address onto the bus, the arbiter may remove the grant from the stepping master. This subject is covered in the next section.

The specification strongly discourages the use of continuous stepping because it results in poor performance and also because it creates violations of the input setup time at all inputs.

Preemption While Stepping in Progress

When the PCI bus arbiter grants the bus to a bus master, the master then waits for bus idle before initiating its transaction. If, during this period of time, the arbiter detects a request from a higher priority master, it can remove the grant from the first master before it begins a transaction (i.e., before it asserts FRAME#).

Assuming that this doesn't occur, the master retains its grant and awaits bus idle. Upon detection of the bus idle state, the master begins to step the address onto the AD bus, but delays the assertion of FRAME# for several clocks until the address is fully driven. During this period of time, the arbiter may still remove the grant from the master. The arbiter hasn't detected FRAME# as-

serted and may therefore assume that the master hasn't yet started a transaction (even though the arbiter can see that the bus is idle). If the arbiter receives a request from a higher-priority master, it may remove the grant from the master that is currently engaged in stepping an address onto the AD bus. In response to the loss of grant, the stepping master must immediately tri-state its output drivers.

It is a rule that the arbiter cannot deassert one master's grant and assert grant to another master during the same clock cell if the bus is idle. The bus may not, in fact, be idle. A master may not have asserted FRAME# yet because it is in the act of stepping the address onto the AD bus.

If the arbiter were to simultaneously remove the stepping master's GNT# and issue GNT# to another master, the following problem would result. On the next rising-edge of the clock, the stepping master detects removal of its GNT# and begins to turn off its address drivers. At the same time, the other master detects its GNT# and bus idle (because the stepping master had not yet asserted FRAME#) and initiates a transaction. This results in a collision on the AD bus.

When the bus appears to be idle, the arbiter must remove the grant from one master, wait one clock cell, and then assert grant to the other master. This provides a one clock cell buffer zone for the stepping master to disconnect completely before the other master detects its grant plus bus idle and starts its transaction.

It is permissible for the arbiter to simultaneously remove one master's grant and assert another's during the same clock cell if the bus isn't idle (i.e., a transaction is in progress). There is no danger of a collision because the master that has just received the grant cannot start driving the bus until the current master idles the bus.

Broken Master

The arbiter may assume that a master is broken if the arbiter has issued GNT# to the master, the bus has been idle for 16 clocks, and the master has not asserted FRAME# to start its transaction. The arbiter is permitted to ignore all further requests from the broken master and may optionally report the failure to the operating system (in a device-specific fashion).

Stepping Example

Figure 8-5 provides an example of an initiator using stepping over a period of three clocks to drive the address onto the AD bus. The initiator can start the transaction on clock three (GNT# sampled asserted and bus idle: FRAME# and IRDY# sampled deasserted). It then begins to drive the address onto the AD bus and the command onto the C/BE bus. During the clock cell four, it continues to drive the address onto the AD bus. During the clock cell five, it finalizes the driving of the address and asserts FRAME#, indicating the presence of the address and command. When the targets sample FRAME# asserted on the rising-edge of clock six (the end of the address phase), they latch the address and command and begin the address decode. Since this is an example of a write transaction, the initiator begins to drive the data onto the AD bus at the start of the data phase (clock six). Once again, it uses stepping, asserting the write data over a period of two clocks. It withholds the assertion of IRDY# until the data has been fully driven.

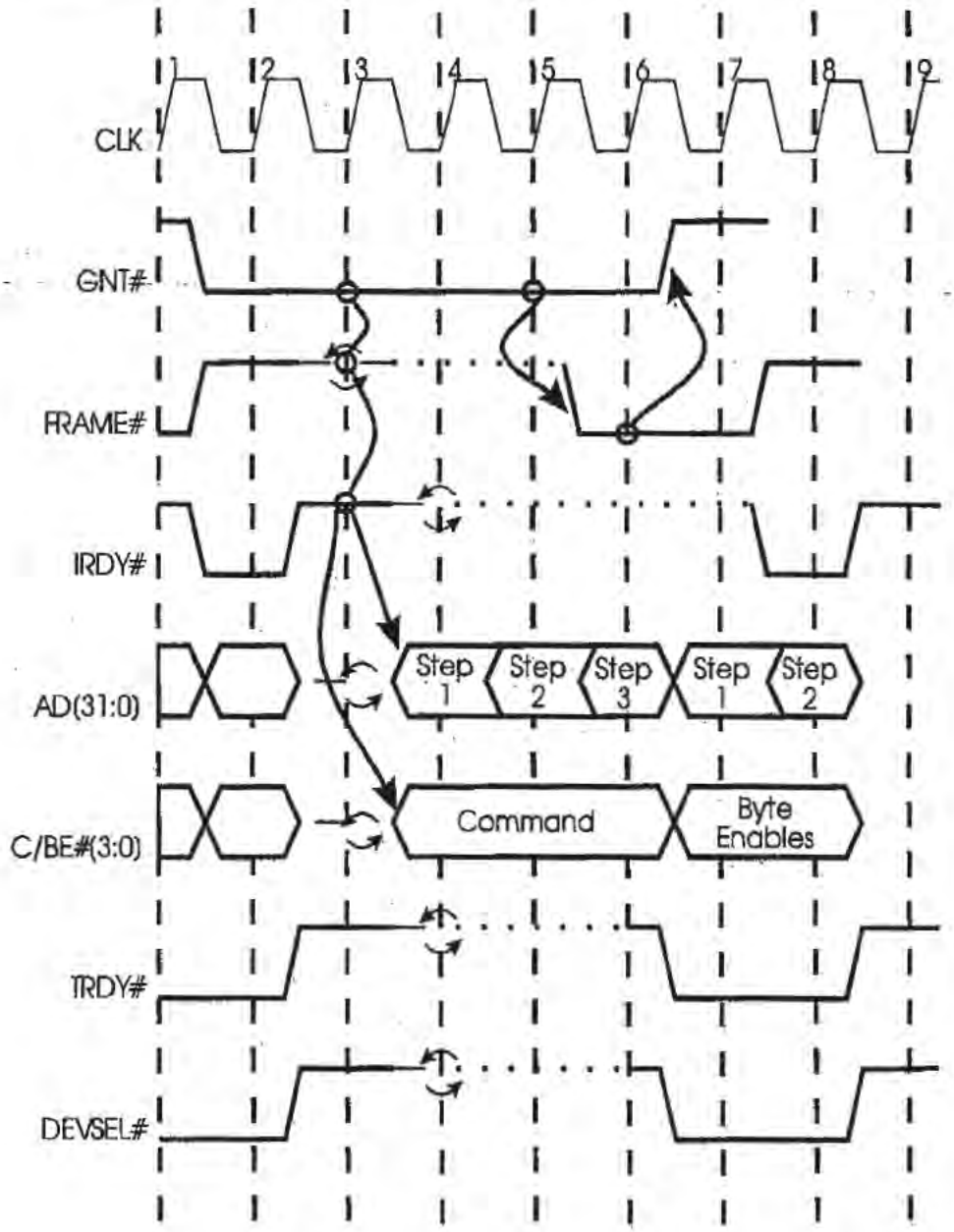


Figure 8-5. Example of Address Stepping

When Not to Use Stepping

Stepping must not be utilized when using 64-bit addressing because targets that respond to 64-bit addressing expect the upper 32 bits of the address to be presented one tick after FRAME# is sampled asserted.

Who Must Support Stepping?

All PCI devices must be able to handle address and data stepping performed by the other party in a transaction. The ability to use stepping, however, is optional.

Response to Illegal Behavior

Upon detection of illegal use of bus protocol, all PCI devices should be designed to gracefully return to the idle state (i.e., cease driving all bus signals) as quickly as possible. The specification is understandably vague on this point. It depends on the nature of the protocol violation as to whether the devices can gracefully return to their idle states and still function properly. As an example, the specification cites the case where the initiator simultaneously deasserts FRAME# and IRDY#. IN this case, when the target detects this illegal end to the transaction, it is suggested that the target deassert all target-related signals and return its state machine to the idle state. In the event that a protocol violation leaves a target device questioning its ability to function correctly in the future, it can respond to all future access attempts with a target abort. If the target thinks that the protocol violation has not impaired its ability to function correctly, it just surrenders all signals, returns to the idle state, and does not indicate any type of error.

Part VII

66MHz PCI Implementation

Chapter 22

Prior To This Chapter

The previous chapter provided a detailed description of issues related to caching from PCI memory targets. This subject was segregated in the latter part of the book because most PCI systems currently on the market do not support cacheable memory on the PCI bus. It injects considerable complexity into system and component design and the rewards may not be justified (due to the resultant degradation in performance).

In This Chapter

This chapter describes the implementation of a 66MHz bus and components.

The Next Chapter

The next chapter provides an overview of the VLSI Technology VL82C59x SuperCore PCI chipset.

Introduction

The revision 2.1 PCI specification defines support for the implementation of buses and components that operate at speeds of up to 66MHz. This chapter covers the issues related to this topic.

66MHz Uses 3.3V Signaling Environment

66MHz components only operate correctly in a 3.3V signaling environment. The 5V environment is not supported. This means that 66MHz add-in cards are keyed to install in 3.3V or universal card connectors and cannot be installed in 5V card connectors.

How Components Indicate 66MHz Support

The 66MHz PCI component or add-in card indicates its support in two fashions: electrically and programmatically.

A 66MHz PCI bus includes a newly-defined signal, M66EN. This signal must be bussed to the M66EN pin on all 66MHz-capable devices embedded on the system board and to a redefined pin (referred to as M66EN) on any 3.3V connectors that reside on the bus. The system board designer must supply a single pullup on this trace. The redefined pin on the 3.3V connector is B49 and is used as a ground pin by 33MHz PCI devices. Unless grounded by a PCI device, the natural state of the M66EN signal is asserted (due to the pullup). 66MHz embedded devices and cards either use M66EN as an input or don't use it at all (this is discussed later in this chapter).

The designer must include a 0.01μF capacitor located within .25" of the M66EN pin on each add-in connector in order to provide an AC return path and to decouple the M66EN signal to ground.

The PCI devices embedded on a 66MHz PCI bus are all 66MHz devices. A card installed in a connector on the bus may be either a 66MHz or a 33MHz card. If the card connector(s) aren't populated, M66EN stays asserted (by virtue of the pullup). If any 33MHz component is installed in a connector, the ground plane on the 33MHz card is connected to the M66EN signal, deasserting it.

How Clock Circuit Sets Its Frequency

The M66EN signal is provided as an input to the PCI clock circuit on the system board. If M66EN is sampled asserted by the clock circuit, it provides a 66MHz PCI clock to all PCI devices. If M66EN is sampled deasserted, the clock circuit supplies a 33MHz PCI clock. It should be fairly obvious that if any 33MHz components are installed on a 66MHz bus, the bus then operates at 33MHz.

Does Clock Have to be 66MHz?

As defined in revision 1.0 and 2.0 of the specification, the PCI bus does not have to be implemented at its top rated speed of 33MHz. Lower speeds are

acceptable. The same is true of the 66MHz PCI bus description found in revision 2.1 of the specification. All 66MHz-rated components are required to support operation from 0 through 66MHz. The system designer may choose, however, to implement a 50MHz PCI bus, a 60MHz PCI bus, etc.

Clock Signal Source and Routing

The specification recommends that the PCI clock be individually-sourced to each PCI component as a point-to-point signal from separate, low-skew clock drivers. This diminishes signal reflection effects and improves signal integrity. In addition, the system board and add-in card designer must adhere to the clock signal maximum trace length defined in revision 2.0 of the specification (2.5").

Stopping Clock and Changing Clock Frequency

As with the 33MHz PCI bus specification, the 66MHz specification states that the clock frequency may be changed at any time as long as the clock edges remain clean and the minimum high and low times are not violated. However, the clock frequency may not be changed except in conjunction with assertion of the PCI RST# signal. As an exception, components designed to be integrated onto the system board may be designed to operate at a fixed frequency (up to 66MHz) and may require that no clock frequency changes occur.

The clock may be stopped, but only in the low state (to conserve power).

How 66MHz Components Determine Bus Speed

When a 66MHz-capable device senses M66EN deasserted (at reset time), this automatically disables the device's ability to perform operations at speeds above 33MHz. If M66EN is sensed asserted, this indicates that no 33MHz devices are installed on the bus and the clock circuit is supplying a high-speed PCI clock.

A 66MHz device uses the M66EN signal in one of two fashions:

- The device is not connected to M66EN at all (because the device has no need to determine the bus speed in order to operate correctly).

- As described above, the device implements M66EN as an input (because the device requires knowledge of the bus speed in order to operate correctly).

System Board with Separate Buses

The system board designer can partition the board into two or more PCI buses. A 66MHz bus can be populated with devices that demand low-latency and high throughput. A separate 33MHz PCI bus is populated only with 33MHz devices.

Maximum Achievable Throughput

The theoretical maximum achievable throughput on a 66MHz PCI bus would be:

- 4 bytes per data phase * 66 million data phases per second = 264MB/second. This would be a 32-bit bus master bursting with a 32-bit target.
- 8 bytes per data phase * 66 million data phases per second = 528MB/second. This would be a 64-bit bus master bursting with a 64-bit target.

Electrical Characteristics

To ensure compatibility when operating in a 33MHz PCI bus environment, 66MHz PCI drivers must meet the same DC characteristics and AC drive points as 33MHz bus drivers. However, 66MHz PCI bus operation requires faster timing parameters and redefined measurement conditions. Because of this, a 66MHz PCI bus may require less loading and shorter trace lengths than the 33MHz PCI bus environment.

Figure 22-1 illustrates the differences in timing between 33 and 66MHz component operation. The chapter entitled "Intro To Reflected-Wave Switching" provides detailed information regarding 33MHz bus timing and the various timing components (e.g., Tval, Tprop, etc.).

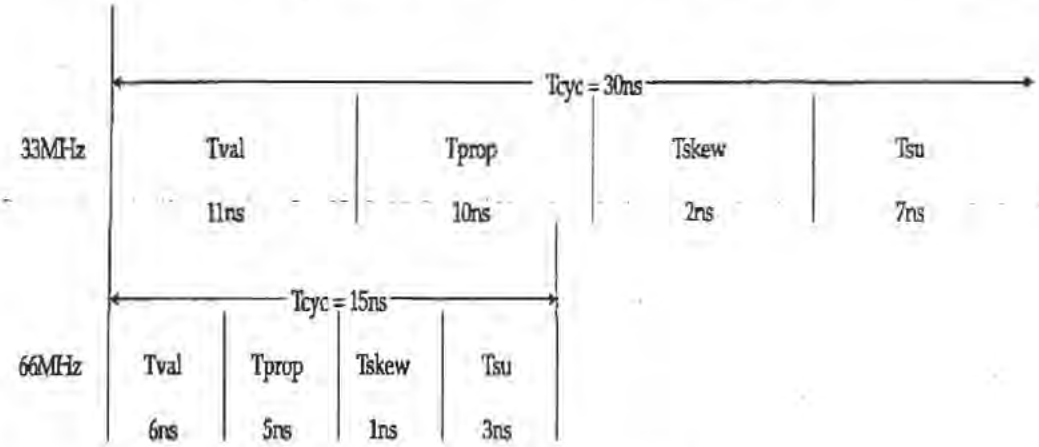


Figure 22-1. 33 versus 66MHz Timing

33MHz drivers are specified by their V/I curves, while 66MHz drivers are specified in terms of their AC and DC drive points, timing parameters, and slew rate. The specification defines the following parameters:

- The minimum AC drive point defines an acceptable first step voltage and must be reached within the maximum Tval time.
- The maximum AC drive point limits the amount of overshoot and undershoot in the system.
- The DC drive point specifies steady-state conditions.
- The minimum slew rate and the timing parameters guarantee 66MHz operation.
- The maximum slew rate minimizes system noise.

66MHz PCI designers must design drivers that launch sufficient energy into a 25Ω transmission line so that correct input levels are guaranteed after the first reflection.

At 66MHz, the clock cycle time is 15ns (vs. 30ns at 33MHz), while the minimum clock high and low times are 6ns each (vs. 11ns at 33MHz). The clock slew rate has a minimum specification of 1.5 and a maximum of 4 volts/ns (same as 33MHz specification). Table 22-1 defines the 66MHz timing parameters and provides a side-by-side comparison with the 33MHz timing parameters. The following exception applies to the 66MHz values in the table: REQ# and GNT# are point-to-point signals and have different setup times than do bussed signals. They have a setup time of 5ns.

Table 22-1. 66MHz Timing Parameters

		66MHz		33MHz	
Symbol	Description	Min	Max	Min	Max
Tval	CLK to signal valid delay, bussed signals	2ns	6ns	2ns	11ns
Tval (ptp)	CLK to signal valid delay, point-to-point signals	2ns	6ns	2ns	12ns
Ton	Float to active delay	2ns		2ns	
Toff	Active to float delay		14ns		28ns
Tsu	Input setup time to CLK, bussed signals	3ns		7ns	
Tsu (ptp)	Input setup time to CLK, point-to-point signals				
Th	Input hold time from CLK	0ns		0ns	
Trst	Reset active time after power stable	1ms		1ms	
Trst-clk	Reset active time after CLK stable	100µs		100µs	
Trst-off	Reset active to output float delay		40ns		40ns
Trrsu	REQ64# to RST# setup time	10Tcyc		10Tcyc	
Trrh	RST# to REQ64# hold time	0ns	50ns	0ns	50ns

When computing the 66MHz bus loading model, a maximum pin capacitance of 10pF must be assumed for add-in boards, whereas the actual pin capacitance may be used for devices embedded on the system board.

Addition to Configuration Status Register

A 66MHz-capable device adds one additional bit to its configuration status register. Bit 5 is defined as the 66MHZ-CAPABLE bit. A 66MHz-capable device hardwires this bit to one. For all 33MHz devices, this bit is reserved and is hardwired to zero. Software can determine the speed capability of a PCI bus by checking the state of this bit in the status register of the bridge to the bus in question (host/PCI or PCI-to-PCI bridge). Software can also check this bit in the status register of each additional device discovered on the bus in question to determine if all of the devices on the bus are 66MHz-capable. If even just one device returns a zero from this bit, the bus runs at 33MHz, not 66MHz. Table 22-2 defines the combinations of bus and device capability that may be detected.

Table 22-2. Combinations of 66MHz-Capable Bit Settings

Bridge's 66MHz-Capable Bit	Device's 66MHz-Capable Bit	Description
0	0	33MHz device located on 33MHz bus. Bus and all devices operate at 33MHz.
0	1	66MHz-capable device located on 33MHz bus. Bus and all devices operate at 33MHz. If the device is an add-in device and is only capable of proper operation when installed on a 66MHz bus, the configuration software may decide to prompt the user to install the card in an add-in connector on a different bus.
1	0	33MHz device located on 66MHz-capable bus. Bus and all devices operate at 33MHz.
1	1	66MHz-capable device located on 66MHz-capable bus. If status check of all other devices on the bus indicates that all of the devices are 66MHz capable, the bus and all devices operate at 66MHz.

Latency Rule

Devices residing on the 66MHz PCI bus are typically low-latency devices. The revision 2.1 specification requires that, on a read transaction, the time from assertion of FRAME# to the completion of the first data phase not exceed 16 PCI clocks. If it will, the target device must issue retry to the master. For multimedia applications, the majority of accesses are writes, not reads. Typically, a target device can accept write data faster than it may be able to supply read data. On a read, the device may need to access a slow medium. The device cannot be permitted to tie up the bus while fetching the requested data.

66MHz Component Recommended Pinout

The revision 2.0 specification suggested a recommended PCI component pinout wherein the signals wrapped around the component in the same order as the pin sequence on the add-in connector. The revision 2.1 specification states that "the designer may modify the suggested pinout...as required" to meet the 66MHz electrical specification.

Adding More Loads and/or Lengthening Bus

Running the PCI bus at 66MHz imposes tighter constraints on trace length and the number of loads the bus supports. The system board designer may choose to run the bus at a lower speed (e.g., 50MHz), thereby permitting longer traces and/or additional loads.

Number of Add-In Connectors

As a general rule, there is only one add-in connector on a 66MHz bus, but the specification does not preclude the inclusion of additional connectors (as long as the electrical integrity of the bus is maintained).

Part VIII

Overview of VLSI Technology VL82C59x SuperCore PCI Chipset

Chapter 23

Prior To This Chapter

The previous chapter described the implementation of a 66MHz bus and components.

In This Chapter

The PCI specification supports many permutations of system and therefore chipset design. This chapter provides an overview of the VL82C59x Super-Core PCI chip set from VLSI Technology. This overview is provided to present an example of PCI chipset implementation. It is not intended to provide a detailed description of the chipset operation. The VLSI component specification should be consulted for that purpose. In addition, it is assumed that the reader already has an understanding of the ISA bus. For detailed information on the ISA bus operation and environment, refer to the Addison-Wesley publication entitled *ISA System Architecture*, also authored by MindShare. The author would like to thank VLSI Technology for providing access to the chipset specification.

Chipset Features

The VLSI VL82C59x chipset provides the core logic necessary to design a Pentium-based system that incorporates both the PCI and ISA buses. It supports all 5V and 3.3V Pentium processors with host bus speeds of up to 66MHz. This includes the P5, P54C, P54CM and P54CT. It also supports dual-P54C processors. The chipset design includes the following features:

- Bridges the host and PCI buses.
- Bridges the PCI and ISA buses.
- Integrated L2 lookaside, direct mapped, write-through cache.
- Integrated system DRAM controller.
- Integrated PCI bus arbiter.

- Provision of posted-memory write buffers in both bridges.
- Supports Pentium processor's pipelined bus cycles.
- Self-configuring system DRAM banks.
- Shadow RAM support
- SMM support.
- Decoupled DRAM refresh.
- Supports synchronized or asynchronous processor and PCI clocks.
- Supports optional posting of I/O writes.
- Optional support for memory prefetching.
- PCI master reads from system DRAM memory can be serviced from processor's L2 cache or system memory.
- PCI master writes to system DRAM memory are absorbed by the bridge's posted-write buffer.
- Supports multiple-data phase PCI burst transactions.

Intro to Chipset Members

Refer to figure 23-1. The VLSI VL82C59x SuperCore PCI chipset consists of the following entities:

- VL82C591 Pentium System Controller. In conjunction with two VL82C592 Data Buffers, the system controller comprises the bridge between the host processor's local bus and the PCI bus.
- VL82C592 Pentium Processor Data Buffer. Taken together, two data buffers provide a triple-ported data bus bridge between the host data bus, system DRAM data bus and the PCI data bus (AD bus).
- VL82C593 PCI/ISA Bridge. The '593 provides the bridge between the ISA and PCI buses. In addition, the '593 incorporates much of the ISA system support logic.

The sections that follow provide additional information about the capabilities of the chipset.

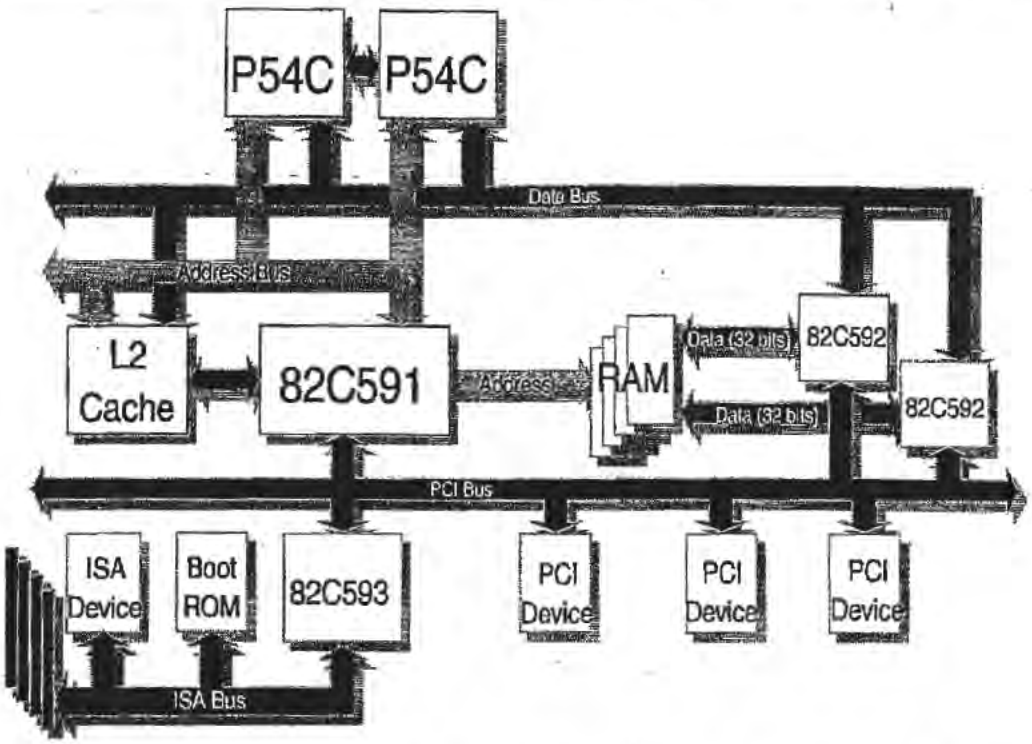


Figure 23-1. System Design Using VLSI VL82C59x SuperCore Chipset

VL82C592 Pentium Processor Data Buffer

As illustrated in figure 23-1, the host/PCI bridging function consists of the VL82C591 Pentium system controller and two VL82C592 data buffers. The two data buffer chips are controlled by the '591. They provide the following basic capabilities:

- On host processor reads from system memory, the '591 reads the requested data from DRAM and instructs the '592 data buffers to pass it to the host data bus.
- On host processor writes to system memory, the '591 instructs the data buffers to accept the write data into the posted-write buffer. This permits the host processor to conclude the memory write quickly. The posted-write buffer then offloads the write data to DRAM memory.

- On PCI-initiated memory reads from system DRAM memory, the '591 reads the requested data from memory and instructs the '592 data buffers to pass it to the requester on the PCI data bus.
- On PCI-initiated memory writes to system DRAM memory, the '591 addresses memory and instructs the '592 data buffers to accept the data presented on the PCI data bus and route it into system DRAM.

A discussion of the data buffer posted write capability can be found later in this chapter.

The following section discusses the functionality of the host/PCI bridge.

'591/'592 Host/PCI Bridge

General

As stated earlier, the host/PCI bridge functionality is provided by the '591 in combination with two '592 data buffers. The bridge performs the following basic functions:

- Services system DRAM memory reads and writes initiated by the host processor.
- Permits host processor(s) L1 cache(s) to snoop system memory accesses initiated by PCI and ISA masters.
- Translates host processor-initiated memory and I/O accesses into PCI memory and I/O accesses.
- Services system memory accesses initiated by PCI and ISA masters.
- Translates specific host processor-initiated I/O operations into PCI configuration read or write operations.
- Translates specific host processor-initiated I/O operations into PCI special cycle transactions.
- Incorporates the PCI and host bus arbiters.
- Translates host processor-initiated interrupt acknowledge bus cycles into PCI interrupt acknowledge transaction.

System DRAM Controller

The controller for system DRAM memory resides within the '591. Each memory bank (up to four) is either 64-bits (without parity) or 72-bits wide (with

parity). Each bank may be up to 256MB in size, yielding a maximum possible memory population of 1GB. In addition, each bank may be populated with 32 or 36-bits memory modules, permitting less-costly memory upgrade. The DRAM configuration registers permit the DRAM controller to work with DRAMs of various speeds and different geometries.

The controller supports two-way interleaved, page-mode memory. One or two pages (one in each bank) can be kept open at a time. For page-mode DRAMs that have a page open timeout of less than 15μs, the controller automatically closes a page that has been open for a period of 10μs. When using DRAMs with a maximum page open timeout in excess of 15μs, the 10μs automatic page close feature may be disabled and the refresh cycles can take care of ensuring that a page does not remain open for an excessive period. Non-page mode DRAM is not supported.

Refresh cycles may be set to occur every 15.625μs, 62.5μs, 125μs or 250μs. DRAM refresh cycles are transparent to the processor. If the processor initiates a DRAM access request simultaneously with a refresh cycle, the processor is stalled (i.e., wait states are inserted in its bus cycle) until the refresh cycle completes.

When a system DRAM parity error is detected, it is reported by the assertion of the PCI SERR# signal (assuming that the SERR# enabled and parity error response bits are set in the bridge's configuration command register). SERR# is typically connected to the '593 which asserts NMI to the host processor when SERR# is asserted. An option permits bad parity to be deliberately written to system DRAM to facilitate test and diagnostics.

Host processor-initiated memory accesses that target locations above the top of installed system DRAM are passed to the PCI bus and are not cached in the L1 and L2 caches. In addition, memory address ranges defined by the bridge's segment attribute and programmed memory region registers are also passed to the PCI bus and are not cached from.

The chipset does not permit the L1 and L2 caches to cache information from memory beyond the host/PCI bridge (i.e., PCI and ISA memory). This being the case, the '591 does not implement the snoop result outputs (SDONE and SBO#).

L2 Cache

The L2 cache controller is embedded within the '591 system controller. It is a direct-mapped, lookaside, buffered write-through cache. The L2 cache only caches information from system DRAM memory, never from PCI or ISA memory. The DRAM controller may be programmed to recognize sub-ranges within the overall memory address range assigned to system DRAM as PCI memory. When the processor initiates a memory transaction targeting an address in any of these programmed sub-ranges, the transaction is passed to the PCI bus and the data is not cached in L1 or L2.

The recommended L2 cache sizes are 256KB, 512KB and 1MB, but the L2 cache may be implemented as any desired size. The limitation is the amount of tag SRAM supplied by the system designer. The tag SRAM (i.e., the cache directory) is external to the '591 and can be of any size. Optionally, the L2 cache may be parity-protected.

The cache controller supports L2 cache line sizes of both 32 and 64 bytes. Additional SRAM is necessary to support the larger line size. When the 64 byte line size is implemented, a processor-initiated read miss in L2 results in the requested 32 byte line being read from DRAM. The line is sent back to the processor and a copy is also stored in the L2 cache. To the L2 cache, this is considered to be half of a line. The cache reads the next 32 bytes from DRAM and establishes it in the L2 as the second half of the 64 byte line.

A write-through cache usually extends the duration of a host processor-initiated memory write until the data has been written through to system memory. In this chipset design, the '591 instructs the '592 data buffers to accept the write data and permits the processor to complete its memory write immediately.

The cache controller supports both asynchronous and synchronous SRAMs. When using asynchronous SRAMs, burst read timing of 3-2-2-2 (three processor bus clocks to transfer the first quadword, and two clocks each for the transfer of each of the other three quadwords in the line) is achievable (at a bus speed of 66MHz). Burst write timing of 4-2-2-2 or 3-2-2-2 is achievable (depending on tag SRAM speed and signal loading). When using synchronous SRAMs, burst reads and write timing of 3-1-1-1 or 2-1-1-1 is achievable (depending on tag SRAM speed and SRAM type). When the processor performs back-to-back burst reads, pipelining reduces access time to 1-1-1-1.

Startup software can determine the following information related to the L2 cache:

- Cache SRAM type (asynchronous, synchronous type one or synchronous type two).
- Cache size.
- Line size.
- Cacheable memory range.
- Wait states imposed by cache SRAM type/speed.

Posted-Write Buffer

General

The posted-write buffer absorbs processor-initiated writes and permits the processor to end the write transaction quickly. The buffer logic then initiates the write to memory (or to PCI). While the buffer is engaged in the write, the processor can start and complete another memory write (assuming the buffer isn't full, it is absorbed by the posted-write buffer as well), a read hit on the L2 cache, or a write to the PCI bus (if the previously posted write was to system memory). The posted-write buffer that absorbs processor writes destined for system memory is eight quadwords deep (a quadword is 64-bits).

The posted-write buffer that absorbs memory writes destined for the PCI (or ISA) bus is one quadword deep. The bridge can only post writes to PCI/ISA memory within regions of memory programmed with the prefetchable attribute (in a '591 device-specific register). Optionally, the '591 can also be programmed to post PCI I/O writes initiated by the host processor. Any time the processor initiates a write to PCI/ISA memory in an area programmed to permit posting, the write is absorbed into the 64-bit PCI posted-write buffer. If the processor should initiate a subsequent memory write within the same quadword, the second write is merged into the bytes already in the buffer. This can result in non-contiguous byte enables asserted during the resulting PCI memory transaction, but this feature can be disabled. When disabled, the '591 uses a byte-reduction algorithm to generate two separate PCI memory writes utilizing only contiguous byte enables. Whenever the '591 has a PCI/ISA memory write posted in the buffer, it arbitrates for PCI bus ownership. When the bus has been acquired, it performs the memory write on the PCI bus. If the processor should initiate another PCI memory write prior to the conclusion of the one already in progress on the PCI bus, the processor is

stalled until the write buffer becomes available at the completion of the current PCI memory write transaction. In addition, bus ownership requests from other PCI bus masters are ignored until the conclusion of the current transaction.

The '591 does not permit a processor-initiated PCI read transaction to be performed on the PCI bus if a processor write to PCI memory is currently-posted in the buffer. The buffer is first flushed to PCI memory before the read is performed on the PCI bus.

The processor initiates burst write operations during the castout of a modified line or a snoop push-back (write-back) operation). The posted-write buffer (located in the data buffers) can accept the burst data at full bus speed (0 wait states).

The write buffer permits posting of memory writes to PCI memory within regions of memory space defined as prefetchable by bridge configuration registers.

A status bit can be checked by software to determine if the write buffer is empty.

Combining Writes Feature

The write buffer supports combining of writes. Assume that the processor performs a memory write to write two bytes into memory locations 00000100h and 00000101h. The processor outputs the following information:

- The quadword-aligned address placed on the host processor address bus is 00000100h.
- Byte enables [1:0] are asserted to indicate that the first two locations in the currently-addressed quadword are being addressed. Byte enables [7:2] are deasserted, indicating that the third through the eight locations in the quadword are not being addressed.
- The two bytes of data destined for memory locations 00000100h and 00000101h are driven onto data paths zero (D[7:0]) and one (D[15:8]).

The posted-write buffer latches the quadword address and the two bytes into the next available quadword location in its FIFO buffer. BRDY# is assert to the processor, permitting it to end the memory write transaction. Now assume that the processor initiates another memory write, this time to memory loca-

tion 00000104h (before the buffer logic has written the previous two bytes into system DRAM memory). Assume that the processor outputs the following information:

- The quadword-aligned address placed on the host processor address bus is 00000100h.
- Byte enable [4] is asserted to indicate that the fifth location in the currently-addressed quadword is being addressed. Byte enables [7:5] and [3:0] are deasserted, indicating that the first through fourth and the sixth through the eight locations in the quadword are not being addressed.
- The byte of data destined for memory location 00000104h is driven onto data path four (D[39:32]).

The buffer recognizes that some portion of quadword 00000100h has already been posted to be written to memory. Instead of using up another quadword-wide buffer location for the new write, it combines the new data being supplied by the processor with the older data in the buffer location. The buffer location now contains three bytes to be written to quadword 00000100h in system DRAM. Although the processor performed two separate memory writes to system memory, the buffer logic only has to perform one write operation when it offloads the data to memory.

Read-Around and Merge Features

If the processor initiates a read from system DRAM while one or more memory write operations reside within the posted-write buffer, the buffer logic performs the read from DRAM before flushing the writes to memory. If the read hits on a posted-write in the buffer, the bytes posted to be written to memory are merged with the data read from memory and the resulting data is supplied back to the processor.

Write Buffer Prioritization

The '591 can be programmed to adjust the priority of posted-write buffer writes to memory relative to memory reads. The following settings are available:

- The write buffer can access memory whenever the DRAM is idle.
- The write buffer can access memory after a minimum of 2, 4, 8, 16, 32 or 64 CPU clocks from the completion of the last DRAM read. The count is restarted at the completion of each read. When any of these settings are selected, the write buffer is permitted to access memory when the proces-

sor generates an access that is not a read (e.g., another write or a PCI transaction).

- Write buffer access to system memory is permitted only when the processor generates a non-system memory read transaction.

Configuration Mechanism

The '59x chipset implements PCI configuration mechanism number one (configuration address port at I/O location 0CF8h and configuration data port at I/O location 0CFCh).

PCI Arbitration

The '591 incorporates the PCI bus arbiter. The arbiter supports the '591, the '593 and up to four additional PCI bus masters. The '591's REQ# and GNT# signals are internally connected to the arbiter. A single signal line is used by the '593 to request and be granted ownership of the PCI bus (refer to the section in this chapter entitled "'593 Characteristics When PCI Master." Optionally, the '593 may use one of the four REQ#/GNT# signal pairs for arbitration.

The '591 never generates fast back-to-back transactions because it doesn't know the address boundaries of different targets.

The priority scheme may be software selected as fixed or rotational. When fixed is selected, the '593's REQGNT# signal has highest priority. This guarantees DMA channels timely access to the bus. Then, in descending order of importance, the priorities of the other masters are master 3, master 2, master 1, master 0 and the processor. The processor has lowest priority. Whenever any of the PCI masters require access to the PCI bus, the '591 asserts HOLD to the processor and takes the bus away from it to grant to the most important PCI master.

When rotational priority is selected, the '593 has highest priority, with priority rotating between bus masters 0 through 3 and the host processor. Refer to figure 23-2.

The arbiter can be programmed to park the bus either on the '591 or on the last master that used the bus. The latter mode can only be selected when rotational priority has been selected.

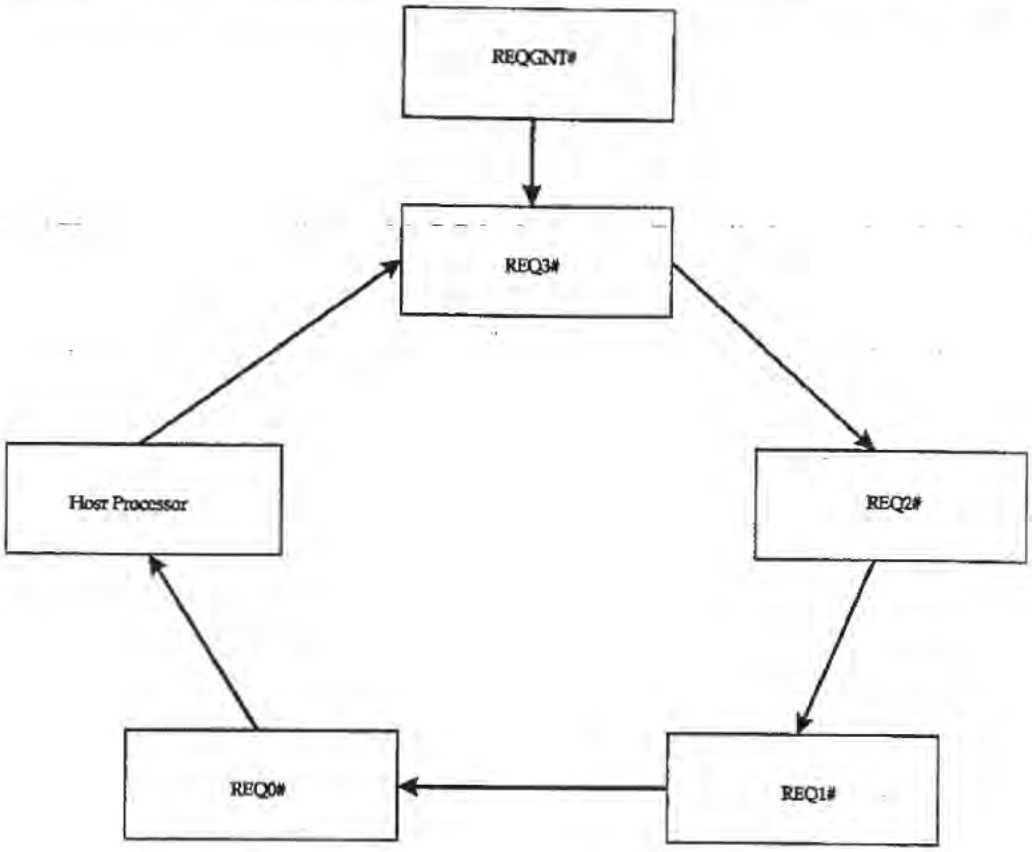


Figure 23-2. Rotational Priority Scheme

Locking

The arbiter in the '591 implements a bus lock. It does not support target locking.

Special Cycle Generation

Software can stimulate the host/PCI bridge to generate a PCI special cycle to PCI bus zero or to any of its subordinate PCI buses using the method defined for configuration mechanism number one; that is, the programmer performs a 32-bit write to the configuration address port specifying the target PCI bus, and sets the target device number, function number and doubleword number

to 1Fh, 7h and 00h, respectively. The programmer then performs a two byte or four byte write to the configuration data port. The bridge performs a special cycle on the target PCI bus, supplying the data written to the configuration data port as the message during the data phase. If the target bus is a subordinate bus, the '591 generates a special cycle request using a type 1 configuration write transaction.

'591 Configuration Registers

Figure 23-3 illustrates the '591's PCI configuration registers. The sections that follow define the manner in which the chipset implements each of these registers. For a description of the '591's device-specific configuration registers, refer to the chipset specification.

Vendor ID Register

The vendor ID for VLSI Technology is 1004h.

Device ID Register

The device ID for the '591 is 0005h.

Command Register

Table 23-1 defines the '591's usage of its command register bits.

Table 23-1. '591 Command Register Bit Assignment

Bit	Description
0	**I/O enable bit.** Hardwired to zero because the '591 doesn't respond to any PCI I/O transactions.
1	**Memory enable bit.** When set to one, PCI bus masters can access system DRAM memory. Reset sets this bit to one.
2	**Master enable bit.** Hardwired to one because the '591 is always enabled to initiate PCI transactions.
3	**Special cycle monitor enable bit.** Hardwired to zero because the '591 does not monitor special cycles generated by other PCI masters.
4	**Memory write and invalidate enable bit.** Hardwired to zero because the '591 never generates the memory write and invalidate command.
5	**VGA color palette snoop enable bit.** Hardwired to zero. Only VGA-compatible devices and PCI-to-PCI bridges are required to implement this bit.
6	**Parity error response bit.** When set to one, the '591 asserts PERR# when a data parity error is detected. Also used to qualify the assertion of SERR# on address phase parity error. Reset clears this bit.
7	**Stepping enable bit.** Hardwired to zero because the '591 never uses address or data stepping.
8	**System error response bit.** When set to one, the '591 is enabled to assert SERR# (if the PARITY ERROR RESPONSE bit is also set to one) when address phase parity error detected or system DRAM parity error. Reset clears this bit.
9	**Fast back-to-back enable bit.** Hardwired to zero because the '591 never performs fast back-to-back transactions.
15:10	**Reserved** and hardwired to zero.

Status Register

Table 23-2 defines the '591's usage of its status register bits.

Table 23-2. '591's Status Register Bit Assignment

Bit	Description
6:0	**Reserved** and hardwired to zero.
7	**Fast back-to-back capable bit.** Hardwired to one, indicating that, when acting as a target, the '591 supports fast back-to-back transactions to different targets.
8	**Signaled parity error bit.** Set to one when the '591, acting as a master, samples PERR# asserted by the target during a write or the '591 asserts PERR# on a read. Reset clears this bit to zero.
10:9	**DEVSEL timing.** Hardwired to 01b, indicating that the '591 has a medium speed PCI address decoder.
11	**Signaled target abort.** Hardwired to zero because the '591 never signals a target abort.
12	**Received target abort.** Set to one when the '591 receives a target abort from a target when acting as master. Reset clears this bit to zero.
13	**Received master abort.** Set to one when the '591 experiences a master abort when acting as master. Reset clears this bit to zero.
14	**Signaled system error.** Set to one by the '591 when it assert SERR#. Reset clears this bit to zero. The SERR# ENABLE and PARITY ERROR RESPONSE bits in the '591's command register must be set to enable the '591 to generate SERR# and set this bit.
15	**Received parity error.** Set to one when the '591 detects an address or data phase parity error. Reset clears this bit to zero.

Revision ID Register

The revision ID register contains 00h in the first release of the '591.

Class Code Register

The class code register contains 060000h. 06h specifies the bridge class. The middle byte, 00h, specifies that the sub-class is host/PCI bridge. The lower byte is always 00h for all revision 2.x-compliant devices.

Cache Line Size Configuration Register

Not implemented. Since the '591 contains the cache controller, it "knows" the system cache line size.

Latency Timer Register

Hardwired with a value of 10h (16d). When acting as a PCI bus master, the '591 never performs bursts of longer than two data phases. The specification states that any device that never performs more than two data phases may hardwire a value into its LT, but the value may not exceed 16d.

Header Type Register

Hardwired with the value 00h. This indicates that the '591 is a single-function device (bit 7 = 0) and that the format of configuration doublewords 4 through 15 adheres to the header type zero definition.

BIST Register

Hardwired to 00h. Bit 7 = 0 indicates that the '591 does not implement a built-in self test.

Base Address Registers

None implemented. The '591 utilizes device-specific registers to set up its system DRAM address decoders. Regarding I/O, the '591 only implements two I/O ports: the configuration address port at I/O address 0CF8h and the configuration data port at I/O address 0CFCh. It has hardwired address decoders for these registers.

Expansion ROM Base Address Register

Not implemented because the '591 does not incorporate a PCI device ROM.

Interrupt Line Register

Not implemented because the '591 does not generate interrupt requests.

Interrupt Pin Register

Hardwired with 00h, indicating that the '591 does not implement a PCI interrupt request output pin.

Min_Gnt Register

The '591 incorporates the PCI bus arbiter and already knows its timeslice and intrinsically knows its own bus acquisition latency requirements.

Max_Lat Register

See Min_Gnt register section (previous section).

Bus Number Register

Hardwired to 00h, indicating that the PCI bus residing directly behind the '591 is PCI bus zero.

Subordinate Bus Number Register

Hardwired to FFh. Any software requests to perform special cycles or configuration reads or writes on buses other than bus zero are therefore passed through the '591 as type one configuration accesses. If the target bus doesn't exist, the type one configuration access will terminate in a master abort.

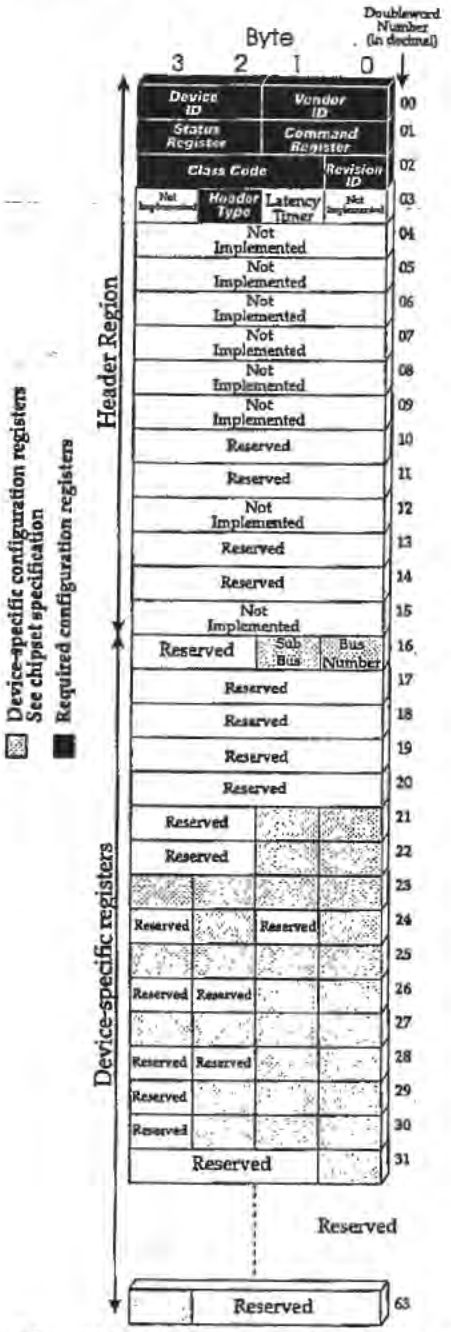


Figure 23-3. '591 PCI Configuration Registers

PCI Device Selection (IDSEL)

When translating accesses to the configuration data port into PCI type zero configuration accesses, the '591 internally decodes the target device number specified in the configuration address port and selects an IDSEL to assert. Rather than implementing an IDSEL output for each physical device position on the PCI bus, the '591 asserts one of the upper AD lines during the address phase of the PCI configuration access. Table 23-3 defines the AD line asserted (set to one) for each device number that may be specified. On the system board, each physical device position resistively-couples one of the upper AD lines to the device's IDSEL input pin.

Table 23-3. Device Number To AD Line Mapping

Device Number Specified (decimal)	AD Line Asserted
0	11 *
1	12
2	13
3	14
4	15
5	16
6	17
7	18
8	19
9	20
10	21
11	22
12	23
13	24
14	25
15	26
16	27
17	28
18	29
19	30
20	31 *
21 - 31	none

* Note: the '591 may be programmed to be either device 0 or device 31.

Handling of Host Processor-Initiated Transactions

Memory Read

When the host processor initiates a memory (code or data) read transaction, one of the following is true:

- The target location is within the range of system DRAM memory.
- The target location falls within the range assigned to system DRAM memory, but is within a sub-range defined as PCI or ISA memory.
- The target location is above the top of installed system memory.

In the first case, the address is considered to be cacheable. The lookaside L2 cache performs a lookup to determine if the requested data is present in the cache. If present, the cache tells the DRAM controller to abort the access to system DRAM and the cache supplies the requested data to the processor. If the requested data isn't currently present in the L2 cache, the DRAM controller proceeds with the memory read to fetch the target line. If the read hits on any posted-writes currently outstanding in the posted-write buffer, the write data is merged with the data read from memory and the requested data is supplied to the processor. The L2 cache also latches a copy of the line of information. If the L2 cache line size is 64 bytes, the cache initiates a second line read from memory to load the second half of its line into the L2 cache.

In the second and third cases, the '591 arbitrates for ownership of the PCI bus and initiates a memory read transaction (note that if the processor already has a PCI memory write posted in the '591, the posted write will be flushed to PCI memory before the PCI memory read is initiated). Because the L2 and L1 caches are not permitted to cache from memory beyond the bridge, the resulting PCI memory read transaction does not fetch an entire line (32 bytes) from memory. Rather, the access consists of one or two data phases (at most, the processor is expecting to get back eight bytes of information). When the PCI memory read transaction is initiated, DEVSEL# is asserted if a memory target on the PCI bus recognizes that it is the target of the transaction. That target then supplies the requested data to the '591 and the '591 supplies it to the processor. If no PCI memory target asserts DEVSEL#, the subtractive decoder built into the '593 eventually asserts DEVSEL# and claims the transaction. The transaction is passed to the ISA bus. If an ISA memory target recognizes the address, that target supplies the data. If the address is not recognized by any ISA memory target, the ISA bus controller in the '593 latches all ones from the

ISA data bus (its quiescent state when not being driven) and that is sent back to the '591 and then to the processor.

Note that the '591 can be programmed to recognize that specific PCI memory regions support prefetching. In this case, when the '591 performs the PCI memory read, it asserts all four byte enables and fetches the entire doubleword being addressed in the data phase (even if the processor had only requested a subset of the doubleword). The requested data is fed back to the processor and the prefetched bytes within the doubleword are stored in the '591's read-ahead buffer. This buffer can hold a quadword of data. If the processor should subsequently request any of the prefetched data, the data is supplied from the read-ahead buffer and the '591 does not perform a PCI memory read transaction.

Memory Write

When the host processor initiates a memory write transaction, there are the same three cases:

- The target location is within the range of system DRAM memory.
- The target location falls within the range assigned to system DRAM memory, but is within a sub-range defined as PCI or ISA memory.
- The target location is above the top of installed system memory.

In the first case, the memory write is absorbed by the posted-write buffer (if the eight quadword FIFO isn't full). The processor can then initiate another transaction immediately. If the buffer is full, the processor's current memory write stalls.

In the second and third case, there are two possible cases:

- Memory write posting is enabled for the addressed area of PCI memory .
- Memory write posting is disabled for that area.

If write posting is enabled and the write buffer is currently-available, the memory write is absorbed by the buffer and the processor is permitted to end its transaction. The '591 then initiates the PCI memory write when it has acquired PCI bus ownership. If write posting is disabled in the target area, the processor is stalled until the PCI memory write has been completed on the PCI bus.

I/O Read

When the processor initiates an I/O read transaction, the target device is one of the following:

- Configuration address port at I/O location 0CF8h.
- Configuration data port at I/O location 0CFCh.
- A PCI or an ISA I/O target device.

In the first two cases, the transaction is not passed through to the PCI bus. These two ports are integrated into the '591. The '591 therefore supplies the requested data directly to the host processor.

In the third case, the '591 stalls the processor until the PCI target (or the '593) supplies the requested data. The data is then routed to the processor, concluding the transaction.

I/O Write

When the processor initiates an I/O write transaction, the target location is one of the following:

- Configuration address port at I/O location 0CF8h.
- Configuration data port at I/O location 0CFCh.
- A PCI or an ISA I/O target,

In the first two cases, the '591 accepts the write data into the target port and the transaction is not passed to the PCI bus.

In the third case, the transaction must be passed to the PCI bus. By default, the '591 does not post I/O writes, but it can be programmed to do so. Assuming I/O write posting is disabled, the processor is stalled until the '591 acquires ownership of the PCI bus and completes the PCI I/O write transaction.

Interrupt Acknowledge

In response to an external interrupt from an 8259A interrupt controller, the processor generates two, back-to-back interrupt acknowledge transactions. The first one is generated to command the interrupt controller to prioritize its pending requests. The processor does not transfer data during this transaction. The processor generates the second interrupt acknowledge to request the

interrupt vector associated with the highest-priority pending request. When the '591 detects the first interrupt acknowledge, it responds with BRDY# to permit the processor to end the transaction. This transaction is not passed through the bridge. When the '591 detects the initiation of the second interrupt acknowledge, it acquires ownership of the PCI bus and performs an interrupt acknowledge transaction. In response, the '593 internally generates two INTA (interrupt acknowledge) pulses to the two 8259A cores that reside inside the '593. During the second INTA, the interrupt controller gates the one byte vector onto PCI data path zero, AD[7:0], and asserts TRDY#. The '591 is already asserting IRDY# so the '591 latches the vector from the AD bus and terminates the transfer. During this period, the '591 has been stalling the processor by keeping BRDY# deasserted until the data is presented on the processor's data bus. The vector is placed on host data path zero, D[7:0], and BRDY# is asserted. The processor latches the vector, concluding the second interrupt acknowledge transaction.

Special Cycle

The host processor is capable of generating the following types of special cycle transactions:

- Shutdown.
- Flush.
- Halt.
- Writeback.
- Flush Acknowledge.
- Branch Trace Message.
- Stop/Grant.

The '591 only passes shutdown, halt and stop/grant through the bridge to the PCI bus. The shutdown special cycle causes the '591 to generate a PCI special cycle transaction with the shutdown message sent during the data phase. The halt special cycle causes the '591 to generate a PCI special cycle transaction with the halt message sent during the data phase. The stop/grant special cycle causes the '591 to generate a PCI special cycle transaction with the halt message sent during the data phase. The stop/grant message is differentiated from a halt by AD4 set to one during the address phase (rather than low for a halt). The '593 is designed to test the state of AD4 during the address phase to determine if the message is a halt or a stop/grant. The other processor-initiated special cycles have the following effects:

- The flush special cycle transaction causes the '591 to invalidate the L2 cache.
- The writeback special cycle transaction has no effect (because the L2 cache is not a writeback cache and therefore does not have any modified lines to be written back to memory).
- The flush acknowledge special cycle transaction is generated by the processor in response to assertion of its FLUSH# input when it has completed writing back all modified lines to memory and has cleared the L1 cache. The author believes (but isn't certain) that the '591 ignores this transaction.
- If enabled to do so, the processor generates the branch trace message special cycle transaction whenever a branch instruction is taken. The '591 ignores this transaction.

Handling of PCI-Initiated Transactions

General

The following sections describe how the '591 responds to transactions initiated on the PCI bus by other masters. It's important to note that the '591 does not support concurrent operation of the host and PCI buses. In other words, when a PCI master other than the '591 has acquired ownership of the PCI bus, it has also acquired ownership of the host bus. The '591 accomplishes this by asserting HOLD to the processor and waiting until HLDA is asserted, indicating that the processor has released ownership of the host bus. The '591's PCI arbiter then grants ownership of both buses to the PCI master. The transaction initiated by the PCI master is passed through to the host bus so that the PCI master can access system DRAM memory (if this is a memory read or write transaction that targets system DRAM). If the transaction is not a memory transaction, or it is a memory transaction, but the target address is not system DRAM memory, then the DRAM controller, L2 cache and the processor (which is not told to snoop) ignore the transaction.

PCI Master Accesses System DRAM

The '591 aliases all three of the PCI memory read transaction types (memory read, memory read line, memory read multiple) to the PCI memory read transaction. If it is a memory transaction and the target address is system DRAM, the following actions are taken:

- The processor's L1 cache is told to snoop the address (via AHOLD and EADS#) and report the snoop result (on HIT# and HITM#).
- The L2 cache performs a lookup.

If the L1 snoop results in a miss or a hit on a clean line (HITM# is not asserted), the PCI master is permitted to continue with the transaction.

1. If a read and the requested data is present in the L2 cache, the L2 cache supplies the data to the PCI master. This is a nice feature. If the PCI master is accessing a data structure that is shared by the host processor, the resulting L2 cache hits can result in remarkable performance for the PCI master.
2. If a read and the requested data is not present in the L2 cache, the DRAM controller accesses system DRAM and the data is supplied to the PCI master from system dram.
3. If a write, the posted-write buffer absorbs the write data from the PCI master. If the L2 and/or L1 caches have a hit, the cache copies of the line are invalidated. If the data isn't resident in either cache, the write has no effect on the caches. The memory write and invalidate command is treated as a memory write.

PCI Master Accesses PCI or ISA Memory

Although the transaction is presented on the host bus, it has no effect on the L1 or L2 caches or on system memory.

PCI Master Accesses Non-Existent Memory

In this case, the '593 asserts DEVSEL# (due to subtractive decode) and passes the transaction to the ISA bus. The '591 passes the transaction to the host bus, but it has no effect (because the target address is not within range of system DRAM memory).

I/O Read or Write Initiated by PCI Master

Although passed to the host bus by the '591, have no effect.

Special Cycle

The '591 ignores special cycle transactions generated by PCI masters.

Type 0 Configuration Read or Write

A PCI master may access the '591's PCI configuration registers by directly generating the standard type 0 configuration read and write transactions. It cannot use the configuration address and data ports to stimulate the '591 to generate configuration transactions because the '591 would then have to use the PCI bus at the same time that the PCI master was using it.

Type 1 Configuration Read or Write

A PCI master may use a type one configuration transaction to access the configuration registers in a device on another PCI bus or to cause the generation of a special cycle on a specified target PCI bus. The '591 is unaffected by these transaction types.

Dual-Address Command (64-bit Addressing)

Because the system DRAM memory resides in the area up to but not above the 1GB address boundary, the detection of a PCI dual-address command has no effect on the '591.

Support for Fast Back-to-Back Transactions

The '591 can act as the target of fast back-to-back transactions, but cannot initiate them when acting as a PCI master.

'593 PCI/ISA Bridge

The '593 provides the following functionality:

- Bridges the PCI and ISA buses.
- Two 8259A interrupt controllers and the APIC I/O module.
- Two 8237A DMA controllers and their associated page registers.
- Real-Time Clock and CMOS RAM function (or, alternately, supports external RTC/CMOS).
- Port B logic.
- Tunable subtractive decoder (can be tuned to assert DEVSEL# in slow time slot).
- Programmable decoders for memory regions that exists on the ISA bus. This permits fast address decode of PCI accesses to the ISA bus. It also

permits memory accesses initiated by ISA bus masters or DMA channels to be contained on the ISA bus and not be passed to the PCI bus.

- Handles shutdown-to-processor INIT conversion. When a PCI special cycle transaction is detected with the shutdown message, the '593 toggles INIT to force a system reboot.
- A20 mask function.
- Processor self-test initiation.
- FERR# to IRQ13 conversion.
- NMI generation. The '593 generates NMI when CHCHK is asserted on the ISA bus, or when SERR# is asserted on the PCI bus.
- Hot reset generation.
- System Management Mode (SMM) interrupt logic.
- Monitors up to 27 different events related to power management.
- Includes watchdog timer for SMM usage.
- Supports software generation of the system management interrupt.
- Includes logic utilized to stop the processor clock.
- Positive decode for system ROM and keyboard controller, permitting fast address decode within these ranges.
- POWERGOOD/Reset logic.
- Speaker timer.
- Support for turbo mode. Can be used to periodically stop the processor's clock to make the processor appear to run slower.
- Integrated X-bus buffers.
- Can increase ISA BUSCLOCK speed up to 16MHZ within specified ISA memory regions.
- Decoupled ISA memory refresh. ISA memory refresh can occur on the ISA bus while PCI transactions are in progress.
- Supports disabling of internal DMA controllers (permitting implementation of an external DMA controller).

'593 Handling of Transactions Initiated by PCI Masters

The '593 responds to PCI transactions as follows:

- When acting as the target of a multiple-data phase PCI transaction, the '593 always disconnects the master upon completion of the first data phase.

- The '593 treats the PCI memory read line and read multiple commands as a memory read.
- The '593 treats the PCI memory write and invalidate command as a memory write.
- The '593 ignores the PCI dual-address command.
- The '593 can respond as the target of fast back-to-back transactions, but cannot generate them.
- The '593 responds to the PCI interrupt acknowledge transaction by claiming the transaction and returning the interrupt vector on the lower data path.
- When the '593 detects a PCI special cycle transaction, it determines if the message delivered during the data phase is a shutdown or a halt (it ignores all other message types). If a shutdown, it issues INIT to the processor to reboot the system. If a halt, it examines the state of AD4 from the transaction's address phase. If AD4 = 0, the processor is halting. The SMM logic can be programmed to take action on this event. If AD4 = 1, the message is really a stop/grant message. This informs the SMM logic that the processor has stopped its clock in response to assertion of STPCLK# by the '593.
- The '593 ignores type one configuration read and writes. A type zero configuration read or write that asserts the '593's IDSEL is permitted to access the '593's PCI configuration registers.
- The '593 handles address and data phase parity errors according to the revision 2.x PCI specification.
- The posted-memory write buffer in the '593 can be enabled/disabled by software. When enabled, the buffer accepts up to 32-bits of write data being presented by a PCI master and then disconnects. It then performs the memory write on the ISA bus. If a PCI master attempts to access the ISA bus while the posted-write is being performed on the ISA bus, the '593 stalls it (by keeping TRDY# deasserted) until the write completes.

Subtractive Decode Capability

The subtractive decoder in the '593 claims any unclaimed PCI memory access (even to addresses above 16MB). This means that areas of memory space above the top of system DRAM memory and above 16MB and not populated by PCI memory devices alias down into ISA's 16MB memory address space.

The subtractive decoder can be tuned to respond with slow DEVSEL#, rather than the normal subtractive DEVSEL# one clock after the slow time slot.

'593 Characteristics When Acting as PCI Master

The '591 incorporates the PCI bus arbiter. When the '593 must pass an ISA bus master or DMA transaction onto the PCI bus, the '593 must issue a request to the '591 and await assertion of its grant. The '59x chipset can handle the arbitration between the '591 and the '593 in one of two ways (software-selectable):

1. The '593 normally drives its REQGNT# output high. When it requires access to the PCI bus, it drives REQGNT# low for one cycle of the processor's bus clock. One clock after that, the '591 takes ownership of the REQGNT# signal and drives it high. The '591 continues to drive REQGNT# high until the arbiter is going to grant the bus to the '593. The '591 then drives REQGNT# low for one cycle of the processor's bus clock. One clock after that, the '593 resumes ownership of the REQGNT# signal and continues to drive it low until it has completed using the PCI bus. When the '593 has concluded using the PCI bus, it drives REQGNT# high and leaves it high until it requires the PCI bus again.
2. Alternately, the '593 and '591 can use a normal PCI REQ#/GNT# signal pair for '593 bus arbitration.

When acting as the PCI master, the '593 cannot generate fast back-to-back transactions.

The '593 recognizes that the arbiter is parking the bus on it when it detects its GNT# asserted and the bus is idle. The '593 then takes ownership of the AD and C/BE buses and drives them low. One clock after driving them low, the '593 drives PAR low.

The '593 only initiates transactions on the PCI bus because:

- A DMA channel is performing a transfer to or from system memory.
- An ISA bus master is performing a system memory or an I/O read or write.

The only types of PCI transactions it generates are therefore memory and I/O read and write transactions.

Interrupt Support

The '593 incorporates two 8259A interrupt controllers and an advanced programmable interrupt controller (APIC) I/O module. All of the system interrupt request signals, IRQ[15:0], are connected in parallel to the pair of 8259A's and to the APIC. This permits the programmer to set up the '593's interrupt logic to handle requests in an 8259A-compatible manner, or, in a multiprocessing environment, to route all interrupts from ISA and PCI to the APIC for delivery to the processors. A description of 8259A and APIC operation is outside the scope of this book. A complete description of the 8259A interrupt controller operation can be found in the Addison-Wesley publication (also authored by MindShare) entitled *ISA System Architecture*. A complete description of the APIC operation can be found in the Addison-Wesley publication (also authored by MindShare) entitled *Pentium Processor System Architecture*.

Eleven of the system IRQ inputs, IRQ[15:14], [12:9], and [7:3] may be individually programmed as either shareable or non-shareable interrupt request lines. Of these eleven, the '593's four PCI interrupt inputs may be routed to any of eight of them (IRQ[15:14], [12:9], [5], and [3]).

DMA Support

The '593 incorporates two 8237A DMA controllers in a master/slave configuration. It also incorporates the page registers necessary to extend the start address registers to a full 32 bits. This enables the DMA controller to transfer data to or from memory anywhere within the 4GB memory address space. It is a constraint, however, that the specified DMA transfer in memory must reside fully with a 64KB-aligned block of memory space (because the 8237A DMA controller cannot issue a carry to the page register when incrementing over a 64KB address boundary).

'593 Configuration Registers

Figure 23-4 illustrates the '593's usage of its PCI configuration space. The sections that follow define the manner in which the chipset implements each of these registers. For a description of the '593's device-specific configuration registers, refer to the chipset specification.

Vendor ID Register

The vendor ID for VLSI Technology is 1004h.

Device ID Register

The device ID for the '593 is 0006h.

Command Register

Table 23-4 defines the '593's usage of its command register bits.

Table 23-4. '593's Command Register Bit Assignment

Bit	Description
0	**I/O enable bit.** When set one, the '593 is enabled to respond to PCI I/O transactions. Reset sets this bit to one.
1	**Memory enable bit.** When set to one, PCI bus masters can access ISA memory. Reset sets this bit to one.
2	**Master enable bit.** When set to one, the '593 is enabled to initiate PCI transactions. Reset sets this bit to one.
3	**Special cycle monitor enable bit.** When set to one, the '593 recognizes special cycles (only shutdown and halt) generated by other PCI masters (usually, the '591). Reset sets this bit to one.
4	**Memory write and invalidate enable bit.** Hardwired to zero because the '593 never generates the memory write and invalidate command.
5	**VGA color palette snoop enable bit.** Hardwired to zero. Only VGA-compatible devices and PCI-to-PCI bridges are required to implement this bit.
6	**Parity error response bit.** When set to one, the '593 asserts PERR# when a data parity error is detected. Also used to qualify the assertion of SERR# on address phase parity error. Reset clears this bit.
7	**Stepping enable bit.** Hardwired to zero because the '593 never uses address or data stepping.
8	**System error response bit.** When set to one, the '593 is enabled to assert SERR# (if the PARITY ERROR RESPONSE bit is also set to one) when address phase parity error detected. Reset clears this bit.
9	**Fast back-to-back enable bit.** Hardwired to zero because the '593 never performs fast back-to-back transactions.
15:10	**Reserved** and hardwired to zero.

Status Register

The '593's configuration status register bit assignment is defined in table 23-5.

Table 23-5. '593's Status Register Bit Assignment

Bit	Description
6:0	**Reserved** and hardwired to zero.
7	**Fast back-to-back capable bit.** Hardwired to one, indicating that, when acting as a target, the '593 supports fast back-to-back transactions to different targets.
8	**Signaled parity error bit.** Set to one when the '593, acting as a master, samples PERR# asserted by the target during a write or the '593 asserts PERR# on a read. Reset clears this bit to zero.
10:9	**DEVSEL timing.** Hardwired to 01b, indicating that the '593 has a medium speed PCI address decoder.
11	**Signaled target abort.** Set to one when the '593 has signaled a target abort to the initiator of a transaction. Reset clears this bit to zero.
12	**Received target abort.** Set to one when the '593 receives a target abort from a target when acting as master. Reset clears this bit to zero.
13	**Received master abort.** Set to one when the '593 experiences a master abort when acting as master. Reset clears this bit to zero.
14	**Signaled system error.** Set to one by the '593 when it assert SERR#. Reset clears this bit to zero. The SERR# ENABLE and PARITY ERROR RESPONSE bits in the '593's command register must be set to enable the '593 to generate SERR# and set this bit.
15	**Received parity error.** Set to one when the '593 detects an address or data phase parity error. Reset clears this bit to zero.

Revision ID Register

The revision ID register contains 00h in the first release of the '593.

Class Code Register

The class code register contains 060100h. 06h specifies the bridge class. The middle byte, 01h, specifies that the sub-class is ISA/PCI bridge. The lower byte is always 00h for all revision 2.x-compliant devices.

Cache Line Size Configuration Register

Not implemented.

Latency Timer Register

Not implemented. When acting as a PCI master, the '593 only performs single data phase transactions initiated by ISA bus masters and DMA channels.

Header Type Register

Hardwired with the value 00h. This indicates that the '593 is a single-function device (bit 7 = 0) and that the format of configuration doublewords four through 15 adheres to the header type zero definition.

BIST Register

Hardwired to 00h. Bit 7 = 0 indicates that the '593 does not implement a built-in self test.

Base Address Registers

None implemented. The '593 utilizes device-specific registers to set up its ISA, ROM and I/O address decoders.

Expansion ROM Base Address Register

Not implemented because the '593 does not incorporate a PCI device ROM.

Interrupt Line Register

Not implemented because the '593 does not generate interrupt requests for itself.

Interrupt Pin Register

Hardwired with 00h, indicating that the '593 does not implement a PCI interrupt request output pin.

Min_Gnt Register

Hardwired to 00h, indicating that the '593 has no specific requirements regarding the timeslice assigned to it. In addition, the '593 does not implement the LT, so the Min_Gnt register is a moot point.

Max_Lat Register

Hardwired to 00h, indicating that the '593 has no specific requirements regarding its arbitration priority level.

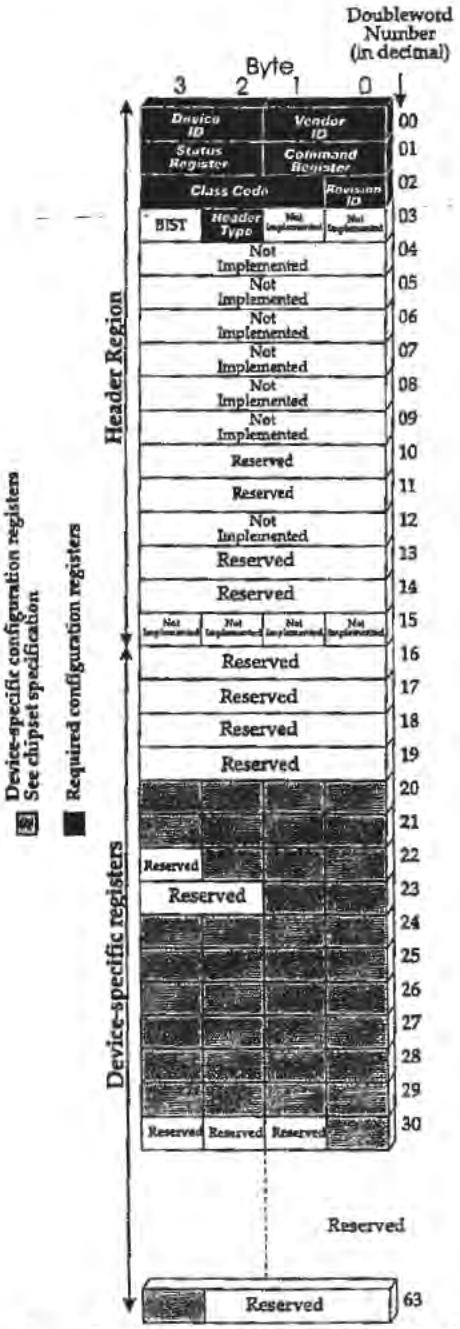


Figure 23-4. VL82C593's Configuration Registers

Access Latency. 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.

AD Bus. The PCI address/data bus carries address information during the address phase of a transaction and data during each data phase.

Address Ordering. During PCI burst memory transfers, the initiator must indicate whether the addressing sequence will be sequential (also referred to as linear) or will use cacheline wrap ordering of addresses. The initiator uses the state of AD[1:0] to indicate the addressing order. During I/O accesses, there is no explicit or implicit address ordering. It is the responsibility of the programmer to understand the I/O addressing characteristic of the target device.

Address Phase. During the first clock period of a PCI transaction, the initiator outputs the start address and the PCI command. This period is referred to as the address phase of the transaction. When 64-bit addressing is used, there are two address phases.

Agents. Each PCI device, whether a bus master (initiator) or a target is referred to as a PCI agent.

Arbiter. The arbiter is the device that evaluates the pending requests for access to the bus and grants the bus to a bus master based on a system-specific algorithm.

Arbitration Latency. 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 system utilization.

Atomic Operation. A series of two or more accesses to a device by the same initiator without intervening accesses by other bus masters.

Base Address Registers. Device configuration registers that define the start address, length and type of memory space required by a device. The type of space required will be either memory or I/O. The value written to this register during device configuration will program its memory or I/O address decoder to detect accesses within the indicated range.

BIST. Some integrated devices (such as the i486 microprocessor) implement a built-in self-test that can be invoked by external logic during system start up.

Bridge. The device that provides the bridge between two independent buses. Examples would be the bridge between the host processor bus and the PCI bus, the bridge between the PCI bus and a standard expansion bus (such as the ISA bus) and the bridge between two PCI buses.

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.

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 and the requesting bus master can 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 will take to complete.

Bus Concurrency. Separate transfers occurring simultaneously on two or more separate buses. An example would be an EISA bus master transferring data to or from another EISA device while the host processor is transferring data to or from system memory.

Bus Idle State. A transaction is not currently in progress on the bus. On the PCI bus, this state is signalled when FRAME# and IRDY# are both deasserted.

Bus Lock. Gives a bus master sole access to the bus while it performs a series of two or more transfers. This can be implemented on the PCI bus, but the preferred method is resource locking. The EISA bus implements bus locking..

Bus Master. A device capable of initiating a data transfer with another device.

Bus Parking. An arbiter may grant the buses to a bus master when the bus is idle and no bus masters are generating a request for the bus. If the bus master that the bus is parked on subsequently issues a request for the bus, it has immediate access to the bus.

Byte Enable. I486, Pentium™ or PCI Bus control signal that indicates that a particular data path will be used during a transfer. Indirectly, the byte enable

signal also indicates what byte within an addressed doubleword (or quadword, during 64-bit transfers) is being addressed.

Cache. A relatively small amount of high-speed Static RAM (SRAM) that is used to keep copies of information recently read from system DRAM memory. The cache controller maintains a directory that tracks the information currently resident within the cache. If the host processor should request any of the information currently resident in the cache, it will be returned to the processor quickly (due to the fast access time of the SRAM).

Cache Controller. See the definition of **Cache**.

Cache Line Fill. When a processor's internal cache, or its external second level cache has a miss on a read attempt by the processor, it will read a fixed amount (referred to as a line) of information from the external cache or system DRAM memory and record it in the cache. This is referred to as a cache line fill. The size of a line of information is cache controller design dependent.

Cache Line Size. See the definition of **Cache Line Fill.**

CacheLine Wrap Mode. At the start of each data phase of the burst read, the memory target increments the doubleword address in its address counter. When the end of the cache line is encountered and assuming that the transfer did not start at the first doubleword of the cache line, the target wraps to start address of the cacheline and continues incrementing the address in each data phase until the entire cache line has been transferred. If the burst continues past the point where the entire cache line has been transferred, the target starts the transfer of the next cache line at the same address that the transfer of the previous line started at.

CAS Before RAS Refresh, or CBR Refresh. Some DRAMs incorporate their own row counters to be used for DRAM refresh. The external DRAM refresh logic has only to activate the DRAM's CAS line and then its RAS line. The DRAM will automatically increment its internal row counter and refresh (recharge) the next row of storage.

CBR Refresh. See the definition of **CAS Before RAS Refresh.**

Central Resource Functions. Functions that are essential to operation of the PCI bus. Examples would be the PCI bus arbiter and "keeper" pullup resistors

that return PCI control signals to their quiescent state or maintain them at the quiescent state once driven there by a PCI agent.

Claiming the Transaction. An initiator starts a PCI transaction by placing the target device's address on the AD bus and the command on the C/BE bus. All PCI targets latch the address on the next rising-edge of the PCI clock and begin to decode the address to determined if they are being addressed. The target that recognizes the address will "claim" the transaction by asserting DEVSEL#.

Class Code. Identifies the generic function of the device (for example, a display device) and, in some cases, a register-specific programming interface (such as the VGA register set). The upper byte defines a basic class type, the middle byte a sub-class within the basic class, and the lower byte may define the programming interface.

Coherency. If the information resident in a cache accurately reflects the original information in DRAM memory, the cache is said to be coherent or consistent.

Commands. During the address phase of a PCI transaction, the initiator broadcasts a command (such as the memory read command) on the C/BE bus.

Compatibility Hole. The DOS compatibility hole is defined as the memory address range from 80000h - FFFFFh. Depending on the function implemented within any of these memory address ranges, the area of memory will have to be defined in one of the following ways: Read-Only, Write-Only, Read/Writable, Inaccessible.

Concurrent Bus Operation. See the definition of **Bus Concurrency.**

Configuration Access. A PCI transaction to read or write the contents of one of a PCI device's configuration registers.

Configuration Address Space. x86 processors possess the ability to address two distinct address spaces: I/O and memory. The PCI bus uses I/O and memory accesses to access I/O and memory devices, respectively. In addition, a third access type, the configuration access, is used to access the configuration registers that must be implemented in all PCI devices.

Configuration CMOS RAM. The information used to configure devices each time an ISA, EISA or Micro Channel™ machine is powered up is stored in battery backed-up CMOS RAM.

Configuration Header Region. Each functional PCI device possesses a block of two hundred and fifty-six configuration addresses reserved for implementation of its configuration registers. The format, or usage, of the first sixty-four locations is predefined by the PCI specification. This area is referred to as the device's configuration header region.

Consistency. See the definition of **Coherency.**

Data Packets. In order to improve throughput, a PCI bridge may consolidate a series of single memory reads or writes into a single PCI memory burst transfer.

Data Phase. After the address phase of a PCI transaction, a data item will be transferred during each data phase of the transaction. The data is transferred when both TRDY# and IRDY# are sampled asserted.

Deadlock. A deadlock is a condition where two masters each require access to a target simultaneously, but the action taken by each will prevent the desired action of the other.

Direct-Mapped Cache. In a direct-mapped cache, the cache is the same size as a page in memory. When a line of information is fetched from a position within a page of DRAM memory, it is stored in the same relative position within the cache. If the processor should subsequently request a line of information from the same relative position within a different page of memory, the cache controller will fetch the new line from memory and must overwrite the line currently in the cache.

Dirty Line. A write-back cache controller has cached a line of information from memory. The processor subsequently performs a memory write to a location within the cache line. There is a hit on the cache and the cache line is updated with the new information, but the cache controller will not perform a memory write bus cycle to update the line in memory. The line in the cache no longer reflects the line in memory and now has the latest information. The cache line is said to be "dirty" and the memory line is "stale." Dirty is another way of saying "modified."

Disconnect. A very slow access target may force a disconnect between accesses to give other initiators a chance to use the bus. This is known as a disconnect. If the current access is quite long and will consume a lot of bus time, the target signals a disconnect, completes the current data phase, and the initiator terminates the transaction. The initiator then arbitrates for the bus again so that it may re-initiate the transaction, continuing the data transfer at the point of disconnection.

DOS Compatibility Hole. See the definition of **Compatibility Hole.**

DRAM controller. The DRAM controller converts memory read or write bus cycles on the host or PCI bus to the proper sequence of actions on the memory bus to access the target DRAM location.

EISA. Extension to Industry Standard Architecture. The EISA specification was developed to extend the capabilities of the ISA machine architecture to support more advanced features such as bus arbitration and faster types of bus transfers.

Exclusive Access. A series of accesses to a target by one bus master while other masters are prevented from accessing the device.

Expansion ROM. A device ROM related to a PCI function. This ROM typically contains the initialization code and possibly the BIOS and interrupt service routines for its associated device.

Functional PCI Devices. A PCI device that performs a single, self-contained function. Examples would be a video adapter or a serial port. A single, physical PCI component might actually contain from one to eight PCI functional devices.

Hidden bus arbitration. Unlike some arbitration schemes, the PCI scheme allows bus arbitration to take place at the same time that the current PCI bus master is performing a data transfer. No bus time is wasted on a dedicated period of time to perform an arbitration bus cycle. This is referred to as hidden arbitration.

Hidden Refresh. If a DRAM controller uses the memory bus to perform DRAM refresh when the system is currently accessing a device other than DRAM memory, this is referred to as hidden refresh. Refreshing DRAM in this fashion doesn't impact system performance.

Hierarchical PCI Buses. When one PCI bus is subordinate to another PCI bus, they are arranged in a hierarchical order. A PCI-to-PCI bridge would interconnect the two buses.

Hit. Refers to a hit on the cache. The processor initiates a memory read or write and the cache controller has a copy of the target line in its cache.

Host/PCI Bridge. A device that provides the bridge between the host processor's bus and a PCI bus. The bridge provides transaction translation in both directions. It may also provide data buffering and/or a second-level cache for the host processor.

Idle State. See the definition for the **Bus Idle State.**

Incident-Wave Switching. See the definition of **Class I Driver.**

Initiator. When a bus master has arbitrated for and won access to the PCI bus, it becomes the initiator of a transaction. It then starts a transaction, asserting FRAME# and driving the address and command onto the bus.

Interrupt Acknowledge. The host ix86 processor responds to an interrupt request on its INTR input by generating two, back-to-back interrupt acknowledge bus cycles. If the interrupt controller resides on the PCI bus, the host/PCI bridge translates the two cycles into a single PCI interrupt acknowledge bus cycle. In response to an interrupt acknowledge, the interrupt controller must send the interrupt vector corresponding to the highest priority device generating a request back to the processor.

Interrupt Controller. A device requiring service from the host processor generates a request to the interrupt controller. The interrupt controller, in turn, generates a request to the host processor on its INTR signal line. When the host processor responds with an interrupt acknowledge, the interrupt controller prioritizes the pending requests and returns the interrupt vector of the highest priority device to the processor.

Level-Two, or L2, Cache. The host processor's internal cache is frequently referred to as the primary, or level-one cache. An external cache that attempts to service misses on the internal cache is referred to as the level-two cache.

Line. See the definition of **Cache Line Fill.**

Line Buffer. If a device has a buffer that can hold an entire line of information previously fetched from memory, the buffer is frequently referred to as a line buffer.

Linear Addressing. If a PCI master initiates a burst memory transfer and sets AD[1:0] equal to a 00b, this indicates to the addressed target that the memory address should be incremented for each subsequent data phase of the transaction.

Local Bus. Generally, refers to the processor's local bus structure. An example would be the 486's bus structure.

Look-Through Cache Controller. A cache controller that resides between its associated processor and the rest of the world is referred to as a look-through cache controller. Look-through cache controllers are divided into two categories: write-through and write-back.

Master Abort. If an initiator starts a PCI transaction and the transaction isn't claimed by a target (DEVSEL# asserted) within five PCI clock periods, the initiator aborts the transaction with no data transfer.

Master Latency Timer. Each bus master must incorporate a Latency Timer, or LT. The LT benefits both the current bus master and any bus master that may request access to the PCI bus while the current bus master is performing a transaction. The LT ensures that the current bus master will not hog the bus if the PCI bus arbitrator indicates that another PCI master is requesting access to the bus. Looking at it from another point of view, it guarantees the current bus master a minimum amount of time on the bus before it must surrender it to another master.

Master-Initiated Termination. The LT count may have expired and the transaction continued without preemption by the arbitrator (removal of its grant) because no other PCI master required the bus. Another bus master may then request and be granted the bus while the current master still has a transaction in progress. Upon sensing the removal of its grant by the arbitrator, the current bus master must initiate transaction termination at the end of the current data transfer with the target. This is referred to as master-initiated termination.

Memory Read Command. The PCI memory read command should be used when reading less than a cache line from memory.

Memory Read Line Command. This PCI command should be used to fetch one complete cache line from memory.

Memory Read Multiple Command. This command should be used to fetch multiple memory cache lines from memory. When this command is used, the target memory device should fetch the requested cache line from memory. When the requested line has been fetched from memory, the memory controller should start fetching the next line from memory in anticipation of a request from the initiator. The memory controller should continue to prefetch lines from memory as long as the initiator keeps FRAME# asserted.

Memory Write Command. The initiator may use the PCI memory write or the memory write and invalidate command to update data in memory.

Memory Write and Invalidate Command. The memory write and invalidate command is identical to the memory write except that it transfers a complete cache line during the current transaction. The initiator's configuration registers must allow specification of the line size during system configuration. If, when snooping, the cache/bridge's write-back cache detects a memory write and invalidate issued by the initiator and has a snoop hit on a line marked as dirty, the cache can just invalidate the line and doesn't need to perform the flush to memory. This is possible because the initiator is updating the entire memory line and all of the data in the dirty cache line is therefor invalid. This increases performance by eliminating the requirement for the line flush.

Message. An initiator may broadcast a message to one or more PCI devices by using the PCI Special Cycle command. During the address phase, the AD bus is driven to random values and must be ignored. The initiator does, however, use the C/BE lines to broadcast the special cycle command. During the data phase, the initiator broadcasts the message type on AD[15:0] and an optional message-dependent data field over AD[31:16]. The message and data are only valid during the clock after IRDY# is asserted. The data contained in, and the timing of subsequent data phases is message dependent.

Miss. Refers to a miss on the cache. The processor initiates a memory read or write and the cache controller does not have a copy of the target line in its cache.

Multi-Function Devices. A physical PCI component may have one or more independent PCI functions integrated within the package. A component that incorporates more than one function is referred to as a Multi-Function Device.

Non-Cacheable Memory. Memory whose contents can be altered by a local processor without using the bus should be designated as non-cacheable. A cache somewhere else in the system would have no visibility to the change and could end up with stale data and not realize that it no longer accurately reflects the contents of memory.

Packets. See the definition of **Data Packets.**

Page Register. The upper portion of the start memory address for a DMA transfer is written to the respective DMA channel's Page register. The contents of this register is then driven onto the upper part of the address bus during a DMA data transfer.

Parking. See the definition of **Bus Parking.**

PCI. See the definition of **Peripheral Component Interconnect.**

Peer PCI Buses. PCI buses that occupy the same ranking in the PCI bus hierarchy (with respect to the host bus) are referred to as peer PCI buses.

Peripheral Component Interconnect (or PCI). Specification that defines the PCI bus. This bus is intended to define the interconnect and bus transfer protocol between highly-integrated peripheral adapters that reside on a common local bus on the system board (or add-in expansion cards on the PCI bus).

Physical PCI Device. See the definition of **Functional PCI Devices.**

Point-To-Point Signals. Signals that provide a direct interconnect between two PCI agents. An example would be the REQ# and GNT# lines between a PCI bus master and the PCI bus arbiter.

Posted-Write Capability. The ability of a device to "memorize" or post a memory write transaction and signal immediate completion to the bus master. As long as there is room in the posted-write buffer, this permits the bus master to complete memory writes with no wait states. After posting the write

and signalling completion to the bus master, the device will then perform the actual memory write.

Preemption. Preemption occurs when the arbitrator removes the grant from one master and gives it to another.

Prefetching. Fetching a line of information from memory before a bus master requests it. If the master should subsequently request the line, the target device can supply it immediately. This shields the master from the slow access time of the actual target memory.

Primary Bus. The bus closest to the host processor that is connected to one side of an inter-bus bridge.

Reflected-Wave Switching. The output drivers commonly implemented in highly-integrated components fall into the class II category. They take advantage of the reflected-wave switching characteristic common to high-speed transmission lines and printed circuit traces in order to achieve the input logic thresholds. When a class II output driver transitions a signal line from a logic low to a high, the low-to-high transition is rather weak and only achieves half of the voltage change required to cross the logic threshold. This transition wavefront is transmitted along the trace and is seen sequentially by the input of each device connected to the line. When the wavefront gets to the end of the transmission line, it is reflected back along the trace, effectively doubling the voltage change and thereby boosting the voltage change past the logic threshold. As the wave passes each device's input, the new valid logic level is detected.

Reserved Bus Commands. Several of the PCI bus command codes are reserved for future use. Targets should ignore reserved bus commands.

Resource Lock. The PCI bus implements a signal, LOCK#. It allows a master (a processor) to reserve a particular target for its sole use until it completes a series of accesses to the target. The master then indicates that it no longer requires exclusive access to the target. One of the nice features of the PCI bus is that it permits other masters to use the bus to access targets other than the locked target during the period that a master has locked access to a particular target.

Retry. If a target cannot respond to a transaction at the current time, it will signal "retry" to the initiator and terminate the transaction. The initiator will

respond by ending the transaction and then retrying it later. No data transfer takes place during this transaction. An example of the need for a retry would be if the target is currently locked for exclusive access by another initiator.

SCSI. Small Computer System Interface. A bus designed to offload block data transfers from the host processor. The SCSI host bus adapter provides the interface between the host system's bus and the SCSI bus.

Secondary Bus. The bus further from the host processor that is connected to one side of an inter-bus bridge.

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.

Single-Function Devices. A physical PCI component may have one or more independent PCI functions integrated within the package. A component that incorporates only one function is referred to as a Single-Function Device.

Slave. Another name for the target being addressed during a transaction.

Snooping. When a memory access is performed by an agent other than the cache controller, the cache controller must snoop the transaction to determine if the current master is accessing information that is also resident within the cache. If a snoop hit occurs, the cache controller must take an appropriate action to ensure the continued consistency of its cached information.

Soft-Encoded Messages. The first two message codes, 0000h and 0001h, are defined as SHUTDOWN and HALT. Message code 0002h is reserved for Intel device-specific messages, while codes 0003h - through - FFFFh are reserved. Questions regarding the allocation of the reserved message codes should be forwarded to the PCI SIG. Also see the definition of **Message**.

Special Cycle Command. See the definition of **Message**.

Special Interest Group, or SIG. The PCI SIG manages the specification.

Speedway. Intel performed over 5000 hours of simulations in order to establish the best possible layout of the PCI bus on a high-frequency system board. The result is the Speedway (trademarked by Intel) definition. This

layout may be used for up to ten physical PCI components operating at speeds up to 33MHz.

Stale Information. See the definition of **Dirty Line.**

Standard Form Factor. Also referred to as the long card form factor, the standard form factor defines a PCI expansion board that is designed to fit into existent desktop machines with ISA, EISA or Micro Channel™ card slots.

Streaming Data Procedures. Advanced bus cycle types implemented on the more advanced Micro Channel machines.

Subordinate Bus Number. The subordinate bus number configuration register in a PCI-to-PCI bridge (or a host/PCI bridge) defines the bus number of the highest-numbered PCI bus that exists behind the bridge.

Subtractive Decode. The PCI-to-expansion bus bridge is designed to claim many transactions not claimed by other devices on the PCI bus. If the bridge doesn't see DEVSEL# asserted by a PCI target within four PCI clock periods from the start of a transaction, the bridge may assert DEVSEL# to claim the transaction and then pass the transaction onto the standard expansion bus (such as ISA, EISA or the Micro Channel).

Tag SRAM. The high-speed static RAM used as a directory by a cache controller.

Target. The PCI device that is the target of a PCI transaction initiated by a PCI bus master.

Target Latency. Defined as the period of time until the currently-addressed target is ready to complete the first data phase of the transaction. This period is a function of the access time for the currently-addressed target device.

Target-Abort. If the target detects a fatal error or will never be able to respond to the transaction, it must signal a target-abort (using the STOP# signal). This will cause the initiator to end the transaction with no data transfer and no retry.

Target-Initiated Termination. Under some circumstances, the target may have to end a transfer prematurely. The following are some examples. A very slow access target may force a disconnect between accesses to give other

initiators a chance to use the bus. This is known as a ***disconnect***. If the current access is quite long and will consume a lot of bus time, the target signals a disconnect, completes the current data transfer, and the initiator terminates the transaction. The initiator then arbitrates for the bus again so that it may re-initiate the transaction, continuing the data transfer at the point of disconnection. If a target cannot respond to a transaction at the current time, it will signal ***retry*** to the initiator and terminate the transaction. The initiator will respond by ending the transaction and then retrying it. No data transfer takes place during this transaction. An example of the need for a retry would be if the target is currently locked for exclusive access by another initiator. If the target detects a fatal error or will never be able to respond to the transaction, it may signal a ***target-abort***. This will cause the initiator to end the transaction with no data transfer and no retry.

Turn-Around Cycle. A turn-around cycle is required on all signals that may be driven by more than one PCI bus agent. This period is required to avoid contention when one agent stops driving a signal and another agent begins.

Type One Configuration Access. The type one access is used to configure a device on a lower-level PCI bus (in a system with hierarchical PCI buses).

Type Zero Configuration Access. The type zero access is used to configure a device on the PCI bus the configuration access is run on.

Utility Bus. The utility bus is located on the system board and is a buffered version of the standard expansion bus (ISA, EISA or the Micro Channel). Devices such as the keyboard controller, CMOS RAM, and floppy controller typically reside on the utility bus. This bus is also frequently referred to as the X-bus.

Vendor ID. Every PCI device must have a vendor ID configuration register that identifies the vendor of the device.

VESA. The Video Electronics Standards Association, or VESA, is a consortium of add-in card manufacturers tasked with developing standards for PC device interfacing.

VESA VL Bus. This is the local bus standard developed by the VESA consortium.

Video Electronics Standards Association (VESA). See the definition of **VESA.**

Video Memory. Memory that is dedicated to the storage of the video image to be scanned out to the display device.

VL bus. See the definition of **VESA VL Bus.**

VL Type A Local Bus. This is the direct-connect version of the VESA VL bus. For more information, refer to chapter two.

VL Type B Local Bus. This is the buffered version of the VESA VL bus. For more information, refer to chapter two.

Wait State. A delay of one PCI clock period injected into a PCI data phase because either the initiator (IRDY# deasserted), the target (TRDY# deasserted), or both are not yet ready to complete the data transfer.

Write Miss. The processor initiates a memory write and the cache controller does not have a copy of the target memory location within its cache.

Write-Back Cache. The write-back cache controller is a variant of the look-through cache controller. When the processor initiates a memory write bus cycle, the cache controller determines whether or not is has a copy of the target memory location within its cache. If it does, this is a write hit. The cache controller updates the line of information in its cache, but does not initiate a memory write bus cycle to update the line in DRAM memory. This permits processor-initiated memory writes to complete with no wait states. The cache line is now dirty and the memory line is stale. The cache controller will not flush its dirty lines to memory until later. In the event of a miss, the data is written to memory.

Write-Through Cache. The write-through cache controller is a variant of the look-through cache controller. When the processor initiates a memory write bus cycle, the cache controller determines whether or not is has a copy of the target memory location within its cache. If it does, this is a write hit. The cache controller updates the line of information in its cache, and also writes it through to DRAM memory. This ensures that the cache and DRAM memory are always in sync. In the event of a miss, the data is written to memory.

X-Bus. See the definition of **Utility Bus.**