CHAPTER 3

# Bus Protocol

The essence of any bus is the set of rules by which data moves between devices. This set of rules is called a "protocol." This chapter describes the basic protocol that controls the transfer of data between devices on a PCI bus.

## PCI Bus Commands

The PCI bus command for a transaction is conveyed on the C/BE# lines during the address phase. Note that when C/BE# is carrying command data it is assertion high (high level = logic 1), whereas when it carries byte enable data it is assertion low.

The PCI bus defines three distinct address spaces with corresponding read and write commands as shown in Table 3-1. The principal distinction between memory and I/O spaces is that memory is generally considered to be "prefetchable," meaning that reads from memory space have no "side effects." In contrast, a read from I/O space may have the side effect of, for example, resetting the Data Ready bit in a status register. If such a register were prefetched but not actually read, data could be lost.

Configuration address space is used only at bootup time to configure the community of PCI cards in a system.

There are some additional read/write commands that apply to prefetchable memory space only. The purpose of *Memory Read Line* is to tell the target that the initiator intends to read most of, if not all of the full current cache line. The target may gain some performance advantage by knowing that it is expected to supply up to an entire cache line. When an initiator issues the *Memory Read Multiple* command, it is saying that it intends to read more than one cache line before disconnecting. This tells the target that it is worthwhile to prefetch the next cache line as soon as possible.

**Table 3-1: PCI command types.**

| C/BE#3 | C/BE#2 | C/BE#1 | C/BE#0 | 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 |

*Memory Write and Invalidate* is semantically identical to Memory Write with the addition that the initiator commits to write a full cache line in a single PCI transaction. This is useful when a transaction hits a "dirty" line in a writeback cache. Because the current initiator is updating the entire line, the cache can simply invalidate the line without bothering to write it back.

The *Interrupt Acknowledge* command is a read implicitly addressed to the system interrupt controller. The contents of the AD bus during the address phase are irrelevant, and the C/BE# indicate the size of the returned vector during the corresponding data phase.

The *Special Cycle* command provides a message broadcast mechanism as an alternative to separate physical signals for sideband communication. The *Dual Address Cycle (DAC)* command is a way to transfer a 64-bit address on a 32-bit backplane.

## Basic Read/Write Transactions

Figure 3-1 shows the timing of a typical read transaction—one that transfers data from the Target to the Initiator. Let's follow it cycle-by-cycle.

### *Clock*

1 The bus is idle and most signals are tri-stated. The master for the upcoming transaction has received its GNT# and detected that the bus is idle so it drives FRAME# high initially.

2 Address Phase: The initiator drives FRAME# low and places a target address on the AD bus and a bus command on the C/BE# bus. All targets latch the address and command on the rising edge of clock 2.

3 The initiator asserts the appropriate lines of the C/BE# (byte enable) bus and also asserts IRDY# to indicate that it is ready to accept read data from the target. The target that recognizes its address on the AD bus asserts DEVSEL# to acknowledge its selection.

This is also a *turnaround cycle*: In a read transaction, the initiator drives the AD lines during the address phase and the target drives it during the data phases. Whenever more than one device can drive a PCI bus line, the specification requires a one-clock-cycle turnaround, during which neither device is driving the line, to avoid possible contention that could result in noise spikes and unnecessary power consumption. Turnaround cycles are identified in the timing diagrams by the two circular arrows chasing each other.

4 The target places data on the AD bus and asserts TRDY#. The initiator latches the data on the rising edge of clock 4. Data transfer takes place on any clock cycle during which both IRDY# and TRDY# are asserted.

5 The target deasserts TRDY# indicating that the next data element is not ready to transfer. Nevertheless, the target is required to continue driving the AD bus to prevent it from floating. This is a *wait cycle*.

6 The target has placed the next data item on the AD bus and asserted TRDY#. Both IRDY# and TRDY# are asserted so the initiator latches the data bus.

7 The initiator has deasserted IRDY# indicating that it is not ready for the next data element. This is another wait cycle.

8 The initiator has reasserted IRDY# and deasserted FRAME# to indicate that this is the last data transfer. In response the target deasserts AD, TRDY# and DEVSEL#. The initiator deasserts C/BE# and IRDY#. This is a *master-initiated termination*. The target may also terminate a transaction as we'll see later.

Figure 3-2 shows the details of a typical write transaction where data moves from the initiator to the target. The primary difference between the write transaction and the read transaction detailed in Figure 3-1, is that write does not require a turnaround cycle between the address and first data phase because the same agent is driving the AD bus for both phases. Thus, the initiator can drive data onto the AD bus during clock 3.

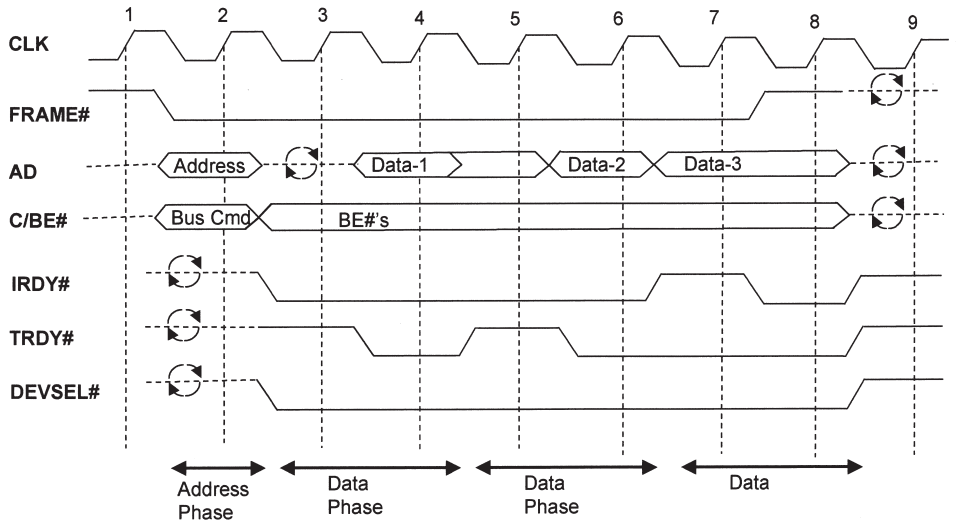


**Figure 3-1: Timing diagram for a typical read transaction.**

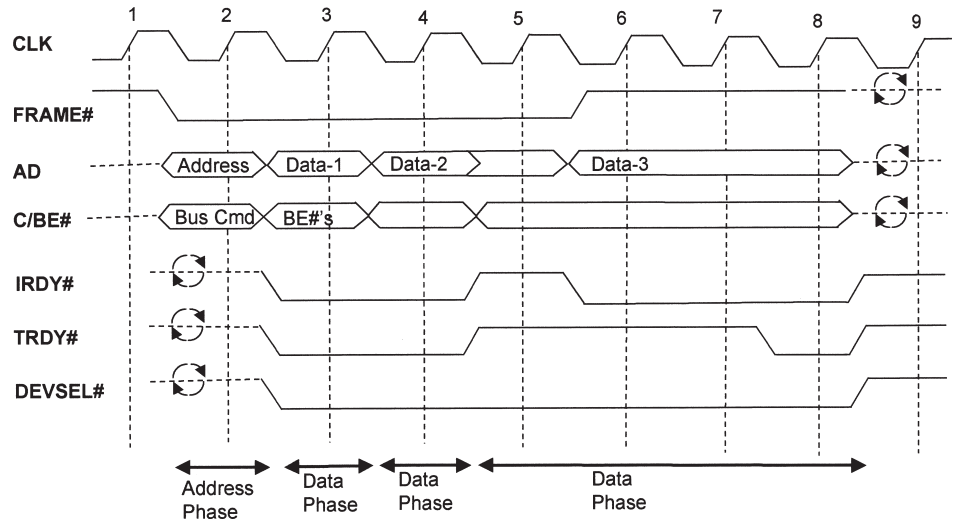


**Figure 3-2: Timing diagram for a typical write transaction.**

### Byte Enable Usage

During the data phases of a transaction, the C/BE# signals indicate which *byte lanes* convey meaningful data. The initiator may change byte enables between data phases but they must be valid on the clock that starts each data phase and remain valid for the entire data phase. The initiator is free to use any contiguous or noncontiguous combination of byte enables, including none, i.e., no byte enables asserted.

Independent of the byte enables, the agent driving the AD bus is required to drive all 32 lines to stable values. This is to assure valid parity generation and checking and to prevent the AD lines from floating.

### Use of AD[1:0] During Address Phase

Since C/BE# conveys information about which of four bytes are to be transferred during each data phase, AD[1:0] is not needed during the address phase of a memory transaction and can be used for something else. Specifically, AD[1:0] indicates how the target should advance the address during a multidata phase burst as shown in Table 3-2. Linear addressing is the normal case wherein the target advances the address by 4 (32-bit transfer) or 8 (64-bit transfer) for each data phase.

*Cache line Wrap* mode only applies if a burst begins in the middle of a cache line. When the end of the cache line is reached, the address *wraps around* to the beginning of the cache line until the entire line has been transferred. If the burst continues beyond this point, the next transfer is to/from the same location in the next cache line where the transfer began.

**Table 3-2: AD[1:0] for memory transfers.**

| AD1 | AD0 | Address Sequence |
|---|---|---|
| 0 | 0 | *Linear (sequential) addressing*. Target increments address by 4 after each data phase. |
| 0 | 1 | *Reserved*. Target disconnects after first data phase. |
| 1 | 0 | *Cache line wrap*. New in Rev. 2.1. If initial address was not beginning of cache line, wrap around until cache line filled. |
| 1 | 1 | *Reserved*. Target disconnects after first data phase. |

Here's an example: Consider a cache line size of 16 bytes (4 DWORDs) and a transfer that begins at location 8. The first transfer is to location 8, the second to location C hex which is the end of the cache line. The third data phase is to address 0 and the fourth to address 4. If the burst continues, the next data phase will be to location 18 hex.

Targets are not required to support cache line wrap. If a target does not support this feature it should terminate the transaction after the first data phase.

Addresses for transfers to I/O space are fully qualified to the byte level. That is, AD[1:0] convey valid address information inferring the least significant valid byte. This in turn implies which C/BE# signals are valid. Thus, for example if AD[1:0] = 00, at a minimum C/BE#[0] must be 0 to transfer the low-order byte but up to four bytes could be transferred. Conversely if AD[1:0] = 11, only the high-order byte can be transferred so C/BE#[3] is 0 and C/BE#[2:0] must be 1. See Table 3-3.

**Table 3-3: AD1:0 for I/O transfers**

*AD1:0 implies which BE# lines are valid*

| AD1 | AD0 | C/BE#3 | C/BE#2 | C/BE#1 | C/BE#0 |
|---|---|---|---|---|---|
| 0 | 0 | X | X | X | 0 |
| 0 | 1 | X | X | 0 | 1 |
| 1 | 0 | X | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 |

0: line must be asserted
1: line must *not* be asserted
X: line *may* be asserted

### DEVSEL# Timing

The selected target is required to "claim" the transaction by asserting DEVSEL# within three clock cycles of the assertion of FRAME# by the current initiator as shown in Figure 3-3. This leads to three categories of target devices based on their response time to FRAME#. A *fast* target responds in one clock cycle, a *medium* target in two cycles and a *slow* target in three cycles. A target's DEVSEL# timing is encoded in the Configuration Space Status Register. The target must assert DEVSEL# before it can assert TRDY# (or AD on a read transaction).

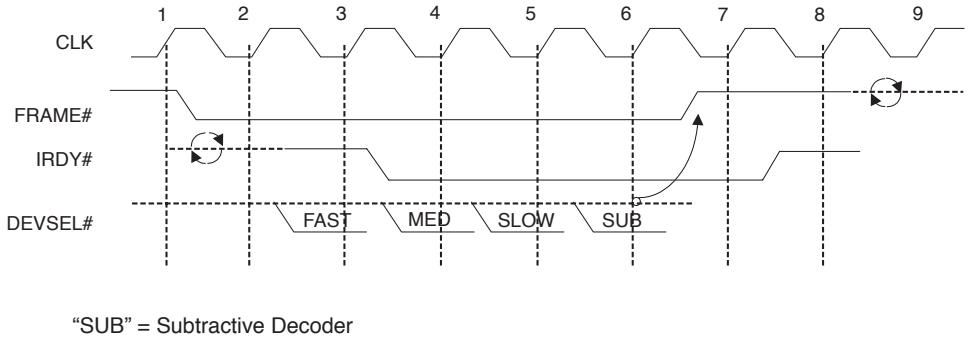


**Figure 3-3: DEVSEL# timing.**

If no agent claims the transaction within three clocks, a *subtractive-decode* agent may claim it on the fourth clock. A PCI bus segment can have at most one subtractive decode agent, which is typically a bridge to another PCI segment or an expansion bus such as ISA, EISA, and so forth. The strategy is that if no agent claims the transaction on this bus segment, then it's probably intended for some agent on the expansion bus segment on the other side of the bridge. So, the bridge claims the transaction by asserting DEVSEL# and forwards it to the expansion bus.

The problem with subtractive decoding is that every transaction on the expansion bus incurs an additional latency of four clock cycles. As an alternative, the bridge could—and in most cases does—implement *positive decoding* whereby it is programmed at configuration time with one or more address ranges to which it will respond. Then it can claim transactions like any other target.

Finally, if all targets on a segment are either fast or medium, as indicated by their status registers, a subtractive decoding bridge could be programmed to tighten up its DEVSEL# response by one or two clock cycles.

If DEVSEL# is not asserted after 4 clocks following FRAME# assertion, the initiator terminates the transaction with a Master-Abort. This means the initiator tried to access an address that doesn't exist in the system.

### Address/Data Stepping

Turning on 32 drivers simultaneously can lead to large current spikes on the power supply and crosstalk on the bus. One solution is to stagger the driver turn on as shown in Figure 3-4. In this example, the 32-bit AD bus is divided into four groups that are turned on in successive clock cycles.

For address stepping, the initiator asserts FRAME# only when all four driver groups are on. Data can likewise be stepped. The example here is a write cycle so the initiator asserts IRDY# only when all four driver groups have switched to the current data item.

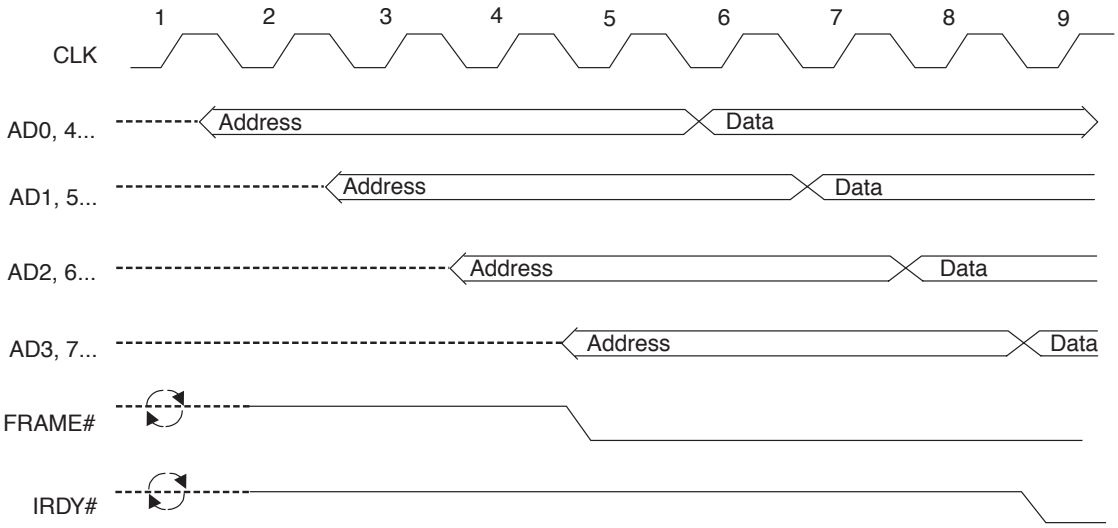


**Figure 3-4: Address/Data stepping.**

Although Figure 3-4 shows stepping synchronized to the PCI clock, this is not required.

Address/Data stepping only applies to *qualified* signals—those whose value is only considered valid when one or more control signals are asserted. The qualified signals consist of AD, PAR and PAR64, and IDSEL. AD is qualified by FRAME# during the address phase, and IRDY# or TRDY# during a data phase. PAR and PAR64 are valid one clock cycle after the corresponding address or data phase. IDSEL is qualified by FRAME# and a configuration command.

There are a couple of problems with address/data stepping. First, it reduces performance by using additional clock cycles. Second, during a stepped address phase, another higher priority master may request the bus causing the arbiter to remove GNT# from the agent in the process of stepping. Since the stepping agent hasn't asserted FRAME#, the bus is technically idle. In this case the stepping agent must tri-state its AD drivers and recontend for the bus.

A device indicates its ability to do address/data stepping through a bit in its configuration command register.

### IRDY#/TRDY# Latency

The specification characterizes PCI as a "low latency, high throughput I/O bus." In keeping with that objective, the specification imposes limits on the number of wait states initiators and targets can add to a transaction.

Specifically, an initiator must assert IRDY# within 8 clock cycles of the assertion of FRAME# on the first data phase and within 8 clock cycles of the deassertion of IRDY# on subsequent data phases. As a general rule, initiator latency should be fairly short because the agent shouldn't request the bus until it is either ready to supply data for a write transaction or accept data for a read transaction.

Similarly, a target is required to assert TRDY# within 16 clocks of the assertion of FRAME# for the first data phase and within 8 clocks of the completion of the previous data phase. This acknowledges the case where a target may need additional time to get a buffer ready when it is first selected, but should be able to deliver subsequent data items with relatively short latency.

### Fast Back-to-back Transactions

Normally, an idle turnaround cycle must be inserted between transactions to avoid contention on the bus. However, there are some circumstances under which the turnaround cycle can be eliminated, thus improving overall performance. The primary requirement is that there be no contention on any PCI bus line.

Depending on circumstances, either the initiator or the target can guarantee lack of contention.

If a master keeps its REQ# line asserted after it asserts FRAME#, it is asking to execute another transaction. As long as its GNT# remains asserted (i.e., no other agents are requesting the bus), the next transaction will be executed by the same master. There is no contention on any lines driven by the master as long as the first transaction was a write.

Furthermore, the second transaction must address the same target so that the same agent is driving DEVSEL# and TRDY#. This implies that the initiator has knowledge of target address boundaries in order to know that it is addressing the same one.

Figure 3-5 illustrates fast back-to-back timing for a master. The master keeps REQ# asserted through the first transaction to request a second transaction. In clock 3 the master drives write data followed immediately in clock 4 by the address phase of the next transaction. This example shows the second transaction as being a write. If it were a read, a turnaround cycle would need to be inserted after the second address phase.

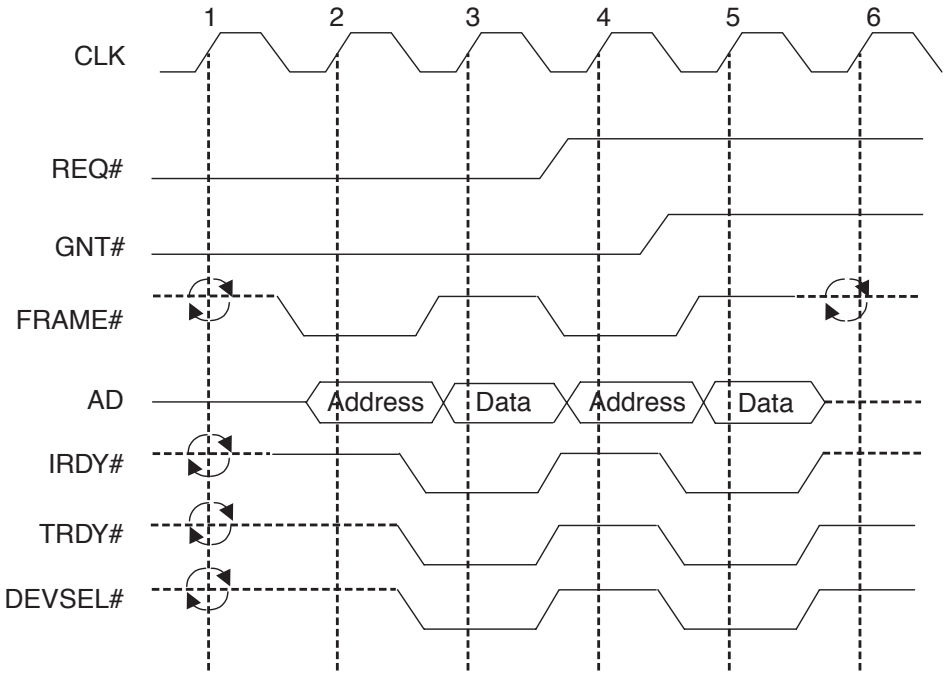


**Figure 3-5: Fast back-to-back timing.**

The entire community of targets on a bus segment can guarantee a lack of bus contention if:

- All targets have medium or slow address decoders AND
- All targets can detect the start of a new transaction without the transition through the idle state

Because fast back-to-back timing includes no idle cycle (both FRAME# and IRDY# deasserted), targets must detect a new transaction as the falling edge of FRAME#. Such targets have the FAST BACK-TO-BACK CAPABLE bit set in their configuration status registers. If all targets are fast back-to-back capable and all targets are either medium or slow, then the target of the second half of a fast back-to-back transaction can be different because the delay in DEVSEL# guarantees a lack of contention.

## Transaction Termination—Initiator

A transaction is "normally" terminated by the initiator when it has read or written as much data as it needs to. The initiator terminates a normal transaction by deasserting FRAME# during the last data phase. There are two circumstances under which an initiator may be forced to terminate a transaction prematurely

### *Initiator Preempted*

If another agent requests use of the bus and the current initiator's latency timer expires, it must terminate the current transaction and complete it later.

### *Master Abort*

If a master initiates a transaction and does not sense DEVSEL# asserted within four clocks, this means that no target claimed the transaction. This type of termination is called a *master abort* and usually represents a serious error condition.

## Transaction Termination—Target

There are also several reasons why the target may need to terminate a transaction prematurely. For example, its internal buffers may be full and it is momentarily unable to accept more data. It may be unable to meet the maximum latency requirements of 16 clocks for first word latency or 8 clocks for subsequent word latency. Or it may simply be busy doing something else.

The target uses the STOP# signal together with other bus control signals to indicate its need to terminate a transaction. There are three types of target-initiated terminations:

*Retry:* Termination occurs before any data is transferred. The target is either busy or unable to meet the initial latency requirements and is simply asking the initiator to try this transaction again later. The target signals retry by asserting STOP# and not asserting TRDY# on the initial data phase.

*Disconnect:* Once one or more data phases are completed, the target may terminate the transaction because it is unable to meet the subsequent latency requirement of eight clocks. This may occur because a burst crosses a resource boundary or a resource conflict occurs. The target signals a disconnect by asserting STOP# with TRDY# either asserted or not.

*Target-Abort:* This indicates that the target has detected a fatal error condition and will never by able to complete the requested transaction. Data may have been

transferred before the Target-Abort is signaled. The target signals Target-Abort by asserting STOP# at the same time as deasserting DEVSEL#.

### Retry—The Delayed Transaction

Figure 3-6 shows the case of a target retry. The target claims the transaction by asserting DEVSEL# but, at the same time, signals that it is not prepared to participate in the transaction at this time by asserting STOP# instead of TRDY#. The initiator deasserts FRAME# to terminate the transaction with no data transferred. In the case of a retry, the initiator is obligated to retry the exact same transaction at some time in the future.

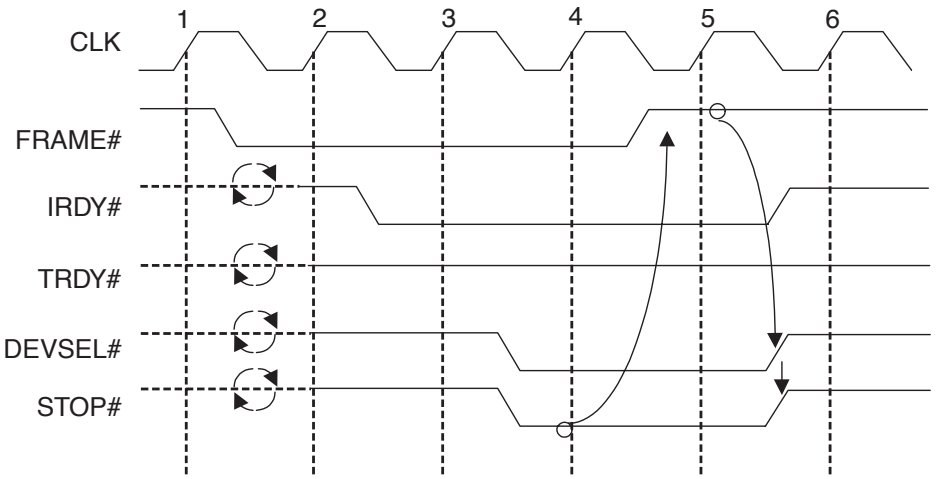


**Figure 3-6: Target retry.**

A common use for the target retry is the *delayed transaction*. A target that knows it can't meet the initial latency requirement can "memorize" the transaction by latching the address, command and byte enables and, if a write the write data. The latched transaction is called a *Delayed Request*. The target immediately issues a retry to the initiator and begins executing the transaction internally. This allows the bus to be used by other masters while the target is busy. Later, when the master retries the exact same transaction and the target has completed the transaction, the target replies appropriately. The result of completing a Delayed Request produces a *Delayed Completion* consisting of completion status and data if the request was a read. Bus bridges, particularly bridges to slower expansion buses like ISA, make extensive use of the delayed transaction.

Note that in order for the target to recognize a transaction as the retry of a previous transaction, the initiator must duplicate the transaction exactly. Specifically, the address, command and byte enables and, if a write the write data, must be the same as when the transaction was originally issued. Otherwise it looks like a new transaction to the target.

Typical targets can handle only one delayed transaction at a time. While a target is busy executing a delayed transaction, it must retry all other transaction requests without memorizing them until the current transaction completes.

Note that there is a possibility that another master may execute exactly the same transaction after the target has internally completed a delayed transaction but before the original initiator retries. The target can't distinguish between two masters issuing the exact same transaction so it replies to the second master with the Delayed Completion information. When the first master retries, it looks like a new transaction to the target and the process starts over.

What happens if a master never retries the transaction? Targets capable of executing delayed transactions must implement a *Discard Timer*. A target must discard a Delayed Completion if the master has not retried the transaction after $2^{32}$ clocks[1].

### Disconnect

The target may terminate a transaction with a Disconnect if it is unable to meet the maximum latency requirements. There are two possibilities—either the target is prepared to execute one last data phase or it is not. If TRDY# is asserted when STOP# is asserted, the target indicates that it is prepared to execute one last data phase. This is called a "Disconnect with data." There are two cases as shown in Figure 3-7: Disconnect-A and Disconnect-B. The only difference between the two is the state of IRDY# when STOP# is asserted. In the case of Disconnect-A, IRDY# is not asserted when STOP# is asserted. The initiator is thus notified that the next transfer will be the last. It deasserts FRAME# on the same clock that it asserts IRDY#.

In Disconnect-B, the final transfer occurs in the same clock when STOP# is sampled asserted. The initiator deasserts FRAME# but the rules require that IRDY# remain asserted for one more clock. To prevent another data transfer, the target must deassert TRDY#. In both cases, the target must not deassert DEVSEL# or STOP# until it detects FRAME# deasserted. The target may resume the transaction later at the point where it left off.

[1] At 33 MHz, this works out to about two minutes.

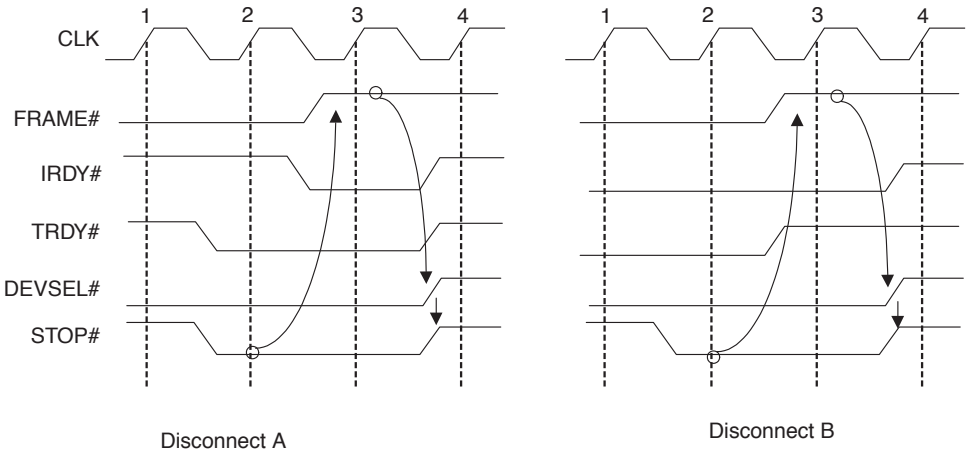


**Figure 3-7: Target disconnect—with data.**

If the target asserts STOP# when TRDY# is not asserted, it is telling the initiator that it is not prepared to execute another data phase. This is called a "Disconnect without data." The initiator responds by deasserting FRAME#. There are two possibilities: either IRDY# is asserted when STOP# is detected or it is not. In the latter case, the initiator must assert IRDY# in the clock cycle where it deasserts FRAME#. This is illustrated in Figure 3-8. Note that the Disconnect without data looks exactly like a Retry except that one or more data phases have completed.

### *Target Abort*

As shown in Figure 3-9, Target Abort is distinguished from the previous cases because DEVSEL# is not asserted at the time that STOP# is asserted. Also, unlike the previous cases where the initiator is invited (or required) to retry or resume the transaction, Target Abort specifically says do not retry this transaction. Target Abort typically means that the target has experienced some fatal error condition. The initiator should probably raise an exception back to its host. One or more data phases may have completed before the target signaled Target Abort.

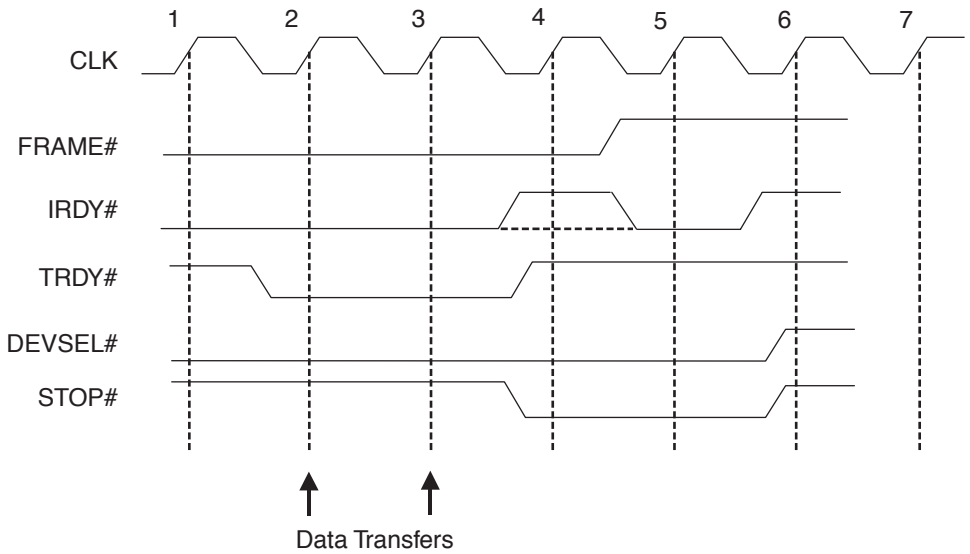


**Figure 3-8: Target disconnect—without data.**

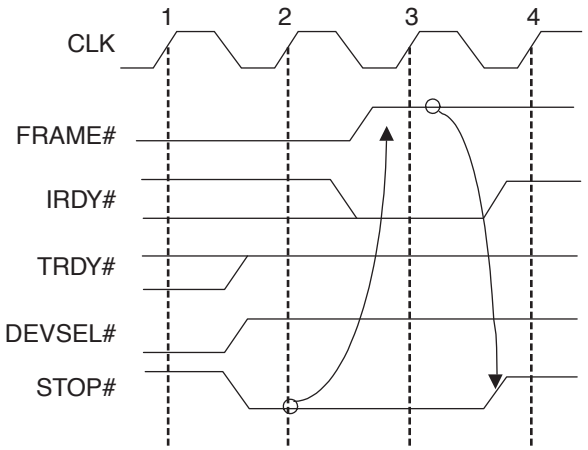


**Figure 3-9: Target abort.**

## Error Detection and Reporting

### Parity Generation and Detection—PAR and PERR#

All bus agents are required to generate *even* parity over the AD and C/BE# busses. The result of the parity calculation appears on the PAR line. Even parity means that the PAR line is set so that the number of bus lines in the logical 1 state, including PAR, is even. All 32 AD lines are always included in the parity calculation even if they are not being used in the current transaction. This is another reason why the driving agent must always drive all 32 AD lines.

With two minor exceptions, all agents are required to have the ability to check parity. The two exceptions are:

- Devices (i.e., silicon) designed exclusively for use on a motherboard.
- "Devices that never deal with, contain or access any data that represents permanent or residual system or application state, e.g., human interface and video/audio devices." In other words, it's not a big deal if a sound card drops a bit now and then.

The agent driving the AD bus during any clock phase computes even parity and places the result on the PAR line one clock cycle later. The receiving agent checks the parity and, upon detecting an error, may assert PERR#. So on a read transaction, PAR is driven by the target and PERR# is driven by the initiator. The target then senses PERR# and may take action if appropriate. On a write transaction, the opposite occurs.

Figure 3-10 illustrates the timing of parity generation and detection. The key point to note is that one clock cycle is required to generate parity and another is required to check it. Looking at it in more detail:

#### Clock

2 Address phase. The selected master places the target address and command on the bus. All targets latch this information.

3 Turnaround cycle for read transaction. The initiator places computed parity for the address phase on PAR.

4 If any agent has detected a parity error in the address phase it asserts SERR# here. This is the first read data phase and also a turnaround cycle for PAR.

5 Target places computed parity on PAR. Otherwise, this is an idle cycle.

6 Initiator reports any parity error here by asserting PERR#. This also happens to be the address phase for the next transaction.

Clocks 7 to 9 illustrate the same process for write transactions. Note that no turn-around is required on either AD or PAR.

Also note that because SERR# is open-drain, it may require more than one clock cycle to return to the non-asserted state.

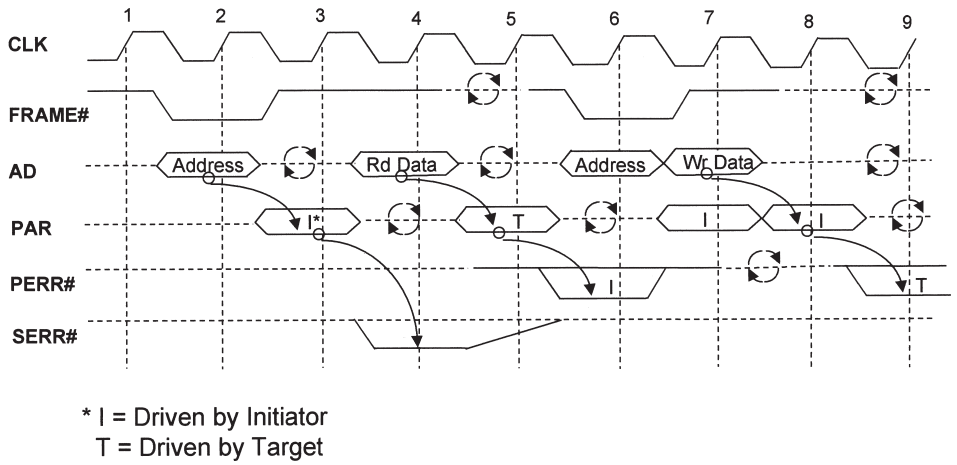


**Figure 3-10: Timing diagram for parity generation and detection.**

Upon detection of a parity error, the agent that is checking parity must set the DETECTED PARITY ERROR bit in its Configuration Status Register. If the - PARITY ERROR RESPONSE bit in its Configuration Command Register is a 1, then it asserts PERR#. Any error recovery strategies are the responsibility of the host attached to the agent that detects the error.

Although bus agents are required to generate parity, there is no requirement that they act on a detected parity error. The ability to detect parity errors and take action is controlled by bits in the device's Configuration Control Register.

### *System Errors—SERR#*

PERR# only reports parity errors during data phases. That is, it is intended to signal an error condition between a specific initiator/target pair. Parity is also generated and checked during the address phase. But if there is an error on the address bus, which target should check and report that error? The answer is they all should. Any target

that detects a parity error during the address phase asserts SERR# and sets the SIGNALED SYSTEM ERROR bit in its Status Register if the SERR# ENABLE bit in its Command Register is set. SERR# is an open-drain signal so it is permissible for more than one agent to assert it simultaneously.

An agent that "thinks" it has been selected in the presence of an address parity error can respond in one of three ways:

- Claim the transaction and proceed as if everything were okay
- Claim the transaction and terminate with Target-Abort
- Don't claim the transaction and let the initiator terminate with Master-Abort

SERR# is also used to signal parity errors on Special Cycles because, like the address phase, a Special Cycle is not directed at a specific target. It may also be used to signal other catastrophic error conditions.

The assertion of SERR# should be considered a fatal condition. The specification suggests that SERR# would most likely be handled as a nonmaskable interrupt.

## Summary

The PCI specification defines a precise set of rules, called a *protocol*, for how data is transferred across the bus. Every bus transaction consists of an address phase and one or more data phases. Both the initiator and the target of a transaction can regulate the flow of data by controlling their respective "ready" signals, IRDY# and TRDY#.

A transaction may be terminated by either the initiator or the target. One reason the target may terminate a transaction is because it is temporarily busy or unable to meet the initial latency requirements. In this case, it tells the initiator to "Retry" the transaction later.

All PCI agents are required to generate even parity on the AD and C/BE lines. With two exceptions, all agents are required to have the ability to check parity whether or not they choose to take any action in response to a detected parity error. Parity errors during data phases are reported on the PERR# line. The SERR# line is used to report parity errors during address phases and Special Cycle transactions. It can also be used to report other system errors. SERR# is considered to be a fatal condition.

The next chapter describes advanced features of the PCI protocol.