Study of Advanced Bus Architecture (CPE 602)
Summer '97

Prepared By : Tarak Modi
Advisor : Dr. Wells

Introduction to PCI

In today's operating environments, it is absolutely necessary that large blocks of data be transferred expeditiously. Examples of such subsystems include Graphics video adapters, Full motion video adapters, SCSI host bus adapters, and FDDI network adapters.

Graphics Interface Performance Requirements
The Windows, OS/2, Sun OpenWin and X-Windows user interfaces require extremely fast updates of the graphics image in order to efficiently move, resize and update multiple windows. Normally, the screen image is stored in video RAM, which means that the processor(s) must be able to move/update large blocks of data within video memory very fast.

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 systems typically reside on the SCSI bus. The time required to read/write to these devices over the bus imposes a direct delay on the end user.

Network Adapter Performance Requirements
When a network adapter is used to transfer entire files of information to/from a server, the rate at which this data can be transferred directly affects the system performance.

Expansion Bus Performance Constraints

Majority of the subsystems reside on a PC's expansion bus. Depending on the machine design, this may be a ISA, EISA, or Micro Channel expansion bus. However, all three expansion buses suffer from inadequate data rate for today's applications (as seen later). In some cases, subsystems such as the video adapter have been integrated on the system board giving an illusion (false), that they do not reside on the expansion bus. However, this is not the case. Most of these subsystems reside on a buffered version of the expansion bus called (eXtension to the expansion bus). It is also sometimes called the utility bus. Thus the subsystems are once again bound by the same mediocre data transfer rates.

Expansion Bus Transfer Rate Limitations

ISA Expansion bus
All transfers performed over the ISA bus are synchronized to a 8.33 MHz bus clock signal (BCLK). It takes a minimum of two bus cycles to perform a data transfer (assuming a zero wait state). This equals to 4.165 million transfers per second. Since the data path is only 16 bits wide, a maximum of two bytes can be transferred during each transmission. Thus the theoretical maximum data transfer rate is 8.33 Mbytes per second.

EISA Expansion Bus

Like the ISA bus, All transfers performed over the ISA bus are synchronized to a 8.33 MHz bus clock signal (BCLK). It takes a minimum of one clock cycle to perform a data transfer (assuming that EISA burst mode transfer is supported). This equates to 8.33 million transfers per second. Since the data path is 32 bits wide, a maximum of four bytes may be transferred during each transaction. Thus the theoretical maximum data transfer rate is 33 Mbytes per second.

Micro Channel Architecture Expansion Bus
Presently, the maximum achievable transfer rate on this bus is 40 Mbytes per second (using the 32-bit streaming procedure). This is based o a 10 MHz bus speed with one data transfer taking place at each cycle. Up to 80 and 160 Mbytes per second data transfer rates are possible with the 64-bit streaming and the enhanced 64-bit streaming data procedures.

A Realistic Situation

Consider 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. A large portion of the screen is devoted to display the document under discussion. To emulate a actual face to face discussion, the system must be capable of simulating flipping through the pages at the rate of 10 frames per second. Consider the image resolution of 1280 X 1024 pixels, and a color resolution of 16 million colors (three bytes per pixel). Thus the amount of video memory equates to 3.93216 Mbytes per image. For an update of 10 pages per second the memory update rate would have to be at least 39.3216 Mbytes per second

The video preview portion of the screen is used to display a real-time video image of the video source local to the unit. Let the image resolution be 320 X 240 pixels, and the color resolution be 256 colors (one byte per pixel). The image must be updated at a rate of 30 frames per second equating to a data transfer rate of 2.3 (320 multiplied by 240 multiplied by 30) Mbytes per second.

Each of the two remote video screen areas displays a full motion video image, one from each of the other participant. These images have a resolution of 640 X 480 pixels, and a color resolution of 256 colors. Once again, the image must be updated at a rate of 30 frames per second equating to a data transfer rate of 9.2 (620 multiplied by 480 multiplied by 30) Mbytes per second.

Each of the three video cameras would transfer data at 200 Kbytes per second.

Thus the total data transfer rate required for the system equates to 60.516 (39.3216 + 2.3 + 2*9.2 + .2*3) Mbytes per second.

The Practical Solutions

Local Bus Concept

To maximize the 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. The video adapter is redesigned to minimize/eliminate the number of wait states inserted into each bus cycle when the processor accesses the video memory and I/O registers. The adapter also incorporates a local processor.

There are three basic methods for connecting a device on the microprocessor's local bus

Direct Connect Approach
As the name indicates this approach is very straightforward: The device is connected directly on the local bus.

This method imposes the following 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 protocol or structure is changed)
- 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 interface is difficult.
- The Intel Overdrive processor may cause the device to exhibit aberrant behavior.
- The processor is not permitted to perform transfers with one device while the local bus device is involved in a transfer with another device.

Buffered Approach

The second approach that can be used to connect a local bus device to the processor's local bus is the buffered approach. The bus buffer receives all the local bus signals, thereby permitting fanout to more than one local bus device. Since the devices are electrically isolated from the local bus, only one load (that of the buffer) appears on the local bus. This is the only real advantage over the Direct Connect approach. A major disadvantage of this approach is that the processor's local bus and the buffered local bus are essentially one bus. Thus simultaneous access by multiple bus masters is not possible.

Workstation Approach

This approach is 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. 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 the L2 cache) or a bus master on the I/O bus can access the 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 the main memory. 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 I/O bus design. Only the cache bridge would require redesign.

The Local Buses

VESA VL Bus Solution

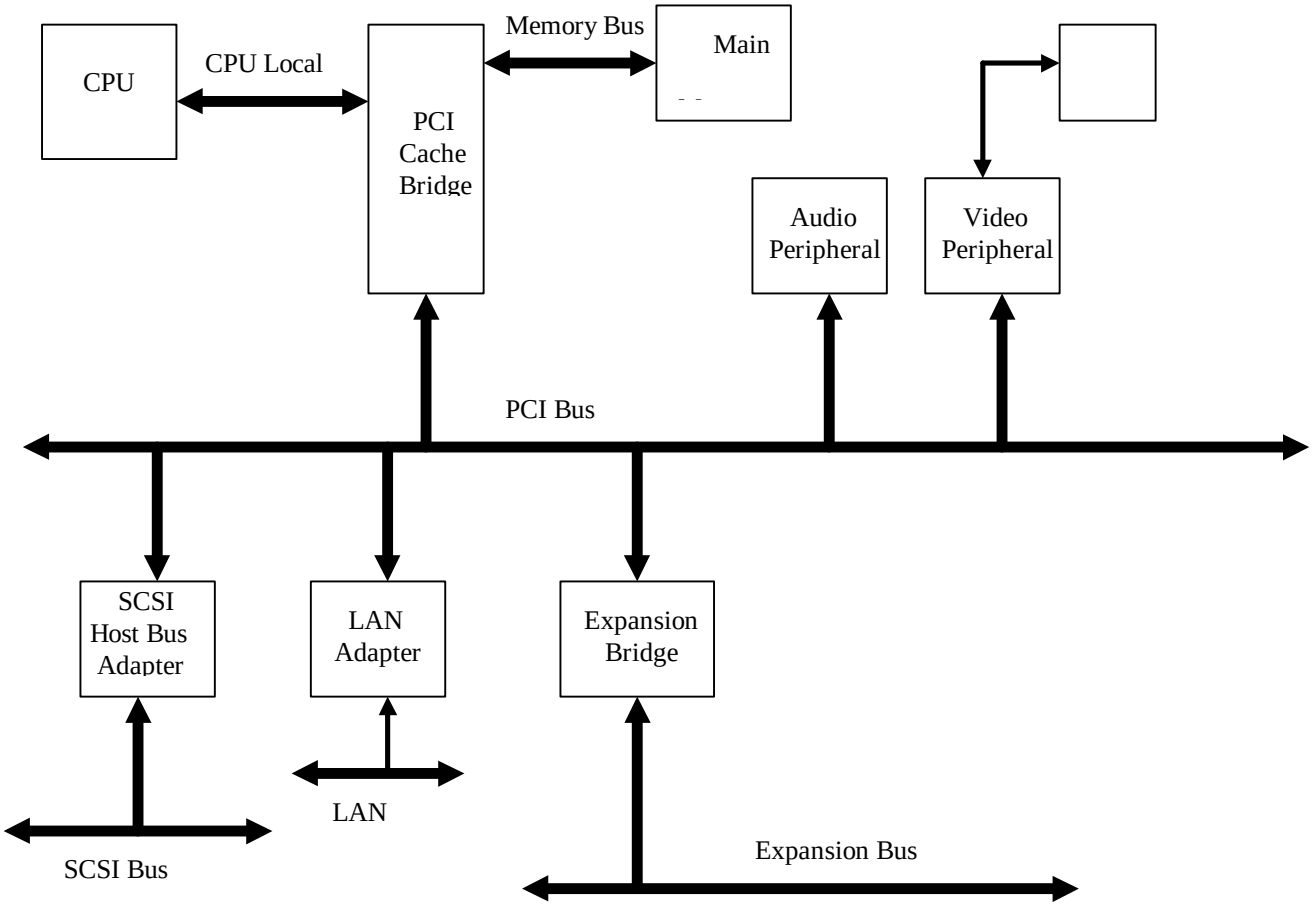
The Video Electronics Standards Association (VESA), an association of companies involved in the design and manufacturing of video graphics adapters commissioned the development of a local bus standard. The local bus specification developed was called the VL (VESA Local) bus. The direct connect interface 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.

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 version became available in June of 1992. Intel made the decision not to back the VESA VL standard because the emerging standard did not take a long term approach to solving the problems presented at the time, and in the future. In addition VL has a very limited support for supporting burst transfers.

PCI stands for Peripheral Component Interconnect. The PCI bus can be populated with adapters requiring fast accesses to each other and system memory with speeds approaching that of the processor's native bus speed. All reads and writes over the PCI are burst transfers.

The PCI design allows the system design to be centered around two of the three approaches discussed earlier : the buffered approach, and the workstation approach. The workstation approach is the preferred of the two due to its flexibility and performance advantages.



The PCI Bus : Basic Relationship of the PCI, Expansion and Memory

The PCI Bus Operation Principles

Burst Transfers

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 the 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 the micro channel environments, the ability to perform burst transfers is the product of negotiation between the bus master and the target device. If either of them do not support burst mode transfers, the packet can only be transferred utilizing a series of separate packets.

PCI data transfers can be accomplished using burst transfers. Many 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 mode transfer, the target terminates the connection at the completion of the first data phase each time, which forces the bus master to re-arbitrate for the bus with each next item. This would seriously impact performance, but may be necessary in some situations.

Assuming that neither the master nor the target inserts wait states, a data object may be transferred on the rising edge of each PCI clock cycle. At a PCI bus clock frequency of 33 MHz, a transfer rate of 132 Mbytes per second may be achieved. This may be doubled in a 64-bit implementation.

PCI Bus Clock
All actions on the PCI bus are synchronized to the PCI clock signal (CLK). The frequency of the CLK may be anywhere from 0 to 33 MHz. Revision 2.1 defines PCI operation up to 66 MHz. All PCI operations consist of an address phase followed by one or more data phases. The exception is where 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 over all burst transfer. Each data phase has a minimum duration of one PCI CLK. Each wait state inserted costs an additional PCI CLK cycle.

Address Phase
Every PCI transaction starts off with an address phase, which is one PCI CLK 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. The PCI target device latches this address and decodes it. By decoding the address and the transaction type, the target device can determine if it is being addressed and the type of transaction in progress.

Claiming the Transaction
Once the target has determined that it is being addressed, it must claim the transaction by asserting DEVSEL#. If the initiator does not sample the DEVSEL# signal within the predetermined amount of time it aborts the transaction.

Data Phase(s)
The data phase 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 Command/Byte Enable signals that are asserted by the initiator during the data phase.
Both the initiator and the target must indicate readiness to complete a data phase, or else the data phase is extended by one wait (PCI CLK) state. The PCI bus defines ready lines used by the initiator and the target as IRDY# and TRDY#. Once the initiator asserts the IRDY# signal, the FRAME# signal becomes de-asserted. This indicates the last data transfer and the PCI bus is returned to idle state.

Reflected Wave Switching

Consider the case where a signal trace is fed by a driver and is attached to a number of device inputs distributed along the signal trace. In the past, the system designer would ignore the electrical characteristics of the trace itself, and only factor in the electrical characteristics of the devices connected to the trace. This was acceptable in the low frequency (1 MHz) range, however in the high frequency environments such as the PCI, traces must switch states at the rate of 25 MHz and up. At these rates, each trace acts like a transmission line and the electrical characteristics of the trace must be factored in the equation.

A transmission line presents an impedance to the driver attempting to drive the voltage change onto the trace, and also imposes a time delay. The trace impedance typically ranges from 50 to 110 ohms

Unlike many buses, the PCI bus does not incorporate termination resistors at the physical end of the bus to absorb voltage changes and prevent the wave front caused by a voltage change from being reflected back down the bus. Rather, PCI uses reflections to its advantage.

The PCI bus is unterminated and uses wavefront reflection to an advantage. A carefully selected , relatively weak output driver is used to drive the signal line halfway to the desired logic state. As an example, the driver would only have to drive the signal from 0 to 1.5 V. As the wavefront passes each device input along the trace, the voltage change is insufficient to register as a logic high. However when the wavefront arrives at the unterminated end of the bus, it is reflected back and doubled to three volts. Upon passing each device input a valid logic "1" is registered. The wavefront is now absorbed by the low impedance inside the driver. This method cuts the driver size and the surge current in half. In many systems the correct operation of the PCI depends on diodes embedded within devices to limit reflections and to successfully meet the specified propagation delay. If a system has long trace runs without connection to a PCI component (e.g. A series of unpopulated connectors), it may be necessary to add diode terminators at the end of the bus to ensure signal quality.

The PCI specification states that devices must only sample their inputs on the rising edge of the PCI clock signal. The physical layout of the PCI bus traces are very important to ensure that signal propagation is within assigned limits. When a driver asserts/se-asserts a signal, the wavefront must propagate to the physical end of the bus, reflect back and make the full passage back down the bus before the signal is sampled on the next rising edge of the PCI CLK signal.

PCI Bus Signals

A PCI device can act as an initiator or as a target device or both. The PCI bus signals may be divided into the following functional groups :

- System Signals
- Address/Data Bus Signals
- Transaction Control Signals
- Arbitration Signals
- Interrupt Request Signals
- Error Reporting Signals
- Cache Support Signals
- Other 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 the 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. The CLK signal my be anywhere from 0 to 33 MHz. It may be varied at any time provided :

- 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

- CLKRUN# Signal

The CLKRUN# signal is optional and is defined for the mobile environment. It is not available on the PCI add-in connector.

- 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 CLK edge. The assertion of RST# also initializes other device specific functions. All PCI outputs must be driven to their benign states. In general, this means they must be tri-stated. Exceptions include

- 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 during reset

Address/Data Bus Signals

- AD Bus (AD[31:0])

This bus 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

- Command or Byte Enable Bus (C/BE#[3:0])

It defines the type of transaction

- Parity Signal (PAR)

This signal is driven by the initiator one clock after completion of the address phase or one clock after the assertion of IRDY# during each data phase of write transactions. It is driven by the currently addressed target one clock after the assertion of TRDY# during each data phase of read transactions. One clock after the completion of the address phase, the initiator drives PAR either high or low to ensure even parity across the address bus, and the four command/Byte enable lines

Thus during each data phase :

- The data bus 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 the assertion of IRDY# or TRDY# during each 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 into all data paths. This is necessary because PAR must reflect even parity over the entire AD and C/BE buses.
- The C/BE bus is driven by the initiator to indicate the bytes to be transferred within the currently addressed doubleword and the data paths to use to transfer the data.

Transaction Control Signals

Signal	Master	Target	Description
FRAME#	In/Out	In	Cycle Frame is driven by the current initiator and indicates the start (when its first asserted) and the 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 and GNT# asserted 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. A data transfer 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 means that the target is driving valid data on the data bus. During a write, TRDY# asserted means that the target is ready to accept data from the master. Wait states are inserted until both TRDY# and IRDY# are sampled asserted.
IRDY#	In/Out	In	Initiator ready is driven by the current bus master. During a write, IRDY# asserted means that the initiator is driving valid data on the data bus. During a read, IRDY# asserted means that the initiator is ready to accept data from the target. In order to determine that bus ownership has been acquired, the master must sample FRAME# and IRDY# both deasserted and GNT# asserted on the same rising-edge of the PCI CLK signal.
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.
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)
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. If a initiator initiates a transfer, and does not detect an asserted DEVSEL# within six PCI CLK signals, it must assume that the target can not 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 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 can not assume ownership of the PCI bus until FRAME# is sampled deasserted (indicating the start of the last data phase), and IRDY# is then sampled deasserted (indicating the completion of the last data phase). This indicates that the current transaction has been completed and the bus has been returned to the idle state.

Interrupt Request Signals

PCI agents that must generate request for service can utilize one of the PCI interrupt request lines, INTA#, INTB#, INTC#, or INTD#.

Error Reporting Signals

- Data Parity Error

The generation of parity information is mandatory for all PCI devices that drive address or data information onto the CA 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 video pixels displayed on the screen.
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 the initiator on detection of the error is design dependent.

- System Error

The system error signal, SERR#, may be pulsed by any PCI to report address parity errors, data parity errors during a special cycle, and critical errors other than parity. 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.

Cache Support Signals

Signal	Description
SBO#	Snoop back off. This signal is an output from the PCI cache and 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 and an input to cacheable memory subsystems residing on the PCI bus. It is deasserted by the bridge while the processor's cache 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#. SBO# sampled deasserted indicates that the PCI indicator 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 indicator 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.

Other Signals

The PCI specification provides a detailed definition of a 64-bit extension to its base line 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.

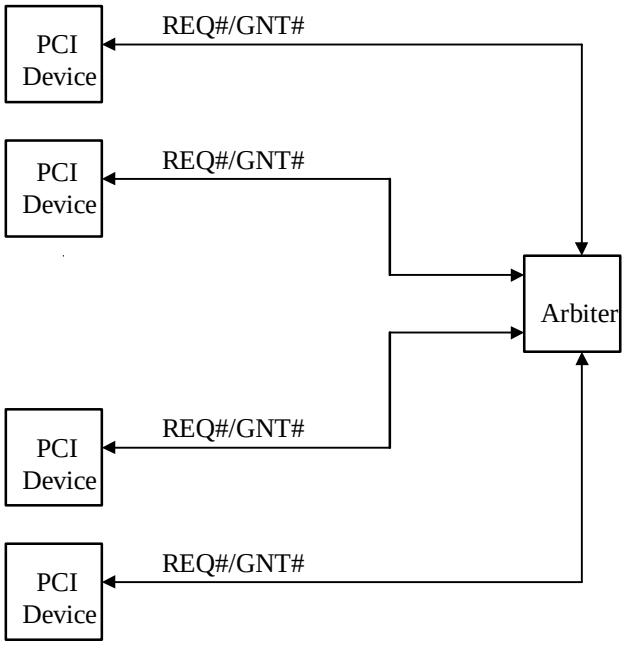
The designer of a PCI device may optionally implement the IEEE 1149.1 boundary scan interface signals to permit in-circuit testing of the PCI device. The boundary scan signals include TCK (Test Clock), TDI (Test Data In), TDO (Test Data Out), TMS(Test Mode Set), and TRST# (Test Reset).

PCI Bus Arbitration

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 PCI specification defines the timing of the request and the grant handshaking, but not the procedure used to determine the winner of the bus access competition. The algorithm used to decide this is therefore system specific.

The 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 to use the bus.



The PCI Bus

As shown above the each possible PCI bus master is connected to the PCI bus arbiter via a separate pair of REQ#/GNT# signals. Although the bus arbiter is shown as a separate component, it is usually integrated into the host /PCI or PCI/expansion bus bridge chip.

Arbitration Algorithm

As stated earlier, the PCI specification does not specify 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 any combination of schemes. The specification for PCI version 2.1 states that the arbiter is required to implement a **fairness** algorithm to avoid deadlocks. The exact verbiage 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 agent attempting to establish a lock is terminated with retry.*

Fairness is defined as a policy in which high priority masters will not dominate the bus to the point of exclusion of low priority masters when they are continually requesting the bus.

Ideally, the bus arbiter should be programmable by the system. The startup configuration software can determine the priority to be assigned to each bus master by reading from the maximum latency (Max_Lat) register within each bus master. The device designer hardwrites this register value, in increments of 250 ns, how often the device requires bus access to achieve adequate performance.

When a potential master is granted access by the arbiter, the GNT# input to that master is asserted. This grants access for one transaction consisting of one or more data phases. If a master is granted access and the master does not initiate a transaction (assert FRAME#) within 16 PCI clocks, the arbiter assumes the master is malfunctioning. Once again, the action taken by the arbiter in such a case is system design dependent.

An Example of an Arbiter with Fairness (From PCI Specification)

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

1. Bus masters that require fast access to the bus in order to achieve good performance, such as a video adapter, ATM (Asynchronous Transfer Mode, not Automated Teller Machine) interface, or a FDDI interface.
2. Bus masters that do not require very fast access to the bus such as a SCSI host bus adapter.

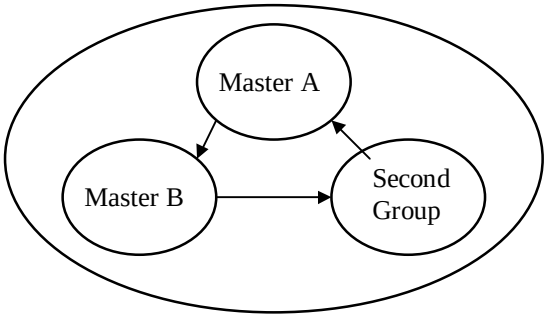
The arbiter segregates the REQ#/GNT# signals into two groups with greater precedence given to those in one group.

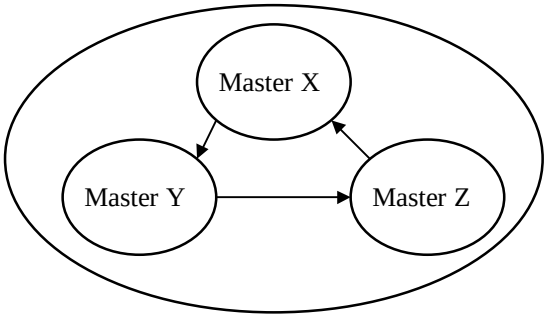
In the example we assume that masters A and B are in the group requiring fast access, while masters X, Y and Z are in the other group. The arbiter may be programmed to treat each group as rotational priority within and between the groups.

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 REQ# asserted even after starting a transaction.

First Group





Second Group

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

The masters in the first group are allowed to access the bus more frequently than the masters in the second group.

Bus Arbitration Issues

· **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 the FRAME# output. This lets the arbiter know of its desire. Depending on other pending requests, the arbiter may or may not permit the master to retain ownership of the bus after the current transaction. The master should keep its REQ# output asserted until it obtains bus ownership again and it has no more bursts to perform. At any instant in time, only one master may use the bus. This means that only one GNT# line will be asserted at a time.

· **Hidden Bus Arbitration**

Unlike some arbitration schemes, the PCI scheme allows bus arbitration to take place while the current master is performing a data transfer. If the arbiter decides to grant ownership of the bus to another master it takes the GNT# signal from the current master and issues it to the next owner of the bus. The next owner however, can not assume bus ownership, until the bus is idled by the current master. No bus time is wasted for a dedicated bus arbitration.

· **Bus Parking**

A master must only assert REQ# output to signal a current need for the bus. It is not allowed to assert REQ# to "park" the bus on itself. However a default bus owner may be designated by the designer by implementing the arbiter logic so that it asserts the GNT# line for that master when no other REQ# lines are asserted.

The specification states that the bus should parked on the last master that acquired the bus. When the arbiter parks the bus on a master (by asserting its grant) and the bus is idle, the master becomes responsible for keeping the AD bus, C/BE bus and PAR from floating. The master must enable its AD[31:0], C/BE[3:0], and one clock later the PAR output drivers. This procedure ensures that the bus doesn't float during bus idle periods. If the arbiter is not designed to park the bus, it should drive the AD bus, C/BE bus and PAR lines itself when the bus is idle.

Request/Grant Timing

When the arbiter determines that it is a masters turn to use the bus, it asserts the master's GNT# line. It may deassert a master's GNT# line on any PCI clock. A master must ensure that its GNT# line is asserted on the rising edge of the PCI CLK. Once the GNT# line is asserted, it 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 the GNT# line by the arbiter tells the master that it will no longer own the bus after the current transaction. The master keeps FRAME# asserted while the current transaction is in progress, and deasserts it when it is ready to complete the final data phase.
- GNT# may be deasserted during the final data phase in response to the current master's REQ# signal being deasserted.
- The GNT# to one master may be deasserted simultaneously with the assertion of another master's GNT#, if the bus is not in the idle state. Idle state is the state in which both FRAME# and IRDY# are deasserted.

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

Arbitration Example Between Two Masters

Assumptions :

- Bus master A requires the bus to perform two transactions. The first consists of a three data phase write and the second consists of a single data phase write.
- The arbitration scheme is fixed and bus master B has a higher priority than bus master A
- Bus master B only requires the bus to execute a single transaction consisting of one data phase.

All PCI signals are sampled on the rising edge of the PCI CLK.

Sample Arbitration Sequence :

1. Bus master A asserts its REQ# line prior to CLK 1. The arbiter samples this request at the rising edge of CLK 1. Master B still doesn't have its REQ# asserted. During CLK 1 the arbiter grants access to A by asserting the GNT# line to A. During CLK 1 master B asserts its REQ# line.
2. Bus master A samples its GNT# line asserted during the rising edge of CLK 2. It also samples FRAME# and IRDY# deasserted indicating that the bus is idle. A asserts FRAME# and begins to drive the address on AD[31:0] and the command on C/BE[3:0]. It also keeps its REQ# line asserted.
3. On the rising edge of CLK 2 the arbiter samples the requests from A and B, and starts the arbitration process to determine the next bus master.
4. During CLK 2 the arbiter removes GNT# from A
5. During the rising edge of CLK 3 master A realizes that it has been preempted, but since the LT timer has not expired it continues with the current transaction. Also, the GNT# line for B is asserted by the arbiter.
6. On the rising edge of CLK 4 master B samples the asserted GNT# signal, indicating that it may be the next owner. B **must** continue to sample the signal until it actually acquires the bus.
7. During CLK 3 master A drives the first data item on the AD bus, and asserts the appropriate C/BE lines. At the rising edge of CLK4, both IRDY# and TRDY# are sampled asserted and the first data transfer occurs. The second data transfer occurs at the rising edge of CLK 5. Also during CLK 5 FRAME# is deasserted. At the rising edge of CLK 6 the third and final data transfer occurs.
8. During CLK 6 master A deasserts IRDY#, thus returning the bus to idle state.
9. On the rising edge of CLK 7, master B samples FRAME# and IRDY# both deasserted and determines that the bus is now in idle state. It also samples GNT# still asserted. It deasserts REQ# line and asserts the FRAME# line. Also during CLK 7 master B starts driving the address and the command buses.

10. At the rising edge of CLK 8, the arbiter samples the REQ# line of A asserted and that of B deasserted. The arbiter deasserts B's GNT# line and asserts A's GNT# line.
11. During CLK 8, master B deasserts its FRAME# line indicating that it has reached the final stage (first and only in this example) of its data phase. It also drives the data and command lines to the appropriate values.
12. On the rising edge of CLK 9 master B samples IRDY# and TRDY# asserted and transfers the data. It then deasserts the IRDY# line.
13. On the rising edge of CLK 10, master A samples the FRAME# and IRDY# lines deasserted and its GNT# line asserted indicating that it has ownership of the bus. It deasserts its REQ# line, and asserts the FRAME# line. Address and Data phases of the transaction proceed as usual.

Access Latency

Arbitration Latency
It is the period of time from the bus master's assertion of REQ# until the bus arbiter asserts the master's GNT#. It depends on the arbitration algorithm, the master's priority, and the number of masters requesting ownership of the bus.

Bus Acquisition Latency
It is the period of time from the reception of GNT# by the requesting bus master until the current bus master surrenders the bus.

Target Latency
It is 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. It depends on the access time of the currently accessed target device.

Bus 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

Bus Access Latency = Arbitration Latency + Bus Acquisition Latency + Target Latency

PCI Bus masters should always use burst transfers to transfer blocks of data between themselves and target PCI devices. In order to insure that the designers 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 (LT)
The LT is implemented inside the bus masters configuration space, and is either initialized by the configuration software at startup or contains a hardwired value. This value is the minimum value of time that the bus master is allowed to have ownership of the bus. When the bus arbiter grants bust ownership to another master while the current master is in the midst of a burst transfer (the current master is preempted) the bus master may keep ownership until one of the following :

- It completes its burst transaction
- The LT timer expires

Even after the LT expires, the current master is allowed one more data transfer after which it must relinquish control of the bus.

Any master that performs more than two data phases per transaction **must** implement the LT.
If the LT value is hard wired, it must not exceed 16.

Target Initiated Termination
A target with very slow access time may monopolize the bus, while a data item is being transferred between it and the master. The PCI specification requires a target to terminate the transfer prematurely if it will tie up the bus for long periods.

There are four possible cases :
1. If the time to complete the first data phase will be greater than 16 PCI CLKs, the target must immediately issue a **retry** to the master. Two exceptions are reading expansion ROM images and configuration startup. After two PCI CLKs have elapsed the master may reassert its request.
2. It will take more than eight PCI CLKs to complete a data phase other than the first, and it is not the final data phase (FRAME# is still asserted). The target must issue a **disconnect** or a **target abort.**
3. If a target can transfer the current data item within eight PCI CLKs but knows in advance that the next item will require more than eight cycles.
4. If the attempt to communicate with a target results in a collision on a busy resource, the target must issue a **retry** immediately.

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# line after it has asserted its FRAME# line for the first transaction. If the arbiter continues to assert its GNT# line after the first transaction, it means that the master still has ownership of the bus and it may immediately start a new transaction. However, the master still must insert an idle state between the two transactions. When it doesn't have to do that it's called Fast Back to Back Transactions. This can only be done if there is a guarantee that there will be no contention between masters and/or targets involved in the two transactions.

There are two scenarios :
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 designer of the bus master must make an informed decision as to whether its worth the additional logic it would take to implement it.

Assume a master whose nature of work requires it to make long burst transfers whenever it acquires bus ownership. Fast back-to-back transactions would not be of real benefit in such a case because one PCI CLK is small compared to the total burst transfer time. However a master that makes one PCI CLK transfers would benefit by the inclusion of such logic.

Scenario One : Master Guarantees Lack of Contention

In this scenario, the master must ensure that, when it performs two back to back transactions with no idle state in between them, there is no contention on any of the signals driven by the master or those driven by the target. An idle cycle is required whenever AD[31:0], C/BE#[3:0], FRAME#, PAR 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 before the next bus master begins to drive the bus. This prevents bus contention.

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 transaction, but must be initiated by the same master. The signals asserted by the target of the first transaction at the completion of the final data phase are TRDY#, and DEVSEL# (sometimes STOP#). Two clocks after the end of the data phase, the target may also drive PERR#. Since it is the rule in this scenario that the same target must be addressed in the second transaction, the same target again drives these signals.

Recognizing a new Transaction
It is a PCI 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 the rising edge of the PCI CLK followed on the next rising edge by the address phase in progress (FRAME# asserted and IRDY# deasserted)
- Final data phase in progress (FRAME# asserted and IRDY# deasserted) on a rising edge of the PCI CLK, followed by on the next rising edge by the address phase in progress (FRAME# asserted and IRDY# deasserted).

Scenario Two : Targets Guarantee Lack of Contention

In the second scenario, the entire community of the PCI targets that reside on the PCI bus and the bus master collectively guarantee lack of contention during fact back to back transactions. A constraint incurred when using the master guaranteed method 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#. The 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 all the targets on the PCI bus meet the above requirements then any bus master can perform fast back to back transactions with different targets in the first and second transactions. The first transaction must still be a write, and the second transaction must still be performed by the same master, to prevent collisions on master related signals.

The above statement implies that there is a method to determine if all the targets support this feature. During system configuration, the software polls each device's configuration status register and checks the state of it FAST BACK TO BACK CAPABLE bit. The designer of a device hardwires this bit to zero if the device does not support this feature. If all the devices indicate support for this capability, then the configuration software can set each master's FAST BACK TO BACK ENABLE bit in its configuration command register. 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 :

1. Normally a target recognizes a bus idle condition by sampling the FRAME# and IRDY# signals

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 the address decode. To support this feature, it must recognize the completion of the final data phase of one transaction by sampling FRAME# deasserted and IRDY#, TRDY# asserted. This would then immediately be followed by the start of the next transaction, as indicated by sampling FRAME# asserted and IRDY# deasserted during the next rising PCI CLK edge.

2. The target must ensure that there isn't contention on TRDY#, DEVSEL#, STOP#, and possibly PERR#. If the address 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. There are two circumstances when a target with a fast decoder does not have to insert this delay :

    - The current transaction was preceded by a bus idle state.
    - The currently addressed target was also addressed in the previous transaction.

Broken Master

The arbiter assumes that a master is broken if after asserting the GNT# line to that master, the bus has been idle for 16 PCI CLK cycles, and the master has not asserted its FRAME# signal to indicate the start of its transaction. The arbiter is permitted to ignore all future requests from a broken master, and may optionally report the failure to the operating system.

PCI BIOS And Cache Support

Purpose of PCI BIOS

The operating system, application programs and device drivers must not directly access the PCI configuration registers, interrupt routing logic, or the special cycle generation logic. The hardware methods utilized to implement these capabilities are platform dependent. Any software that directly accesses these capabilities is therefore by definition, platform specific. This leads to compatibility problems between different platforms, and sometimes even between different versions of the same platform.

Instead a request should be issued to the PCI BIOS. The BIOS is platform specific. It is implemented in firmware, and possibly in the operating systems hardware abstraction layer (HAL). The PCI BIOS supports the following services :

- Permits determination of configuration mechanism(s) supported by the PCI chipset
- Permits determination of the chipset's ability to generate special cycles and the mechanism(s) used to do so.
- Permits determination of the range of PCI buses present in the system.
- Searches for all instances of a specific PCI device or a device that falls within a class.
- Permits generation of special cycles.
- Allows the caller to get PCI interrupt routing options and then to assign one to the device.
- Permits read and write (if possible) of a device's configuration registers.

Operating System Environments Supported

General
Different operating systems have different operational characteristics such as usage of system memory, calling BIOS services, etc. In the systems based on the Intel x86 family, the operating system falls into one of the following category :

- Real Mode OS
- 286 Protected mode
- 386 Protected mode. This mode is divided into the segmented and flat models.

The PCI BIOS specification defines the following rules regarding the implementation of the PCI BIOS and the software that calls it :

- The PCI BIOS must support all of the OS environments.
- The BIOS must preserve all registers and flags with the exception of those used for return parameters and errors.
- Caller will be returned to with the state of the interrupt flag the same as it was on entry.
- Interrupts will not be enabled during the execution of the BIOS function call.
- The BIOS routines must be reentrant (i.e. they must be callable from within themselves).
- The OS must define a stack memory area at least 1 KB in size for the BIOS.
- The stack segment and the code segment defined by the operating system for the BIOS must have the same size (16 or 32-bit).
- Protected mode OS that call the INT 1Ah BIOS must set the CS register to F000h.
- The OS must ensure that the privilege defined for the BIOS permits interrupt handling and performance of I/O instructions.
- Implementers of the BIOS must assume that the CS for the BIOS defined by the OS is execute only

and that the DS is read only.

Real-Mode

Real-mode OSs, such as MS DOS, are written to be executed the 8088 processor. That processor is capable of addressing upto 1 MB of memory. Using four 16-bit segment registers (CS, DS, SS, and ES), the programmer defines four segments of memory, each with a fixed length of 64 KB.

MS DOS makes calls to the BIOS by loading a subset of the processor's register set with request parameters and then executing a software interrupt instruction that specifies entry 1Ah in the interrupt table containing the entry for the entry point to BIOS. Upon executing the INT 1Ah instruction, the processor pushes the address of instruction that follows the INT 1Ah onto the stack. Now the processor reads the pointer from 1Ah in the interrupt table and starts executing at the indicated address. This is the entry point for BIOS.

An alternative way to call the BIOS is to make the call directly to the BIOS entry point at the physical memory location 000FFE6Eh.

286 Protected Mode

The BIOS specification refers to this as the 16:16 mode because the 286 processor has 16-bit segment registers, and the programmer specifies the address of an object by specifying a 16-bit offset within a segment.

When operating in this mode, the 286 addresses memory differently. Rather than containing the upper four hex digits of the segments physical five hex digit start address in memory, the value in the segment register is referred to the segment selector. It points to an entry in the segment descriptor table that is built and maintained by the operating system. Each entry in this table has an eight byte value defining :

- the 24-bit physical start address of the segment in memory.
- the length of the segment
- the type of access (read only, read/write, execute only, etc.)

The method of calling the BIOS remains the same as in real-mode OSs.

386 Protected Mode

Segmented Mode
The 386 processor changed the maximum size of each segment from 64 KB to 4 GB in size. The 486 and the pentium processors have the same segment size as the 386. The 386 also gas a 32-bit register set. This mode is also called the 16:32 mode in the BIOS specification. Rather than containing the upper four hex digits of the segments physical five hex digit start address in memory, the value in the segment register is referred to the segment selector. It points to an entry in the segment descriptor table that is built and maintained by the operating system.

Each entry in this table has an eight byte value defining :

- the 32-bit physical start address of the segment in memory.
- the length of the segment
- the type of access (read only, read/write, execute only, etc.)

In the 32-bit OS environment the BIOS is not called in the conventional way. Rather, the calling program executes a FAR call to the BIOS entry point. This implies that the BIOS entry point is known.

Flat Mode
A much simpler memory model is to set all of the segment registers to point to segment descriptors that define each segment starting at a physical memory location 00000000h, and with a length of 4 GB. This is called the flat memory model. The BIOS specification refers to this as the 0:32 mode. As in the segmented mode, the BIOS is not called in the conventional way. Rather, the calling program executes a FAR call to the BIOS entry point. This implies that the BIOS entry point is known.

Calling the PCI BIOS

As stated earlier the 16-bit PCI BIOS is called by either executing an INT 1Ah call or by directly calling the PCI BIOS at memory location 00FFE6Eh. The 32-bit BIOS is called by performing a FAR call. In both cases the caller must first load the required request parameters into the processor's register set. On entry, the AH register must obtain the PCI function ID of B1h. The AL register must contain the PCI sub-function identification.

Example : PCI BIOS Present Call

- AH is set to B1h and AL is set to 01h.

On return the register set contains the following values :

- EDX contains the ASCII string "PCI", with DL = "P", DH = "C", and the byte above DL = "I". The upper byte of the EDX set is set to ASCII space (32).
- AH = 00h
- AL is a byte with the following bit information

Bits 2, 3, 6, and 7:		Reserved
Bit 0	:	Set to 1 if the PCI bridge uses config mechanism #1.
Bit 1	:	Set to 1 if the PCI bridge uses config mechanism #2.
Bit 4	:	Set to 1 if special cycle is supported via config mechanism #1.
Bit 5	:	Set to 1 if special cycle is supported via config mechanism #2.

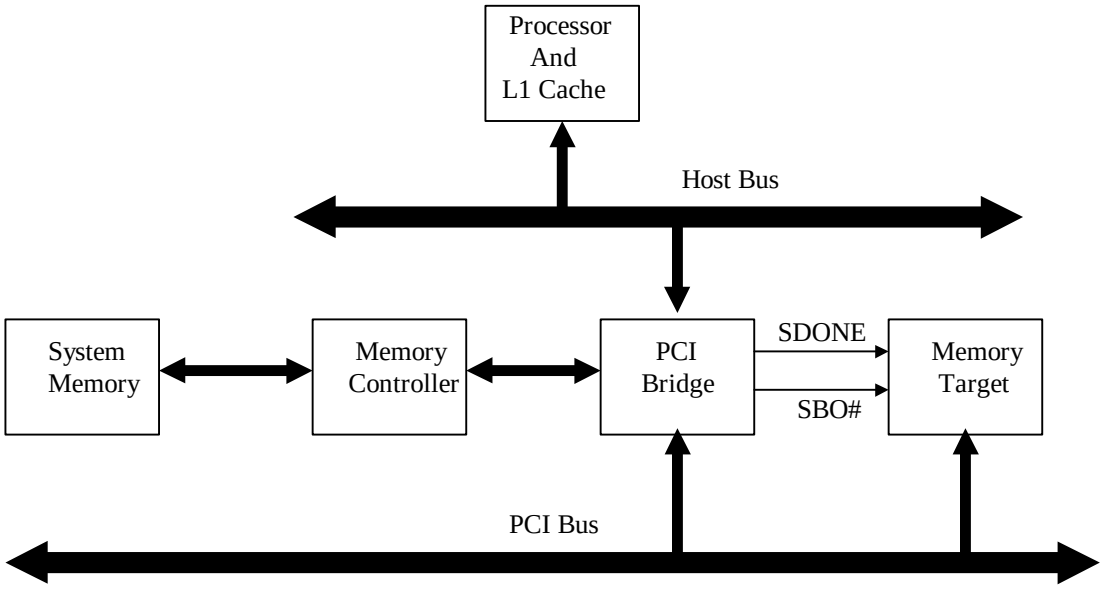
- BH = BIOS major version in BCD
- BL = BIOS minor version in BCD
- CL = number of the last PCI bus in the system
- Carry bit is cleared if BIOS present

PCI BIOS Function Request Codes

Function Request	AH	AL
PCI Function ID	B1h	00h
Test For PCI BIOS Present	B1h	01h
Find PCI Device	B1h	02h
Find PCI Class Code	B1h	03h
Generate a Special Cycle	B1h	06h
Read Configuration Byte	B1h	08h
Read Configuration Word	B1h	09h
Read Configuration DoubleWord	B1h	0Ah
Write Configuration Byte	B1h	0Bh
Write Configuration Word	B1h	0Ch
Write Configuration DoubleWord	B1h	0Dh
Get PCI Interrupt	B1h	0Eh
Set PCI Interrupt : Used to route a device's interrupt request line to the specified system interrupt request line	B1h	0Fh

The BIOS specification contains detailed description for each of these function calls.

PCI Cache Support



Most PCI systems available in the market today do not support cacheable memory on the PCI bus. It creates considerable complexity and the rewards may not be justified due to the degraded system performance.

Definition of Cacheable memory
A cacheable PCI memory target is a memory device residing on the PCI bus that the processor's L1 cache (and if present the L2 cache) can cache information from.

Reason for Supporting Cacheable Memory on the Bus
There are quite a few PCI design rules related to support for cacheable memory targets on the PCI bus. There are so many rules, in fact, that it makes one wonder why anyone would even consider caching from memory targets that reside on the PCI bus.
The specification includes support for cacheable memory on the PCI bus for system designs where the processor, main memory and all other major system devices reside directly on the PCI bus. The processor's front end logic has been redesigned to interface directly to the PCI bus. In this type of system, therefore, the cache and the memory that it caches from reside directly on the PCI bus.

Integrated Cache /Bridge
Many systems that incorporate the PCI bus also incorporate a level two (L2) cache associated with the host processor. These systems possess the following basic elements:

- The host processor possibly with an integrated level one cache.
- L2 cache controller and cache SRAM.
- System DRAM memory and controller.
- The host/PCI bridge.
- The PCI bus.
- PCI memory and I/O targets.
- PCI bus master.
- Expansion bus such as ISA, EISA or Micro Channel.

The L2 cache controller integrated with the bridge may use either a write through or a write back policy when handling memory writes initiated by the host processor.

PCI Cache Support Protocol

Basics

Whenever a memory access is initiated on the PCI bus by a bus master other than a host PCI bridge, the bridge must ensure the following:

- If it implements a write back controller, it must ensure that the current initiator isn't about to read data from or update data in a stale line in memory. It must also ensure that it invalidates its copy of a clean cache line in the event of a snoop hit on a write by a PCI bus master.
- If it implements a write through cache controller, it must ensure that it invalidates its copy of a cache line in the event of a snoop hit on the write by a PCI bus master.

To prevent a PCI bus master from reading data from or updating data in a stale memory line, the bridge must have some way of informing the PCI memory target of the snoop result. Under some circumstances the target memory must force the PCI initiator to retry the transaction later. The bridge implements two output signals, SDONE and SBO#, to inform the cacheable PCI memory target of the snoop result. SDONE is asserted by the bridge when it has completed snooping a PCI memory access. When SDONE is asserted, this indicates that the snoop result is available on the bridge's SBO# output signal. SBO# is asserted by the bridge in conjunction with SDONE to force the addressed PCI memory target to issue a retry to the PCI initiator in the event of a snoop hit on a modified cache line.

Case 1: Clean Snoop

Let us assume that the PCI cache is a write back cache. All cacheable PCI memory targets must monitor SDONE and SBO# during memory accesses. The currently addressed cacheable memory target must insert wait states in the data phase (by keeping TRDY# deasserted) until the snoop results are made available by the bridge. In this example the result of the snoop indicates either a miss or a hit on a clean line (SDONE asserted and SBO# deasserted) and the bridge signals this to the memory target. This is referred to as a clean snoop. The target then asserts TRDY#, accepts the data from the initiator and the initiator ends the transaction. The following is a description of the process:

1. The bus master starts the transaction when it samples its GNT# asserted and bus idle (FRAME# and IRDY# deasserted). It asserts FRAME#, drives the address onto the AD bus and the memory write command onto the C/BE bus.

2. The memory target samples FRAME#, the address and the command on the clock edge two and begins the address decode process.

3. The address space completes on the rising edge of clock two and the initiator ceases to drive the address and command, and begins to drive the write data on to the AD bus and the byte enables onto the C/BE bus. It asserts IRDY# to indicate the presence of the data to the target. It also deasserts FRAME#, indicating that it's ready to complete the last data phase. The currently addressed target asserts DEVSEL# during clock two to claim the transaction.

4. Because the memory target is cacheable memory, it must monitor SDONE and SBO# and is not permitted to accept the data from the initiator until the results of the snoop become available. The target keeps TRDY# deasserted until clock four because SDONE has not yet been asserted. On the rising edge of clock five SDONE is sample asserted, indicating that the results of the snoop are available on SBO#. SBO# is sampled deasserted, indicating a clean snoop.

5. The target asserts TRDY# during clock five. The target and initiator sample IRDY# and TRDY# asserted on the rising edge of clock six. The target accepts the data, completing the data phase. The initiator deasserts IRDY#, returning the bus to the idle state. The target deasserts TRDY# and DEVSEL#.

6. The bridge deasserts SDONE in clock 5, indicating that it is ready to receive another snoop address.

Case 2: Snoop Hit On Modified Line Followed By Write Back

Assume that a PCI bus master starts a memory read or write transaction targeting a cacheable memory target on the PCI bus. The address is snooped by the bridge, resulting in a snoop hit on a modified line. Unless something is done to prevent it, the bus master is about to read data from or write data into a stale line in memory. To prevent this, the bridge instructs the memory target to issue a retry to the initiator. The bridge then arbitrates for bus ownership and initiates a write transaction (also referred to as a write back) to deposit the fresh line into memory before the bus master re-attempts the memory access. In order to ensure that the bridge receives bus ownership quickly, the specification recommends that the bridge assert a point to point signal to the arbiter along with its REQ#. Upon seeing this, the arbiter would grant bus ownership to the bridge next. Having deposited the fresh line into memory, the bridge marks its copy of the line as clean if the write back was caused by a memory read attempt , or invalid if the write back was caused by a memory write attempt. Following is the sequence of events:

1. The bus master starts the transaction when it samples its GNT# asserted and bus idle (FRAME# and IRDY# deasserted). It asserts FRAME#, drives the address onto the AD bus and the memory write command onto the C/BE bus.

2. The memory target samples FRAME#, the address and the command on the clock edge two and begins the address decode process.

3. The address space completes on the rising edge of clock two and the initiator ceases to drive the address and command, and begins to drive the write data on to the AD bus and the byte enables onto the C/BE bus. It asserts IRDY# to indicate the presence of the data to the target. It also deasserts FRAME#, indicating that it's ready to complete the last data phase. The currently addressed target asserts DEVSEL# during clock two to claim the transaction.

4. Because the memory target is cacheable memory, it must monitor SDONE and SBO# and is not permitted to accept the data from the initiator until the results of the snoop become available. The target keeps TRDY# deasserted until clock four because SDONE has not yet been asserted. On the rising edge of clock five SDONE is sample asserted, indicating that the results of the snoop are available on SBO#. SBO# is sample asserted, indicating that it was a snoop hit on a modified line.

5. As a result, the target keeps TRDY# deasserted and asserts STOP# during clock four, instructing the initiator to stop the transaction on this data phase with no data transferred and to retry the transaction later.

6. When the initiator samples STOP# and DEVSEL# asserted and TRDY# deasserted, it deasserts IRDY# during clock five returning the bus to the idle state. The bridge must leave SDONE and SBO# asserted until it accomplishes the write back of the modified line to the cacheable memory target. As a result, any bus master that attempts to access any cacheable memory during this interim period is retried by the memory target because of the bridge's continued indication of a snoop hit on a modified line.

7. The bridge then arbitrates for access to the bus so that it may deposit the fresh line into memory i.e. to perform the write back. The arbiter grants the bus to the bridge and the bridge starts the write back. During the address phase, the bridge changes the setting on SBO# to indicate a clean snoop. This transition from the indication of a snoop hit on a modified line to a clean line instructs the memory to accept the entire line being deposited in memory by the bridge. If the memory target cannot immediately accept the entire line it must insert wait states (by keeping TRDY# deasserted) until it can accept the data.

8. During the next clock cycle, the bridge deasserts SDONE, returning the snoop signals to the standby state. The bridge performs the required number of data phases to deposit the entire line in memory and ends the transaction.

9. The bus master retries the transaction and this time a clean snoop results (because the state of the line is invalid). The memory target asserts TRDY# and accepts the data being written to it.

Treatment Of Memory Write And Invalidate Command

When a bus master initiates a memory write and invalidate transaction, it is guaranteeing that it is going to write the entire line into memory. When the bridge detects such a transaction it takes one of the two actions:

1. In the event of a snoop miss or a snoop hit on a clean or modified line, the bridge can signal a clean snoop back to the memory target, and the PCI memory target can assert TRDY# and start accepting

the data. If the result is a snoop hit on a modified line, the bridge can invalidate the line and indicate a clean snoop because the current bus master is guaranteeing that it will update the entire line in memory rendering the cache's copy stale. From a performance standpoint, this is the preferred approach.

2. In the event of a snoop hit on a modified line, the cache could signal the result as a hit on a modified line. As a result, the PCI memory target asserts STOP#, issuing a retry to the master. This process has been described in case 2 of this paper. This approach works but is wasteful.

Non-Cacheable Access Followed Immediately By Cacheable Access

Problem:

Upto this point in the discussion it has been assumed that the bridge and the cacheable memory target have the ability to latch on line address at a time. Continuing this assumption, consider the following scenario.

1. A bus master starts a single data phase memory transaction with a non-cacheable memory target. The bridge latches the address for snooping and the memory target latches it for decode. The bridge keeps SDONE deasserted while it performs the snoop, but, because the memory target is non-cacheable, the target is not connected to the snoop result signals and therefore permits the initiator to complete the transaction while the snoop is still in progress.

2. A bus master starts another memory access to a cacheable memory target. This memory targets controller was monitoring SDONE's activity for the previous transaction and has not yet seen it asserted. This indicates that the bridge is not yet done snooping the line address of the previous transaction. Because the bridge can only latch and snoop one address at a time in this scenario, the cacheable memory target must issue a retry to its initiator.

3. During the premature termination of this transaction the snoop results are presented (for the first transaction) but alas nobody cares.

4. When the memory access to the cacheable memory target has been stopped by the retry, a bus master initiates another single data phase access targeting a non-cacheable memory target. Once again this transaction completes without a waiting snoop completion.

5. The transaction to the cacheable memory that was stopped earlier is now re-initiated and this time too results in a retry because the snoop for the previous access is still in progress.

6. In theory an access to a cacheable memory target may never complete if the bus is experiencing interleaved single data phase accesses to non-cacheable and cacheable memory targets.

Solution: Snoop Address Buffer With Two Entries

The revision 2.0 specification made it a requirement that the bridge and cacheable memory targets have the ability to latch two addresses: the first for the snoop in progress and the second for the transaction just initiated.

1. In the scenario described above, the bridge latched the snoop address for the first access.

2. When the access completes, the access to the cacheable memory target starts. The bridge latches this address to be snooped after completion of the first snoop. The cacheable memory target is required to insert wait states into the data phase (by keeping TRDY# deasserted) until the result of the first snoop is presented.

3. It then monitors SDONE again on each clock edge until the result of the second snoop becomes available. When this snoop result is presented, it reacts accordingly.

4. If the target of the second transaction asserts TRDY# to indicate its readiness to complete the data transfer or STOP# to abort the transaction, before or coincident with the presentation of the result of the first snoop, this indicates that the target of the second access is also non-cacheable. In response, the bridge discards the second snoop address.

Gambling Cacheable Memory Targets

It is permissible within the constraints of a specification to build a cacheable memory target that does not wait for the results of a snoop before allowing a memory write access to complete. The goal would be performance enhancement during burst writes to cacheable memory targets. This implies that data could be written to a stale line in memory. In order to implement this feature the following steps would have to occur:

1. The cacheable memory target allows the memory write to complete without awaiting the result of the snoop. It must latch and remember the date written and the address it was written to so that it may "fix" the situation if necessary. After the memory write has completed the target must continue to monitor the snoop result signals until the results of the bridge's snoop becomes available.

2. All cache memory targets that reside on the PCI bus are required to "watch" memory transactions even when they are not the target of the current transaction. This being the case, other cacheable memory targets on the bus are aware that the next snoop result that shows up on the snoop result signals is for a previously completed transaction.

3. If no other access to a cacheable memory target occurs before the snoop result for the write is presented, the snoop result is ignored.

4. If, however, an access is started to any cacheable memory target on the bus, that cacheable target can assert DEVSEL# and claim the transaction but it must insert wait states until the snoop result for the previously concluded transaction is presented by the bridge.

5. The first time that SDONE is asserted during the transaction, the snoop result is latched by both the target of the current transaction and by the target that accepted the write data earlier (the gambler).

6. If the snoop result is clean, the gambler breaths a sign of relief and the target of the current transaction continues to keep TRDY# deasserted until the result of its snoop is presented. If the snoop result is clean, it asserts TRDY# and permits its master transfer data. If the result is a hit on a modified line, it issues a retry to its master so that the bridge can get the bus and freshen the stale line in memory.

7. If the result of the first snoop had been a hit on a modified line, the gambler gambled and lost. The target of the current transaction issues a retry to its master. It also realizes that the bridge cannot snoop its address because it must perform the write back of the modified line to memory. The gambler

also realizes that the bridge will get the bus and burst write the fresh line into memory. It must accept the line and then make changes that were made earlier by it on that line (during the gamble).

PCI Bridge With No Cache

When the bridge does not incorporate a cache and none of the PCI masters on the bus have caches, the SDONE and the SBO# input pins on any PCI memory targets should be tied high. This instructs the memory target to reply to any memory accesses within its assigned memory address range. If the PCI master has a cache, it would control the SDONE and SBO# lines to the memory targets. If a PCI memory target is designated as non-cacheable, it can tie SDONE and SBO# high or it can just ignore them.

PCI Bridge With Write Through Cache

A write through cache controller always propagates all host processor memory writes through to memory. This means that all lines contained in the cache are always the same as their respective lines in memory. By definition, all snoops result in clean snoops. The cache must be given the opportunity all memory writes performed by PCI masters so that it can invalidate a line when it determines that a master is changing data within that memory line. In addition, the bridge must initiate an invalidate cycle on the host bus so that the host processors internal L1 cache can snoop the write to determine whether to invalidate the L1 copy of the cache line. The bridge can ignore all memory reads because there is no danger of reading stale information.

When the bridge incorporates a write through cache, the bridge need only implement SDONE. SBO# may be tied high. This gives the bridge the ability to signal a STANDBY or CLEAN SNOOP to the currently addressed PCI memory target. Each time SDONE is asserted, a cacheable memory target can permit a memory transaction to complete. It is recommended that cache targets implement both SDONE and SBO# so that they can support operation in both write through and write back environments.

PCI Bridge Incorporates Write Back Cache

All PCI memory targets should connect to both SDONE and SBO#.

Burst Transfer Crossing Line Boundaries

When a PCI master initiates a burst transfer, the master drives the start address on to the AD bus during the address phase of the multiple data phase transaction. The cache latches this start address and snoops it. As the burst progresses the memory target must monitor the address of the currently addressed doubleword in order to determine if the burst transfer crosses over a cache line boundary. This is necessary because the master is not outputting the addresses onto the AD bus for each data item transferred and the cache must receive the next cache line address in order to snoop it.

To determine when the burst is crossing over the line address boundary, the memory agent must know the cache line size. This information would either be hard wired or programmed (preferred) into the PCI memory targets cache line size configuration register. If the transfer crosses the boundary, the memory target must issue either a disconnect or retry to the master. This forces the master to end the current transfer, re-arbitrate for the bus and then re-initiate the transfer, specifying the start address of the next line as the start address of the next transaction. This allows cache to then latch and snoop the new address.

Snoop Results

SDONE	SBO#	Description
0	X	Standby: Snoop results pending. Informs the addressed memory target that the bridge is snooping the transaction. Upon decoding its address, the memory target claims the transaction by asserting DEVSEL#, but it inserts wait states until the bridge indicates either a clean snoop, or a hit on a modified line.
1	1	Clean snoop: OK to proceed with data transfer. The bridge has determined that no intervention if necessary (i.e. the line has not been modified) and the transaction may proceed. In response, the memory target accepts the data from the (on a memory write) or supplies the data to the bus master (on a memory read).
1	0	Hit on a Modified Line: Issue a retry to the master. The addressed PCI memory target issues a retry to the master, causing it to abort the transaction with no data transferred. The cache writes the modified line into the PCI memory, invalidates the line (on snoop of write) or marks the line clean (on snoop of read), and then permits the master to retry the memory access to the line in memory.