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.

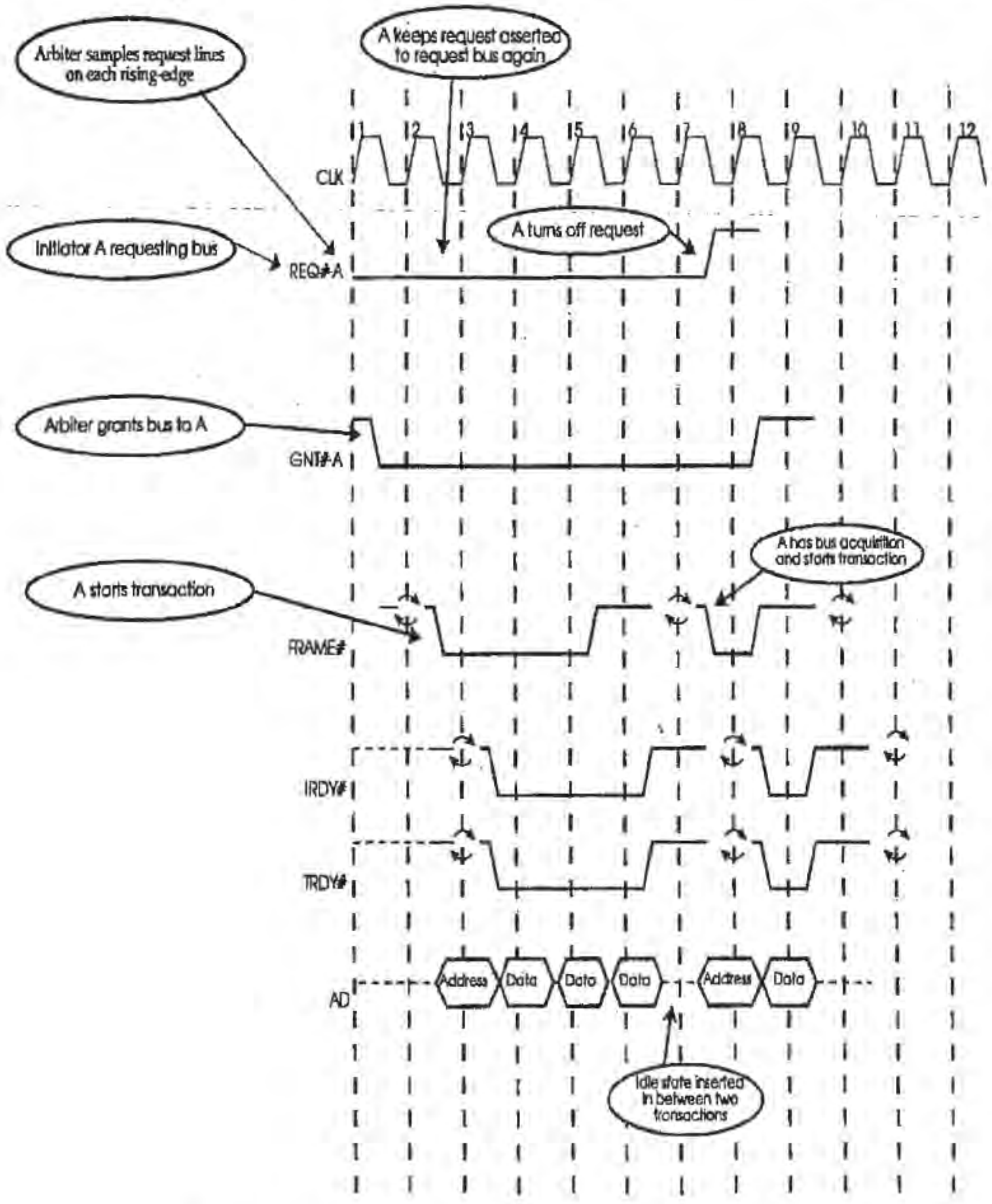
Arbiter samples request lines on each rising-edge
A keeps request asserted to request bus again
CLK
1
2
3
4
5
6
7
8
9
10
11
12
Initiator A requesting bus
REQ#A
A turns off request
Arbiter grants bus to A
GNT#A
A has bus acquisition and starts transaction
A starts transaction
FRAME#
IRDY#
TRDY#
AD
Address
Data
Data
Data
Address
Data
Idle state inserted in between two transactions

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.

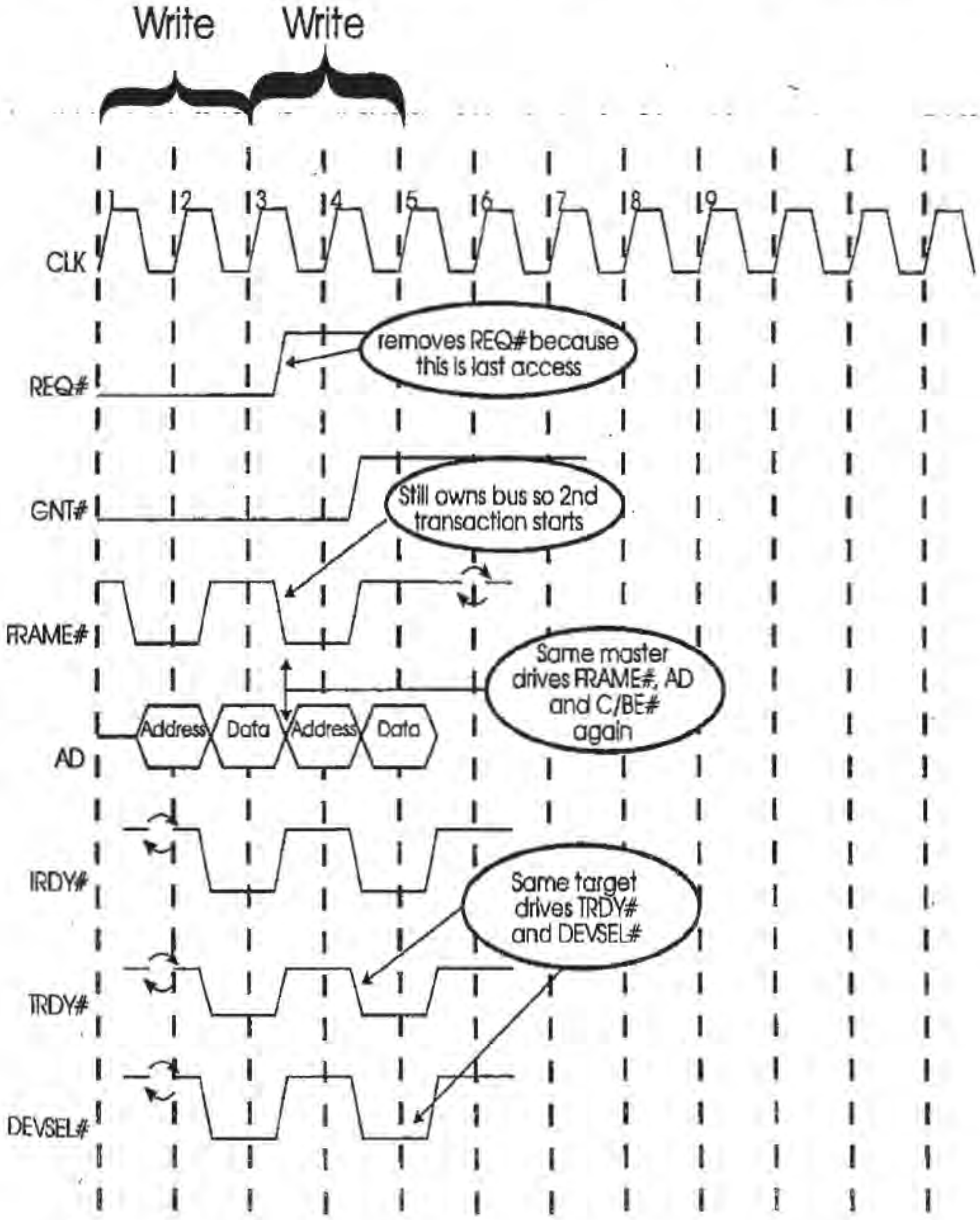
Write
Write
1
2
3
4
5
6
7
8
9
CLK
REQ#
removes REQ# because this is last access
GNT#
Still owns bus so 2nd transaction starts
FRAME#
AD
Address
Data
Address
Data
Same master drives FRAME#, AD and C/BE# again
IRDY#
TRDY#
Same target drives TRDY# and DEVSEL#
DEVSEL#

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, PReP-compliant 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.

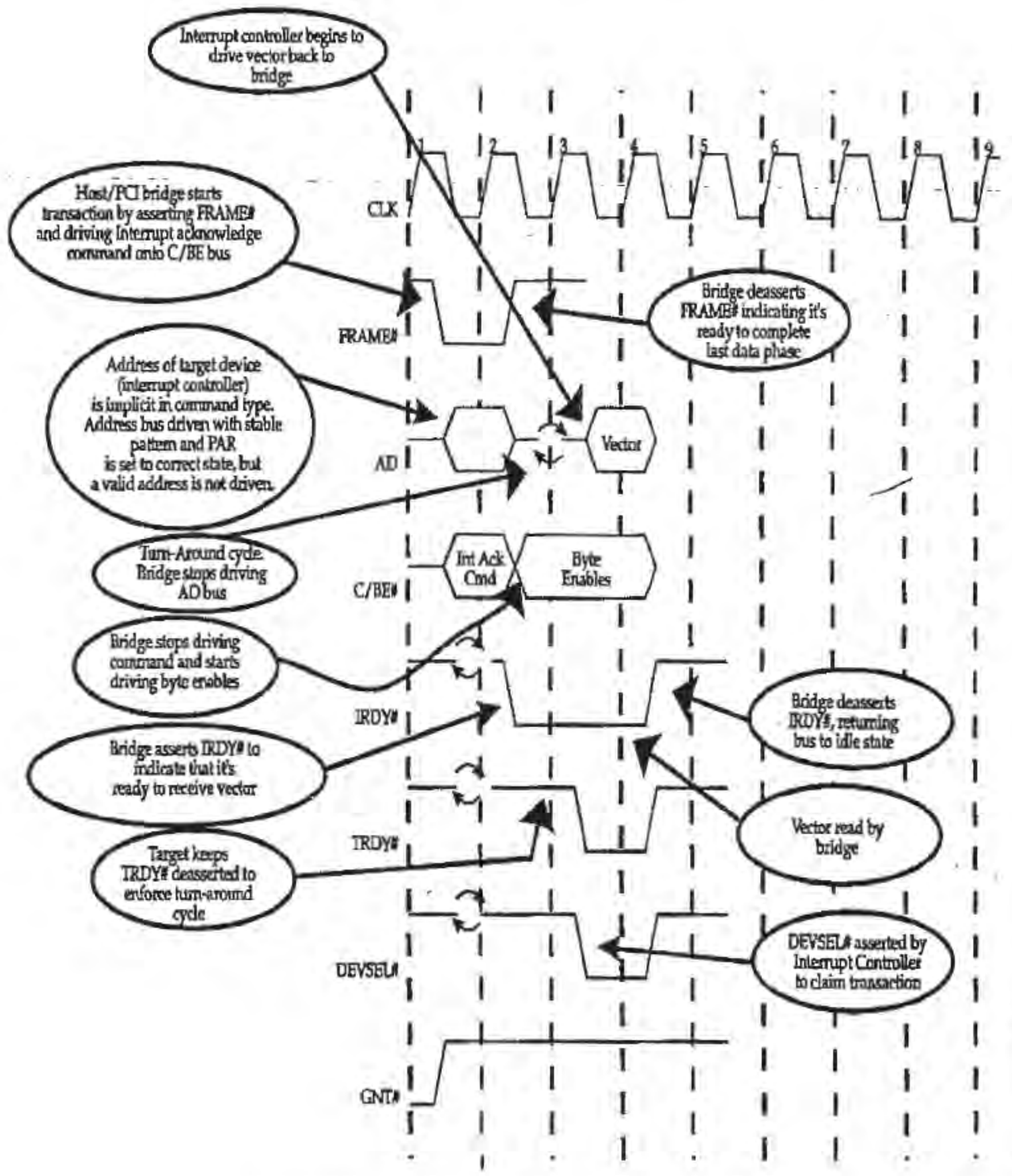
Interrupt controller begins to drive vector back to bridge
Host/PCI bridge starts transaction by asserting FRAME# and driving Interrupt acknowledge command onto C/BE bus
Bridge deasserts FRAME# indicating it's ready to complete last data phase
Address of target device (interrupt controller) is implicit in command type. Address bus driven with stable pattern and PAR is set to correct state, but a valid address is not driven.
Turn-Around cycle. Bridge stops driving AD bus
Bridge stops driving command and starts driving byte enables
Bridge asserts IRDY# to indicate that it's ready to receive vector
Target keeps TRDY# deasserted to enforce turn-around cycle
Bridge deasserts IRDY#, returning bus to idle state
Vector read by bridge
DEVSEL# asserted by Interrupt Controller to claim transaction
CLK
FRAME#
AD
C/BE#
IRDY#
TRDY#
DEVSEL#
GNT#
Vector
Int Ack Cmd
Byte Enables
1
2
3
4
5
6
7
8
9

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 transaction. 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.

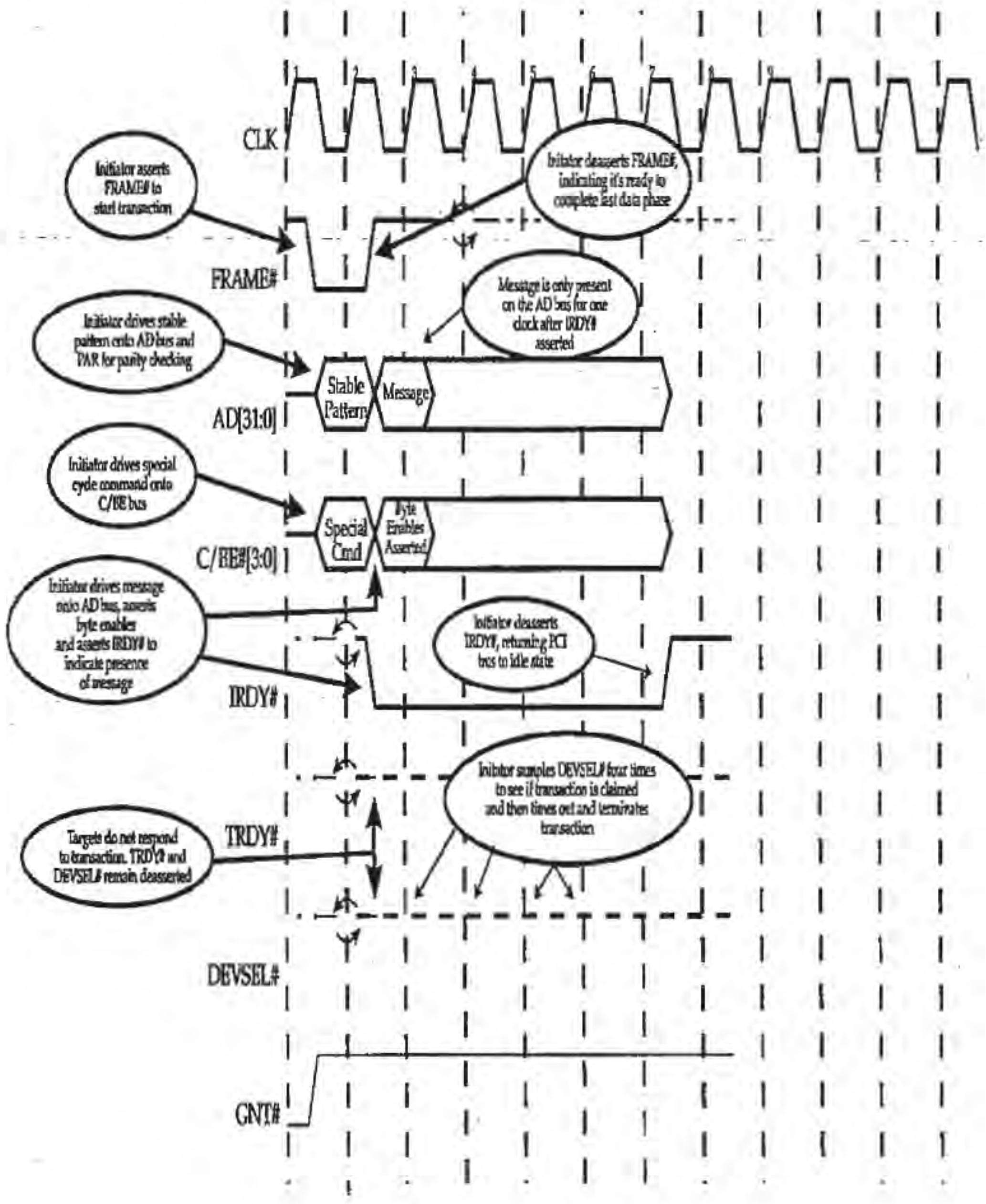
1
2
3
4
5
6
7
8
9
CLK
Initiator asserts FRAME# to start transaction
Initiator deasserts FRAME#, indicating it's ready to complete last data phase
FRAME#
Message is only present on the AD bus for one clock after IRDY# asserted
Initiator drives stable pattern onto AD bus and PAR for parity checking
Stable Pattern
Message
AD[31:0]
Initiator drives special cycle command onto C/BE bus
Special Cmd
Byte Enables Asserted
C/BE#[3:0]
Initiator drives message onto AD bus, asserts byte enables and asserts IRDY# to indicate presence of message
Initiator deasserts IRDY#, returning PCI bus to idle state
IRDY#
Initiator samples DEVSEL# four times to see if transaction is claimed and then times out and terminates transaction
Targets do not respond to transaction. TRDY# and DEVSEL# remain deasserted
TRDY#
DEVSEL#
GNT#

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#.

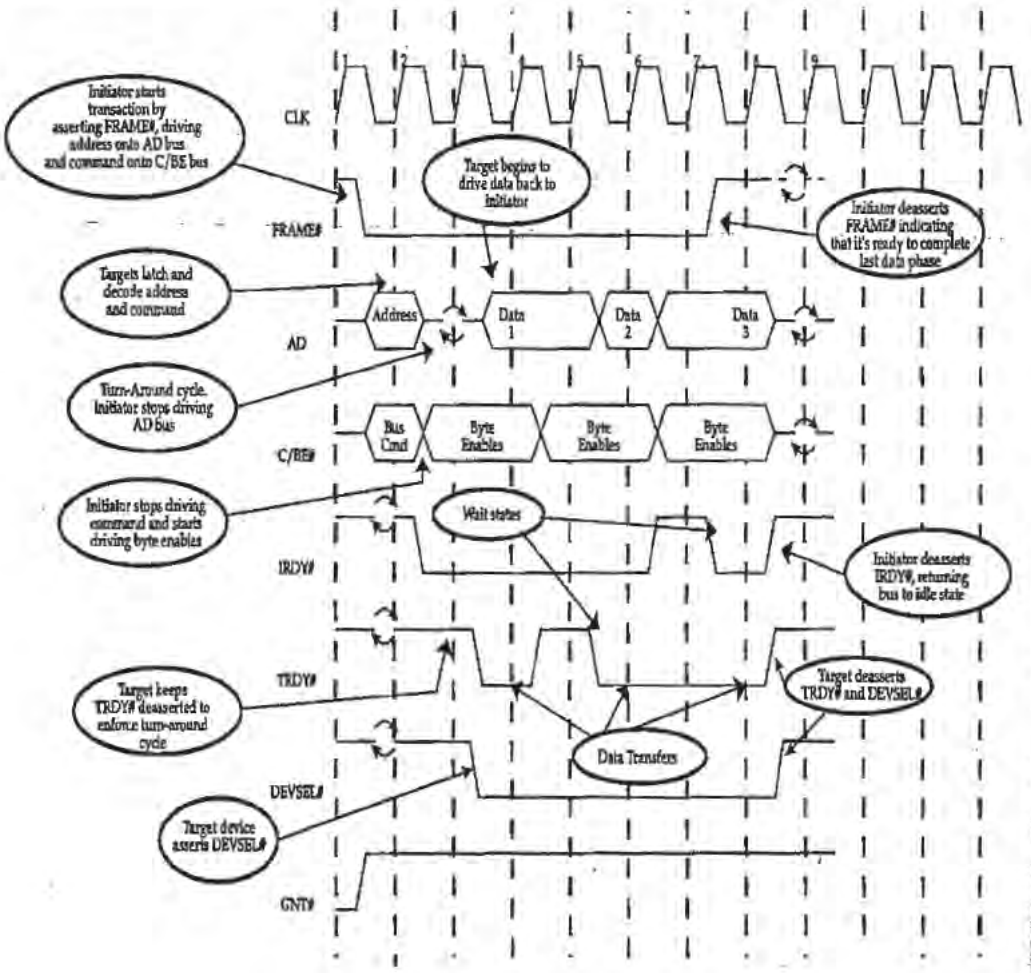
CLK
FRAME#
AD
C/BE#
IRDY#
TRDY#
DEVSEL#
GNT#
Initiator starts transaction by asserting FRAME#, driving address onto AD bus and command onto C/BE bus
Target begins to drive data back to initiator
Initiator deasserts FRAME# indicating that it's ready to complete last data phase
Targets latch and decode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

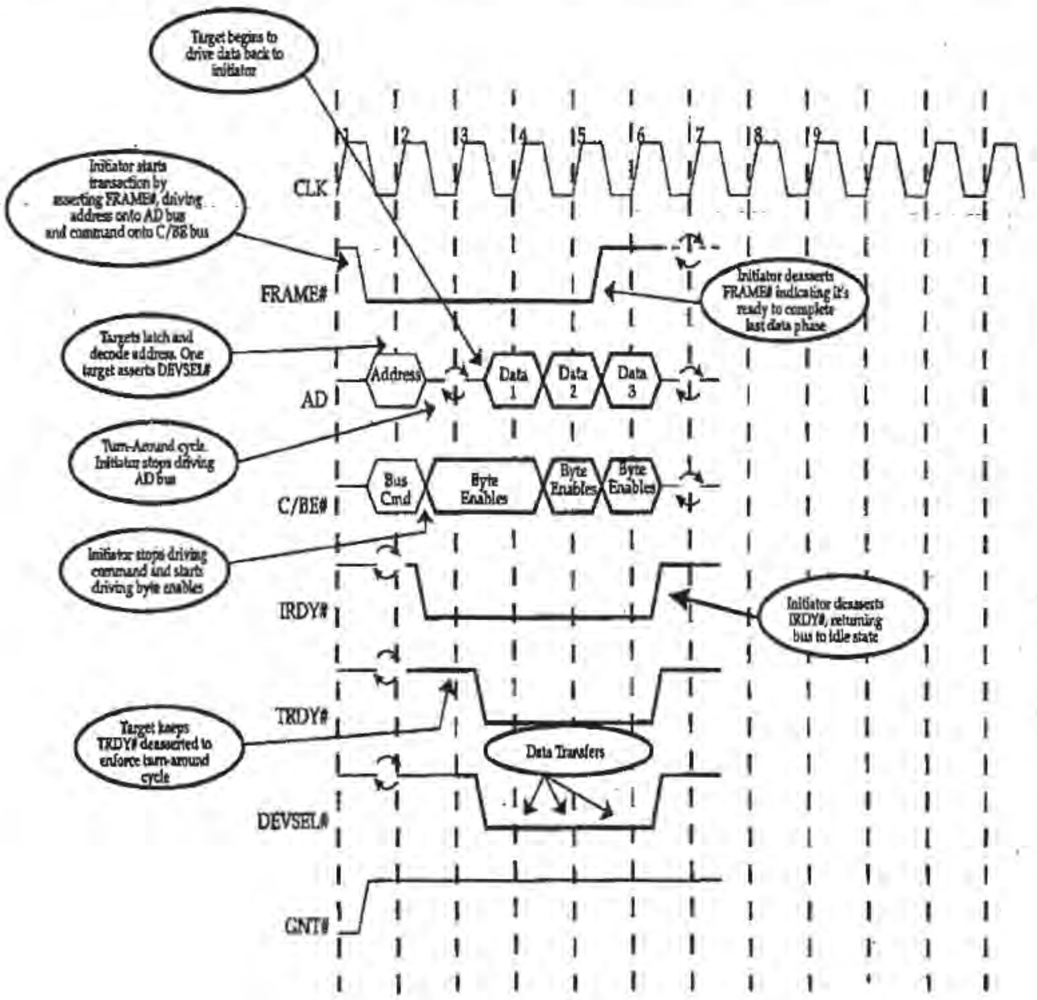
Target begins to drive data back to initiator
1
2
3
4
5
6
7
8
9
CLK
Initiator starts transaction by asserting FRAME#, driving address onto AD bus and command onto C/BE# bus
FRAME#
Initiator deasserts FRAME# indicating it's ready to complete last data phase
Targets latch and decode address. One target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

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#.

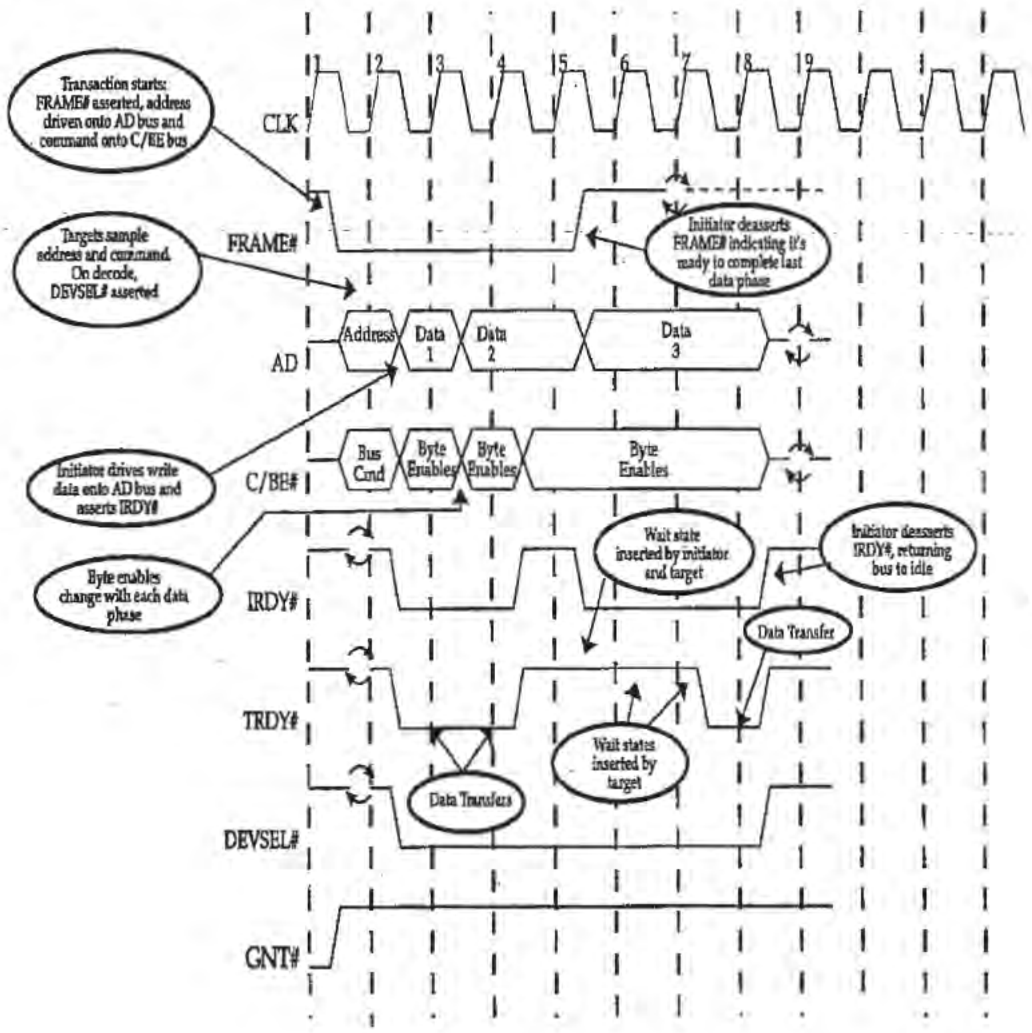
Transaction starts: FRAME# asserted, address driven onto AD bus and command onto C/BE bus
Targets sample address and command. On decode, DEVSEL# asserted
Initiator drives write data onto AD bus and asserts IRDY#
Byte enables change with each data phase
Initiator deasserts FRAME# indicating it's ready to complete last data phase
Wait state inserted by initiator and target
Initiator deasserts IRDY#, returning bus to idle
Data Transfer
Wait states inserted by target
Data Transfers
CLK
FRAME#
AD
C/BE#
IRDY#
TRDY#
DEVSEL#
GNT#
Address
Data 1
Data 2
Data 3
Bus Cmd
Byte Enables
1
2
3
4
5
6
7
8
9

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.

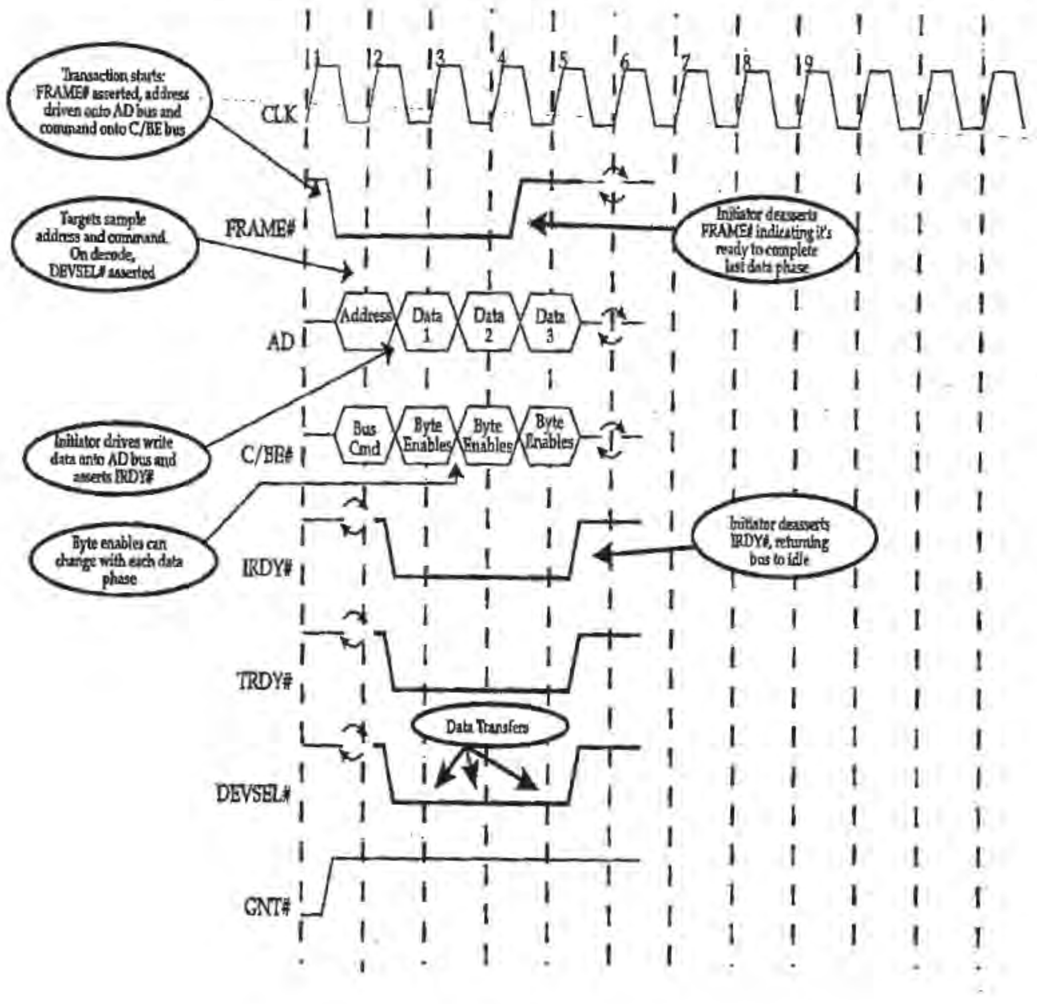
Transaction starts: FRAME# asserted, address driven onto AD bus and command onto C/BE bus
Targets sample address and command. On decode, DEVSEL# asserted
Initiator drives write data onto AD bus and asserts IRDY#
Byte enables can change with each data phase
Initiator deasserts FRAME# indicating it's ready to complete last data phase
Initiator deasserts IRDY#, returning bus to idle
Data Transfers
CLK
FRAME#
AD
C/BE#
IRDY#
TRDY#
DEVSEL#
GNT#
Address
Data 1
Data 2
Data 3
Bus Cmd
Byte Enables
Byte Enables
Byte Enables
1
2
3
4
5
6
7
8
9

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.